Use bookmarks or directly click on the titles to navigate.

4/8/02 Monday

Welcome and introduction	Si
Workshop goals and discussions	Werbos
Exploitng control knowledge in RL	Barto
Approximate solutions of pursuit evasion games involving UAVs & UGVs	Sastry
Adaptive dynamic programming for dynamic resource allocation problems	Powell
Linguistic communication between components of DHP?	Lendaris
Multiobjective control problem by RL	Bien
From perturbation analysis to Markov decision processes and RL	Cao
Dynamic programming problems in financial engineering	Sircar
MaxQ Hierarchical RL, value functions, action smoothing	Dietterich
Studies in neuro-fuzzy adaptive control via hybrid systems and ADP	Esogbue
Learning by PUNISH/REWARD and learning by supervision	Widrow

4/9/02 Tuesday	
Direct neural dynamic programming, a design-centric approach vs. prior results	Si
New directions in hierarchical RL: Concurrency, Multi-Agency, and Partial observability	Mahadevan
Reinforcement learning from limited observations	Ungar
The Linear Programming approach to ADP	Van Roy
Robust RL control with static and dynamic stability	Anderson
Robust solutions to Markov Decision Problems	El Ghaoui & Nilim
Algebraic and adaptive learning for heuristic neural control	Ferrari&Stengel
Backpropagation through time and and its relationship with derivative adaptive critics	Prokhorov

4/10/02 Wednesday	
What's beyond … for ACDs?	Wunsch
ADP for control of nonlinear and distributed parameter systems	Balakrishnan
Learning control of a class of failure avoidance problems with known analytically derived cost function	Liu
Excitation & turbine ACD based neurocontrol of multiple generators on the electric power grid	Venayagamoorthy
Power systems and optimization workshop	Momoh/Werbos/Harley/Amin
Conclusion, recommendation, plan	All

Backpropagation: General Principles and Issues for Biology

Paul J. Werbos

National Science Foundation*, Room 675

Arlington, VA 22203

pwerbos@nsf.gov

Backpropagation

703-292-8339

FAX 703-292-9147 (unreliable)

*The views herein are those of the author not those of NSF.

(Footnote required by our lawyers.)

INTRODUCTION

Like reinforcement learning, backpropagation has is a large collection of methods, mathematics and research areas which has become very fragmented as it has been propagated, popularized, interpreted, enhanced and implemented in different ways across a wide variety of application domains. There are two relatively standard definitions of backpropagation (Arbib, 1995):

o Backpropagation is a procedure for *efficiently* calculating the derivatives of some function of the outputs of any nonlinear differentiable system, with respect to all inputs and parameters of that system, through calculations proceeding *backwards* from outputs to inputs. It permits “local” implementation on parallel hardware (or wetware).

o Backpropagation is any technique for adapting the weights or parameters of a nonlinear system by using such derivatives or the equivalent.

One family of backpropagation methods, called vanilla backpropagation, accounts for most applications of artificial neural networks (ANNs) published in journals today. Yet in many challenging engineering applications – applications where computational intelligence is especially important to success – more advanced and powerful forms of backpropagation are essential; if applications in the pipeline are weighted by economic value added, rather than words published, these are probably more important than vanilla backpropagation. Finally, one may argue that some form of backpropagation would be essential to replicating (or understanding) some of the high-level information-processing capabilities of the mammalian brain (Pribram, 1998; Werbos, 1994); yet neuroscientists have rightly pointed out that vanilla backpropagation is too simple to be plausible even as one part of a model of functional circuit-level computation in the brain.

Backpropagation has been used to perform at least five different types of task, in computational intelligence. Suppose that $\underline{\mathbf{Y}}=\underline{\mathbf{f}}(\underline{\mathbf{X}},W)$ denotes any nonlinear input-output mapping, such as an ANN, which inputs a vector $\underline{\mathbf{X}}$, outputs a vector $\underline{\mathbf{Y}}$, and contains a set of tunable weights or parameters W.

Backpropagation has been used to perform:

1. Supervised learning, in which the system is given a *training set* of pairs of values for $\underline{\mathbf{X}}$(t) and $\underline{\mathbf{Y^*}}$(t), and tunes W so as to make $\underline{\mathbf{Y}}$(t) a good predictor of $\underline{\mathbf{X}}$(t). In some variations of this task, the pairs are presented one-at-a-time, while in others there is a fixed database.
2. Gradient learning, in which the system is given a training set of pairs of values for $\underline{\mathbf{X}}$(t) and $\nabla_Y U(\underline{\mathbf{Y}}(t),t)$, where U is some (unknown) function, and where W is tuned so as to maximize or minimize U over time.
3. Neuroidentification, which is like supervised learning, except that the nonlinear system to be adapted can be described as a system which outputs two vectors at each time t:

$$\underline{\mathbf{Y}}(t)=\underline{\mathbf{f}}_Y(\underline{\mathbf{X}}(t),W,\underline{\mathbf{R}}(t-1)) \tag{1a}$$

$$\underline{\mathbf{R}}(t)=\underline{\mathbf{f}}_R(\underline{\mathbf{X}}(t),W,\underline{\mathbf{R}}(t-1)) \tag{1b}$$

4. Probability distribution learning, which is like 1 or 3, except that the task is to accurately represent a probability distribution. This can be done by outputting $\underline{\mathbf{Y}}=\underline{\mathbf{f}}(\underline{\mathbf{X}},W,\underline{\mathbf{u}})$, where $\underline{\mathbf{u}}$ is a vector of random numbers, and seeking $\Pr(\underline{\mathbf{Y^*}}|\underline{\mathbf{X}})=\Pr(\underline{\mathbf{Y}}|\underline{\mathbf{X}})$.
5. Pass-through of derivatives, as in "backpropagation through a model," where $\nabla_Y(\underline{\mathbf{Y}}(t),t)$ is given and the task is to output $\nabla_X(\underline{\mathbf{Y}}(t),t)$.

This list is only slightly oversimplified. It is important to note that supervised learning is only one of five ways to use backpropagation. The design choices and tradeoffs, in performance and in software, vary greatly from area to area. Vanilla backpropagation, in turn, is only one of the many forms of backpropagation used in supervised learning.

Intelligent control systems, like biological brains, are not static input-output devices. Sophisticated decision-making capabilities over time require components which perform some of tasks 2-5 above. (White and Sofge, 1992; Narendra and Lewis, 2001; www.iamcm.org.) Supervised learning can be useful as a research test bed, in developing tools which could be extended to tasks 2-5. It has major value as a present market for ANNs. (See www.hnc.com and www.neuralware.com and www.nd.com, for examples, involving areas such as pattern recognition for OCR systems, financial risk assessment systems, and airport screening systems in the wake of 9/11, to pick just a few.) But its importance within the larger scheme of backpropagation and basic science should not be overstated.

This paper first reviews the chain rule for ordered derivatives, the original mathematical foundation for the use of backpropagation in all 5 areas. Next it reviews backpropagation for supervised learning, including vanilla backpropagation. Unfortunately, the introductory nature of this article prohibits discussion of the four other tasks, despite their greater importance both to engineering and to cognitive neuroscience; see the references and web pages cited herein for discussions of those larger tasks.

THE CHAIN RULE FOR ORDERED DERIVATIVES

The chain rule for ordered derivatives addresses the following task: given a nonlinear, distributed system which computes some output result R(**Y**), where **Y**=**f**(**X**,W) (or **Y**=**f**(**X**,W,**u**)), how can we efficiently compute *all* the derivatives of R with respect to all components of the inputs vector **X** and the weights in W?

There are two cases which commonly arise.

When there is a finite bound on the time it takes to compute **Y**, then in principle we can always represent the sequence as an ordered, "feedforward" sequence of computations of the form:

$$R = f_{N+1}(f_N,...,f_0) \qquad (2a)$$

$$Y_n=f_N(f_{N-1}, ..., f_0) \qquad (2b)$$

$$...$$

$$f_k=X_k \qquad (2c)$$

$$...$$

$$f_0=1 \qquad (2d)$$

where the initial nodes in the network just "load in" or register the inputs, **X** and W (and **u**), where each intermediate node is allowed to be *any* function of preceding nodes, where the last block of nodes but one represent the output vector, and where the result is treated as if it were a node in the network.

(When the distributed system is partially ordered, there are typically many equivalent representations of this form.) My 1974 Harvard PhD thesis (Werbos 1994) proved that one may simply compute, recursively:

$$\frac{\partial^+ R}{\partial f_i} = \sum_{k=i+1}^{n} \frac{\partial^+ R}{\partial f_k} \cdot \frac{\partial f_k}{\partial f_i}\,, \qquad (3)$$

where the partial derivative on the far right represents the *direct* impact of f_i on f_k in that equation (in equations 2) which determines f_k, and where the ordered derivatives (the derivatives with a "+" superscript) represent the *total* impact of a change in f_i on the result R. This system may be initialized by using the fact that $\partial^+R/\partial R=1$; however, we usually just assume that the terms $\partial^+R/\partial Y_i$ are available as a starting point. The ordered derivatives of R with respect to the inputs are what we need for the 5 tasks outlined above.

This chain rule was published in several papers between 1977 and 1983, particularly in a major conference paper (reprinted in Werbos 1994) which elaborated on the applications to layered neural networks, intelligent control, and various forms of sensitivity analysis. It also discussed forwards propagation or conventional perturbation, a method which is neither biologically plausible nor useful in large-scale engineering applications where computational cost is important. (Forwards propagation has been reinvented under different names many times.) Anderson and Rosenfeld (1998) published some of the history of backpropagation. See Werbos (1994) and White and Sofge (1992) for many examples and intuitive explanations of this chain rule. Rumelhart and McClelland (1986) played a central role in popularizing vanilla backpropagation and reviving the neural network field in general.

Note that the calculation of derivatives through a stochastic system is achieved most efficiently and most simply by representing $\underline{\mathbf{u}}$ as another set of exogenous inputs. Complex papers have described the use of complicated sampling methods as an alternative; however, from a statisticians' viewpoint, the former is like a paired comparison test, while the latter is like an unpaired comparison.

In social science, biology and engineering, one often encounters nonlinear systems defined implicitly, as a system of simultaneous equations. These can be written as:

$$\underline{\mathbf{Y}}=\underline{\mathbf{f}}_Y(\underline{\mathbf{X}},W,\underline{\mathbf{y}}) \qquad (4a)$$

$$\underline{\mathbf{y}} = \underline{\mathbf{f}}_y(\underline{\mathbf{X}},W,\underline{\mathbf{y}}), \qquad (4b)$$

where $\underline{\mathbf{f}}$ itself is a feedforward system which outputs both $\underline{\mathbf{Y}}$ and $\underline{\mathbf{y}}$. Equations 4 define a Simultaneous Recurrent Network (SRN). (Unfortunately, later authors used “SRN” to denote a “Simple Recurrent Network,” a related but distinct concept.) To calculate the derivatives of R($\underline{\mathbf{Y}}$) with respect to $\underline{\mathbf{X}}$ and W one must use a numerical estimation or iterative method. Several choices exist, all requiring use of backpropagation through $\underline{\mathbf{f}}$ as part of the larger calculations.

Simultaneous backpropagation is an efficient forwards-time method to calculate these derivatives in the general case. In 1987, I proved that simultaneous backpropagation converges to the correct derivatives, at least as quickly as the original system in equations 4 converges, *if* $\underline{\mathbf{y}}$ exactly solves equations 4. (White and Sofge, 1992, chapter 3, gives algorithm and citations.) It was first applied in 1981, to calculate the sensitivities of a Department of Energy model of the natural gas industry used in a major deregulation study. The methods of Pineda and Almeida are special cases of this general method.

Some psychologists use *simple truncation* to estimate derivatives of SRNs. (Indeed, they sometimes define Simple Recurrent Networks as networks adapted on that basis.) They perform one pass of backpropagation , and implicitly treat values of $\underline{\mathbf{y}}$ from the next-to-last iteration as if they were external constants. This does not yield correct derivatives. In training cellular SRNs to assist in maze navigation, Pang and I found that simple truncation led to useless results, while *backpropagation through time (BTT)* – though slow in our crude implementation – worked well. (See Tang and Chua 1999, and xxx.lanl.gov/abd/adap-org/9806001.) BTT gives *exact* derivatives, and requires only the same computational time required to converge the original system; therefore, it was better in guaranteeing performance in a computational experiment, even though it is not a plausible model of brain computation.

Because BTT is not biologically plausible, a forwards-time consistent approximator – the Error Critic (also discussed in adap-org 9806001) – merits further research. Preliminary empirical work by Danil Prokhorov at Ford suggests that the Error Critic might have superior performance in some control applications. Neither BTT nor the Error Critic require starting from a fully-converged value of $\underline{\mathbf{y}}$.

BACKPROPAGATION FOR SUPERVISED LEARNING

Basic and Vanilla Backpropagation: Principles

Backpropagation, in our general formulation, can be used in almost any nonlinear regression or statistical estimation. From a statistician's viewpoint, the supervised learning task over a fixed database (or "training set") is the same as the task of nonlinear regression, for which an enormous literature already existed when the ANN field was reborn in 1987, at the IEEE Intemational Conference on Neural Networks. But the traditional literature from statistics asked the user to pick a function **f** specifically based on prior information about the process which generates **Y***. The neural network field seeks to develop and use general-purpose functions **f** which can learn to approximate any well-defined nonlinear function and can be implemented as distributed systems made up of common, modular components ("neurons"). Computational speed and convergence were challenges in traditional nonlinear regression packages, even more than with ANNs.

Basic backpropagation uses a combination of backpropagation, steepest descent and least squares error to adapt a specific choice of **f** – the Generalized Multilayer Perceptron (GMLP). The GMLP may be defined by:

$$x_0 = 1 \qquad (5a)$$

$$x_i = X_i \qquad \text{(i=1 to m)} \qquad (5b)$$

$$\begin{cases} v_j = \sum_{k=0}^{j-1} W_{jk} x_k \\ x_j = s(v_j) \end{cases} \qquad \text{(k=m+1 to N+n)} \qquad (5c)$$

$$Y_i = x_{i+N} \qquad \text{(i=1 to n)} \qquad (5d)$$

where the components of the vectors **X** and **Y** are usually restricted to the interval from –1 to +1, and where the squashing or sigmoidal function s is normally chosen to be the hyperbolic tangent, tanh. The variables “x_i” are thought of as the scaled frequency output of a model neuron, and “v_j” is thought of as the voltage stimulating that neuron. The number of “hidden neurons,” N-m, must be chosen by the user; however, the weights W_{jk} can be adapted in a totally automatic way in principle. (Of course, when there is prior knowledge telling us that should zero out certain weights, it is easy enough to do so.)

Vanilla backpropagation is that special case of basic backpropagation where there are “three layers” – the input layer (x_i for $i \leq m$), the output layer (x_i for $i > N$), and the hidden layer (all neurons in-between) – and where all the weights are zeroed out apriori except for the weights connecting the input layer to the hidden layer, and the hidden layer to the output layer. Most papers published using backpropagation today use vanilla backpropagation, with intuitive modifications to steepest descent and least squares error as required to get decent learning performance. In vanilla backpropagation, as in basic backpropagation, the user only needs to decide how many hidden neurons to use, in order to invoke the system. The user must also develop the training set of $\underline{\mathbf{X}}$ and $\underline{\mathbf{Y}}$ pairs; this is essentially the same as the problem of choosing independent and dependent variables in a classical linear regression analysis. Companies which make money using vanilla backpropagation often say that 90 percent of their effort usually lies in constructing a good database. The use of vanilla backpropagation is similar to the use of linear regression, except that nonlinear relationships can be learned, and a larger training set is needed to make that really work.

Some researchers have explored an intermediate case, between GMLP and the 3-layer network, where the hidden neurons are divided into L sequential strings of neurons, called intermediate layers. Sontag, for example, proved that a 2-hidden-layer network can approximate certain nonlinear mappings encountered in direct inverse control more reliably than the vanilla design. DeFiguerido has published papers on research issues and methods for choosing L. Nauta’s description of the cerebellar system sounds similar to a multilayer feedforward design, but he discusses important additional loops in the system, like time-delayed recurrence in the Purkinje layer important to shaping action over time. (Nauta and Feirtag, 1986).

In applying the chain rule for ordered derivatives to equation 5, basic and vanilla backpropagation try to minimize the usual square error function well-known from statistics:

$$E(\underline{Y}*(t),\underline{Y}(t)) = \tfrac{1}{2}\sum_{i=1}^{n}(Y_i*(t)-Y_i(t))^2 \qquad (6)$$

To calculate the derivatives, we begin by doing a simple calculation and introducing a short-hand abbreviation for the ordered derivatives of interest:

$$F\_Y_i(t) = \frac{\partial^+ E}{\partial Y_i} = Y_i(t) - Y_i*(t) \qquad \text{i = 1 to n} \qquad (7)$$

The prefix “F_” may be thought of as “derivative Feedback back to....”. The chain rule for ordered derivatives yields the recurrence relations:

$$\begin{cases} F\_x_i(t) = F\_Y_{i-N}(t) + \sum_{j=i+1}^{N+n} W_{ji} F\_v_j(t) \\ F\_v_i(t) = s'(v_i) * F\_x_i(t) \end{cases} \qquad \text{i=N+n,......, m+1 or 1} \qquad (8)$$

$$F\_W_{ij}(t) = F\_v_i(t) * x_j(t) \qquad \text{all adapted weights ij} \qquad (9)$$

The asterisk signifies multiplication just in the last two equations, for clarity. F_Y_{i-N} is taken as zero for $i \le N$. In equation 8, we only need to do recurrence back to m+1 when we only need to know F_W_{ij}.
When we need to know the derivatives back to the inputs, F_X_i, we take the recurrence back to 1.

When adaptation is done one-observation-at-a-time, each weight W_{ij} is changed by -λ*F_W_{ij} at each time, where λ is a positive parameter called “the learning rate.” When adaptation is done in “batch” mode, over an entire database, F_W_{ij} is calculated by adding F_W_{ij}(t) over all observations t, and one usually uses a more powerful gradient based optimization method to adapt the weights. See Werbos (1994) for some pseudocode.

Performance Issues for Today’s User

A brief article cannot do full justice to the enormous literature on performance issues across all the many application domains. It could not even give full citations to all the texts considered definitive within their particular application domain or school of thought. The diverse literature suggests that no single individual – the author included – could properly critique the whole set of these texts in detail.

The user who wishes to use existing software might begin by downloading the SNNS shareware, from one of the many links listed in the pages of the Neural Network Society under www.ieee.org. The commercial websites cited above are also worth visiting.

Performance tradeoffs in data mining applications have been discussed in numerous papers in the Proceedings of the IEEE conference on Knowledge Discovery and Datamining (KDD). Mark Embrechts, for example, has done extensive studies on financial databases and drug-discovery types of databases.

In pattern recognition, Richard Duda – co-author of the earlier classic text on pattern recognition – has updated his text to include various ANN designs and tradeoffs. In the early 1990's, the US postal service funded extensive studies and evaluations of competing methods (neural and nonneural) for automated recognition of ZIP code digits. Recipients of such grants often announced that they had achieved the "best reported performance" (usually within the methods they tried), but Jay Lee of the postal service reported to this author that two systems using MLPs (one from AT&T) did best overall. Zip code segmentation, not digit recognition, is now the barrier to automation, and many of us believe that some sort of recurrent network is needed to overcome that barrier. Certainly MLPs and local networks and classic AI have not achieved adequate or human-like performance.

In chemical engineering, chapter 10 of White and Sofge (1992) discussed many applications and tricks then used, which are not so different from what is used today.

In engineering as a whole, the bulk of supervised learning applications use MLPs or "local networks" – usually, radial basis functions (RBFs), self-organizing maps (SOMs), CMAC, or even Adaptive Resonance Theory (ART) or modified ART. Naive, unmodified steepest descent with unscaled data can easily be used to achieve slow learning in all these systems, but modern software applied to reasonable databases generally does much better. With the best software, accuracy of learning and generalization is generally better with the MLP, when there is ample training data and more than a handful of inputs, but learning speed is generally better with local networks. Local networks also work better than naive MLPs in some diagnostic applications, when there are few "bad cases" in the training set;

however, more comprehensive diagnostic systems usually require a hybrid of approaches combined with time-series data. See www.iamcm.org for more discussion of applications and tradeoffs in engineering applications.

On the theoretical side – numerous theorems were proved long ago, showing that a host of nonlinear systems, ranging from fuzzy logic and Taylor series to all types of ANN, are universal approximations of well-behaved nonlinear functions. Barron of Yale and Sontag of Rutgers initiated a new stream of results in the 1990's, with larger practical applications. This paper cannot review that entire literature, but one of Barron's results was a particular watershed. He proved that the number of parameters required for a given level of approximation accuracy for smooth functions grows exponentially, with respect to the number of inputs, for ALL fixed linear basis networks, including Taylor series and local networks and traditional fuzzy logic. With MLPs, by contrast, the number of required parameters grows in a gentle polynomial fashion. Sontag showed that ratio-based approximators (not so useful in practical many-input networks) also have good approximation properties, and I would speculate that elastic fuzzy logic does also. But for now, MLPs clearly have the edge.

Standard texts by Haykin, Wasserman, Principe and others have been strongly recommended by various users for general applications. Fiesler and Beale (1997) provide extensive details on various approaches to MLPs.

DISCUSSION AND FUTURE RESEARCH

None of the existing systems used in supervised learning (ANN or other) replicates the brain's ability to combine one-trial learning and powerful generalization. Good theoretical approaches exist which should be able to close that gap, but more research is needed – especially cross-disciplinary research, developing designs which *combine* the strengths of different approaches. Among the many relevant sources are books by Vapnik, by Roychowdhury et al, Tikhonov, Trafalis, chapter 10 of White and Sofge (1992), and even chapters of Pribram (1998) discussing the interplay of short-term and long-term memory in memory-based learning. Recent work by Sejnowski on cortical learning is important to the latter. Also, Phatak has studied error functions with penalty functions that provide fault-tolerance and brain-like redundancy in addition to addressing learning speed and generalization.

Equally important to many engineering applications (such as diagnostic analysis and decision-making for large networks) is the use of supervised learning when the inputs or outputs form relational networks rather than vectors. See the discussion of ObjectNets in www.iamcm.org. Cellular SRNs are a special case of ObjectNets, relevant to tasks like image segmentation, maze navigation and the classification of connectedness, a classical challenge posed by Minsky and Papert (1969) which cannot be met by MLPs. Chua's cellular neural net chips – fabricated in several countries – may well provide many times more throughput than any other general-purpose electronic chip.

REFERENCES

Anderson, J., and E.Rosenfeld, eds, 1998, Talking Nets, Cambridge, MA: MIT Press.

Arbib, Michael A. ed , 1995, The Handbook of Brain Theory and Neural Networks, Cambridge, MA: MIT Press.

Fiesler, E. and Beale, R. eds., 1997, Handbook of Neural Comp utation, New York: Oxford University Press.

Minsky, M. and Papert, S., 1969, Perceptrons: An Introduction to Computational Geometry, Cambridge, MA: MIT Press.

Narendra, K. and Lewis, F., eds, 2001, Special Issue on Neural Network Feedback Control, Automatica, Vol. 37, No. 8, August 2001.

auta, W. and Feirtag, M., 1986, Fundamental Neuro-anatomy, W.H.Freeman.

*Pribram, Karl H., ed., 1998, Brain and Values, Hillsdale, NJ: Erlbaum.

Rumelhart, D. and McClelland, J. (1986) *Parallel Distributed Processing*. Cambridge, MA: MIT Press.

*Werbos, P., 1994, The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting, New York: Wiley.

*White, D. and Sofge, D., eds, 1992, Handbook of Intelligent Control, Van Nostrand.

Yang, T. and Chua, L.O., 1999, Implementing back-propagation-through-time learning algorithm using cellular neural networks, Int'l J. Bifurcation and Chaos, 9:1041-1074.

CDC02-INV1801

Neural Networks for Control: Research Opportunities and Recent Developments

Paul J. Werbos
National Science Foundation*, Room 675
Arlington, VA 22203
pwerbos@nsf.gov

Abstract

Artificial neural networks offer both a challenge to control theory and some ways to help meet that challenge. We need new efforts/proposals from control theorists and others to make progress towards the key long-term challenge: to design generic families of intelligent controllers such that one system (like the mammal brain) has a general-purpose ability to adapt to a wide variety of large nonlinear stochastic environments, and learn a strategy of action to maximize some measure of utility across time. New results in nonlinear function approximation and approximate dynamic programming put this goal in sight, but many parallel efforts will be needed to get there. Concepts from optimal control, adaptive control and robust control need to be unified more effectively (as has been done in some of the recent work on stability).

The Challenge to Researchers: Context and Motivation

From the view of an NSF Program Director*, neural network control is, first and foremost, a crucial challenge to the research community. The ECS Division has long been seeking more proposals – especially cross-disciplinary proposals, well grounded in control theory – which can rise more effectively to this challenge.

Somehow or other, we know that the smallest mammal brain achieves a high degree of competence in learning to perform very complex, novel tasks in a highly nonlinear environment fraught with all kinds of uncertainties. It does so in a general-purpose way, without the use of formal symbolic logic (except in one or two species, in some situations). The effort to understand how this could be possible is one of the key challenges to basic mathematical science in this century.

Neuroscience is unlikely to answer these questions without some sort of cross-disciplinary collaborations. A well-known neuroscientist once stated: ”I have asked myself what would have happened if we had used our methods to try to understand how a radio works. First we would pull out a capacitor, watch the radio whine, and publish a paper announcing the discovery of ‘the whine center.’ Then, on a new grant, we would buy a new radio, pull out a resistor, and announce ‘the buzz center.’ A thousand radios later, we would have a complete map of the functional centers of the radio...” Some of the more modern methods may be more like doing a spectral analysis of the radiation emitted by a CPU, when the PC is in various states, like boot-up, idle, word-processing and so on.

Many neuroscientists have reached out to the system dynamics community or the physics community, in search of ideas to guide the development of mathematical models. There is growing interest in “complex adaptive systems,” not only in biology, but in engineering areas of growing national attention, such as management of critical infrastructures and the “system of systems” in the wake of 9/11.

In the end, any effort to reverse-engineer or understand higher capabilities of the brain in serious mathematical terms requires some specification of what kinds of capabilities we are looking for. We need an operational definition of what we are aiming at. With an appropriate definition, it should be obvious to many people in the control community both that the problem is very challenging, and that the CDC community has a critical role to play in meeting the challenge. The next subsection will propose an operational definition.

In general, the electrical engineering community faces major challenges in attracting and retaining the best graduate students, who (like Congress) need to be convinced that new work in this area can be exciting and of fundamental importance both to scientific understanding and to the emerging needs of humanity. Facing up to these challenges will be important to the health of the profession.

A Specific Challenge and Associated Issues

There are many debates [1,2] about the exact nature of higher intelligence in the mammal brain. Those debates clearly go well beyond engineering. However, consider the following concept or challenge to engineering: to develop a family of intelligent control designs, such that any member of the family has a general purpose ability to learn the "optimal" strategy of action in any "well-behaved" complex nonlinear stochastic environment, when the system is given only three specific pieces of information: (1) a vector of observations $\underline{\mathbf{y}}$(t) at each sampling time t; (2) a vector of controls $\underline{\mathbf{u}}$(t) which it decides on itself; (3) the utility function U($\underline{\mathbf{y}}$) whose expected value of future time the system tries to maximize. The challenge is to achieve all this, in a design which fits at least the major gross hardware constraint we know that the brain does meet: real-time operation implemented in a highly parallel distributed "computer" made up of billions of *relatively* simple, modular processing elements ("neurons").

Even though the brain is not an exact utility maximizer, it is clear that this does capture much of what we see in the higher levels of intelligence in the mammal brain[1,2]; the brain does have capabilities which are hard to believe, if one looks out from the perspective of today's technology, yet it proves that capabilit ies of this sort do exist.

Before discussing the strategy of how to reach this goal, we need to examine the goal itself in more detail.

First, as a general matter, when I try to evaluate the potential impact of an effort in this area, I ask myself: "What *difference* would this particular work make to the expected delay time between now and the time when we really meet the full challenge?" The best effort will usually not be an effort aimed at reaching the final goal in one easy step. That is impossible. There are many parallel efforts possible, which represent *just one big step* beyond the present state of the art, providing pieces of what we will need to achieve the ultimate goal. Much of what we really need today is new general-purpose mathematics, applicable to nonlinear systems in general – *including* artificial neural networks (ANNs) as a special case, but not limited to them. Much of the best work in ANNs has actually been using neural networks as a context for developing that kind of more general mathematics. Indeed, backpropagation itself – the most widely used algorithm in the ANN field – is actually a more general mathematical algorithm[3].

Second, we need to think about the role of prior information and domain -dependent knowledge. There have been many extreme polarized debates, in the past, between people who believe in learning or data-driven approaches, versus those who believe in genetically-determined ideas or prior knowledge. Both in engineering and in neuroscience, the extreme positions are untenable, in my view. In the most challenging applications, the ideal strategy may be to look for a learning system as powerful as possible, a system able to converge to the optimal strategy without any prior knowledge at all – and then initialize *that* system to an initial strategy and model as close as possible to the most extensive prior knowledge we can find. (See also [4, foreword].) Some research is needed to get the best possible results in learning "without cheating." In each application domain, research is also needed to find out how to "cheat" most effectively. The first kind of research is most important to fundamental scientific progress, but the second is also needed as part of the effort to deliver products of importance to the needs of society. In biology, many people have argued that cells to do edge-detection, for example, appear very early in the life of an organism; yet researchers have shown that cells in the lateral part of the brain can learn to take over as edge detectors, after damage to the usual visual areas. Powerful learning and prior information are both needed. But for higher intelligence, we are looking more for the ability to learn and adapt in a general-purpose way.

Third, we cannot expect the brain *or any other physical device* to guarantee an exact optimal strategy of action in the general case. That is too hard for any physically realizable system. We will probably never be able to build a device to play a perfect game of chess or a perfect game of Go. In computer science terms, those problems are all "NP hard." But in engineering and in biology, we do not need or ask for absolutely perfect solutions. We look for the best possible approximations, trying to be as exact as we can, but not giving up on the true nonlinear problems of real interest.

Fourth, the notion of "well-behaved" is extremely subtle, and itself points towards one of the parallel strands of research that needs to be taken further. Decades ago, statisticians realized that it is impossible to learn very much from streams of time -series data, if there are billions of variables, and if one imposes the usual "flat priors" of maximum-likelihood statistics. Even simple ANNs are possible only because there are some implicit notions of "Occam's Razor" priors which allow inference, both in brains and in ANNs. Almost all theorems about nonlinear function approximators make similar implicit assumptions about the "smoothness" of the function to be approximated; there are some control applications where the usual notions about smoothness break down, and the usual nonlinear function approximators perform very badly, compared to others less well-known. Issues of this kind need to be explored further [3; 4, chapter 10].

Fifth, the issue of stability and safety is subsumed here into the choice of utility function U, in this formulation. When the world is modeled, mathematically, as the truly uncertain place it really is, we can never give a 100% guarantee that bad events are absolutely impossible. The brain was evolved to *minimize* the probability of sudden death, in an environment where an absolute guarantee cannot be achieved. Many practical users of control systems would rather be certain that the probability of accidents is minimized, in a full-up stochastic simulation of the real world, rather than having iron-clad guarantees that accidents could never happen if only the world were simple and linear. Stability theory will be an important tool in developing learning systems which can actually converge to strategies which minimize the probability of accidents, and it will be important to our ability to obtain and understand experience in using learning-based designs on complex real-world systems. But it is only one of several important strands of research, relevant to the larger goal. At higher levels of systems design and management, the President's Economic Adviser has recently urged engineers to place more weight on performance issues, and to address the tradeoffs between performance and safety in a more balanced way, grounded in modern risk analysis (i.e. in the maximization of total expected utility[5]).

Sixth, I would agree with the classical AI researchers who argue that the highest levels of intelligence seen in brains on earth is intelligence based on *symbolic reasoning*, not the subsymbolic intelligence I am talking about here. But 99% of the human brain is identical in its underlying wiring and learning abilities to the brains of the smallest mouse. Before science is able to truly understand how symbolic reasoning works in the human brain, it must first develop a deeper understanding of the remaining 99% of the brain. From a larger viewpoint, it is a good thing that many people do research on symbolic reasoning, even before scientific closure is possible on those issues; however, research aimed at subsymbolic intelligence is clearly on the critical path to developing deeper understanding of such higher levels of intelligence.

Strategies, Tasks and Tools

Most CDC members will immediately see that the challenge above is a challenge in optimal control. It may seem, at first, that the challenge here is simply the old challenge of "solving the curse of dimensionality" in dynamic programming. But it is more than that. The challenge is also to learn the model of the environment and to solve the dynamic programming problem as accurately as possible, concurrently. (Some computer scientists advocate a purely model-free approach, without any learning of how to predict or even do state estimation; this does not scale well to large problems, and is not consistent with what we know about brains or animal learning [2,4,6,7,8].)

It is also well-known that the general nonlinear robust control problem is equivalent to the problem of solving a nonlinear "Hamilton-Jacobi-Bellman" equation as accurately as possible. If one allows off-line learning, then the challenge posed above is equivalent to the challenge of giving nonlinear robust control the tools that it needs to address the general nonlinear case as accurately as possible. Many of the near-term opportunities to achieve practical results with neural network control do involve a clever use of off-line learning, in part because of verification and validation issues [9,10]. We may be entering a period where the difference between nonlinear robust control and neural network control may start to become more semantic and emotional rather than real and mathematical.

Adaptive control is relevant because of the need for systems based on learning or adaptation. Curiously enough, it now seems that methods derived from neural network control may finally solve the old problem of universal stability in adaptive control for the linear MIMO case; however, even the preliminary theorems on those lines [7] make heavy use of quadratic stability concepts from the linear robust control world. Greater collaboration between experts in linear robust control and adaptive control may be necessary to grasp this new opportunity, close at hand as it is. Clearly this is one of several very important open research opportunities.

The greater challenge here clearly depends on our ability to bring together the capabilities of all three of these communities more effectively.

The most important breakthrough which makes this a viable direction for research is the development of the field which some people now call neuro-dynamic programming[11] or, more recently, Adaptive Dynamic Programming (ADP). ADP originated in three previously-independent small strands of research led by Bernard Widrow ("adaptive critics"), Andrew Barron ("reinforcement learning") and myself ("approximate dynamic programming" and "reinforcement learning"), in the first major workshop on neural networks for control held back in 1990[12]. (See [7] for a review of the actual history, and for mathematical details of new adaptation methods important to strong stability.) The 1981 international conference paper which first described backpropagation in detail as a method for adapting multilayer neural networks also gave the general form of the method, for arbitrary nonlinear systems, and described how to use it as part of a parallel distributed design for model-based ADP [15,chapter 7].

Since then, however, the various schools have drifted apart to some degree. Reinforcement learning methods have become amazingly popular in AI, where they are commonly regarded as "the answer" to higher-level decision-making and planning problems. Yet the simple model-free designs in general use do not really address continuous variable problems, and have difficulties in scaling up to large problems, as has been noted many times in engineering applications in the past[4,12,13]. Even their performance in game-playing applications has been somewhat overstated; the only researcher who has ever achieved human expert-level performance in a difficult strategic game, based on learning without heavy prior knowledge, actually used evolutionary computing to train the "Critic" network in his system [14]. Clearly engineers have a critical role to play in developing designs which can scale up to handling larger problems, and can address the issue of partially-observed systems, by combining learning-based system identification and ADP together. This has already begun, but considerably more work remains to be done.

In the CDC talk, the author will mention some of the recent progress in model-based ADP work, and the further research challenges important to areas like the control of complex network systems like electric power grids [6]. Particularly notable are the recent success of Wunsch, Harley and Venayagamoorthy in controlling a physical network of turbogenerators able to maintain robust operation in the face of disturbances much greater than the previous state of the art allowed; the success of Balakrishnan in benchmark evaluations of success in difficult missile interception problems; success by Ferrari and Stengel in improving performance over well-tuned classical methods in aircraft control; and success by Lendaris' group at Portland State in tasks ranging from simulated vehicle skid control through to logistics control – all using model-based ADP methods. Major new results in stability have also been achieved, some by presenters in this session, and some by the Hittle/Young/ Anderson group at Colorado State (with application to improved energy efficiency in buildings), among others. See the references for mathematics, algorithms, and further citations.

References

1. Karl H.Pribram, ed. , *Brain and Values*, Erlbaum: Hillsdale, NJ, 1998.
2. K. Yasue, M. Jibu & T. Della Senta, eds, *No Matter, Never Mind : Proceedings of Toward a Science of Consciousness : Fundamental Approaches (Tokyo '99)* . John Benjamins Pub Co, 2002.
3. P. Werbos, Backpropagation: General principles and issues for biology. In M. Arbib, ed., *Handbook of Brain Theory and Neural Networks*, Second Edition, MIT Press, 2002.
4. White & D.Sofge, eds, *Handbook of Intelligent Control*, Van Nostrand, 1992.
5. H. Raiffa, *Decision Analysis*, Addison-Wesley, 1968.
6. wwwimacm.org
7. xxx.lanl.gov/abs/adap-org/9810001
8. P. Werbos, Neurocontrollers, in J.Webster, ed, *Encyclopedia of Electrical and Electronics Engineering*, Wiley, 1999
9. J.S. Baras and N.S.Patel, Information state for robust control of set-valued discrete time systems, *Proc. 34th Conf. Decision and Control (CDC)*, IEEE, 1995. p.2302.
10. M. Motter, ed, Special Session on Intelligent Flight Control, *ACC Conf. Proc.*, 2001/
11. D.P.Bertsekas and J.N.Tsisiklis, *Neuro-Dynamic Programming,*. Belmont, Mass.: Athena Scientific, 1996
12. W.T.Miller, R.Sutton & P.Werbos (eds), *Neural Networks for Control*, MIT Press, 1990, now in paper

13. S. Haykin, *Neural Networks: A Comprehensive Foundation*, Second Edition, Prentice-Hall, 1998.
14. K. Chellapilla & D.B. Fogel, Anaconda defeats Hoyle 6-0: A case study competing an evolved checkers program against commercially available software. *In Proc. Cong. on Evolutionary Computation (CEC2000*), IEEE Press, Piscataway, NJ, 2000, pp. 857-863.
15. P. Werbos, *The Roots of Backpropagation*, Wiley, 1994. Contains complete reprints.

Exploiting Control Knowledge in Reinforcement Learning

Andrew G. Barto
(barto@cs.umass.edu)

Abstract

Reinforcement learning (RL) algorithms have been applied to many problems, but in most applications the learning system interacts with a simulated environment rather than with a real, physical environment. While there many sound reasons for doing simulation-based RL, the original motivation for RL was to emulate the ability of animals to learn from real experiences in real environments. Working against this goal is the fact that in complex environments, RL algorithms tend to need large amounts of experience, and that behavior during learning may violate reasonable constraints that ensure safety and durability of the physical apparatus. In this talk, I describe several approaches for taking advantage of more orthodox control methods to both reduce the complexity of learning problems and to provide acceptable system behavior during learning. Both of these approaches take advantage of the idea that any of an RL system's action choices can trigger temporally-extended courses of action that terminate after variable periods of time. This idea, formalized by Sutton, Precup, and Singh's notion of an "option", provides a principled way to integrate conventional control knowledge into an RL system. The RL system's action vocabulary can include options for turning on controllers that were designed by conventional means.

The first application of this idea that I describe is extremely simple but can be very powerful in extending the utility of RL. Here, the objective is to drive a nonlinear system to a target state in minimum time. A standard closed-loop controller is designed based on a local linearization of the system about the target state. This controller is guaranteed to drive the system to the desired target state from any state within some neighborhood of the target and to stabilize it there. The largest such neighborhood is called the stability region of the controller. If an RL system takes the option of turning on the local controller for any state within this stability region, then its task becomes one of learning to hit the stability region in minimum time. Depending on the size of the stability region, this can be a much easier learning task. Unfortunately, it is usually not easy to specify a local controller's stability region. We use an overly-large approximation of the stability region and show how the RL system can learn to avoid those subregions that are not in the actual stability region. I illustrate this method using a simulated double pendulum and a simple chaotic control problem. I also point out that we see the pervasive presence of analogs of this approach when we examine animal behavior.

The second method for integrating orthodox control methods with RL uses knowledge in the form of Lyapunov functions. By constraining an RL controller to select actions that always cause the system state to descend on a Lyapunov function, performance during learning can be significantly enhanced and certain performance guarantees can be provided. Stable control is ensured while the RL system learns how to improve performance with respect to an optimal control criterion. Applications of this approach require designing Lyapunov-descending options for the RL system, which learns when to apply them.

Although these approaches are relatively simple, we think that these and related ideas for integrating RL with control knowledge are essential in moving toward real embedded RL applications.

Heuristic Search in Infinite State Spaces Guided by Lyapunov Analysis

Theodore J. Perkins and **Andrew G. Barto**
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
{perkins,barto}@cs.umass.edu

Abstract

In infinite state spaces, many standard heuristic search algorithms do not terminate if the problem is unsolvable. Under some conditions, they can fail to terminate even when there are solutions. We show how techniques from control theory, in particular Lyapunov stability analysis, can be employed to prove the existence of solution paths and provide guarantees that search algorithms will find those solutions. We study both optimal search algorithms, such as A*, and suboptimal/real-time search methods. A Lyapunov framework is useful for analyzing infinite-state search problems, and provides guidance for formulating search problems so that they become tractable for heuristic search. We illustrate these ideas with experiments using a simulated robot arm.

1 Introduction

As the boundaries become less distinct between artificial intelligence and fields more reliant on continuous mathematics, such as control engineering, it is being recognized that heuristic search methods can play useful roles when applied to problems with infinite state spaces (e.g., Boone [1997], Davies *et al.* [1998]). However, the theoretical properties of heuristic search algorithms differ greatly depending on whether the state space is finite or infinite.

For finite state space problems, a variety of well-understood algorithms are available to suit different needs. For example, the A* algorithm finds optimal solutions when they exist, and is also "optimally efficient"—no other heuristic search algorithm has better worst-case complexity. Variants, such as IDA*, allow more memory-efficient search at the cost of greater time complexity. Conversely, suboptimal search methods, such as depth-first search or best-first search with an inadmissible heuristic, can often produce some solution, typically suboptimal, more quickly than A* can find an optimal solution [Pearl, 1984; Russell and Norvig, 1995]. The RTA* algorithm is able to choose actions in real-time, while still guaranteeing eventual arrival at a goal state [Korf, 1990]. However, if the state space is infinite, none of these algorithms is guaranteed to have the same properties. A* is complete only if additional conditions hold on the costs of search operators, and it does not terminate if the problem admits no solution [Pearl, 1984]. Suboptimal search methods may not terminate, even if a closed list is maintained. RTA* is not guaranteed to construct a path to a goal state [Korf, 1990].

A further difficulty when infinite state spaces are considered is that the most natural problem formulations often include infinite action spaces. To apply heuristic search, one must select a finite subset of these actions to be explored in any given state. In doing so, the possibility arises that an otherwise reachable goal set becomes unreachable.

Some of these difficulties are unavoidable in general. With an infinite number of possible states, a search problem may encode the workings of a Turing machine, including its memory tape. Thus, the question of whether or not an infinite-state search problem has a solution is in general undecidable. However, it is possible to address these difficulties for useful subclasses of problems. In this paper we examine how Lyapunov analysis methods can help address these difficulties when it is applicable. A Lyapunov analysis of a search problem relies on domain knowledge taking the form of a Lyapunov function. The existence of a Lyapunov function guarantees that some solution to the search problem exists. Further, it can be used to prove that various search algorithms will succeed in finding a solution. We study two search algorithms in detail: A* and a simple iterative, real-time method that incrementally constructs a solution path. Our main goal is to show how Lyapunov methods can help one analyze and/or formulate infinite-state search problems so that standard heuristic search algorithms are applicable and can find solutions.

The paper is organized as follows. In Section 2 we define heuristic search problems. Section 3 covers some basics of Lyapunov theory. Sections 4 and 5 describe how Lyapunov domain knowledge can be applied to prove that a search algorithm will find a solution path. The relationship between Lyapunov functions and heuristic evaluation functions is also discussed. Section 6 contains a demonstration of these ideas on a control problem for a simulated robot arm. Section 7 concludes.

2 State Space Search Problems

Definition 1 *A state space search problem (SSP) is a tuple* $(S, G, s_0, \{O_1, \ldots, O_k\})$*, where:*

- *S is the state set. We allow this to be an arbitrary set.*
- *$G \subset S$ is the set of goal states.*
- *$s_0 \notin G$ is the initial, or start, state.*
- *$\{O_1, \dots, O_k\}$ is a set of search operators. Some search operators may not be applicable in some states. When a search operator O_j is applied to a state $s \notin G$, it results in a new state $Succ_j(s)$ and incurs a cost $c_j(s) \geq 0$.*

A solution to an SSP is a sequence of search operators that, when applied starting at s_0, results in some state in G. An optimal solution to an SSP is a solution for which the total (summed) cost of the search operators is no greater than the total cost of any other solution.

For infinite state spaces, the infimal cost over all solution paths may not be attained by any path. Thus, solutions may exist without there being any optimal solutions. This possibility is ruled out if there is a universal lower bound $c_{low} > 0$ on the cost incurred by any search operator in any non-goal state [Pearl, 1984]. We will use the phrase "the SSP's costs are bounded above zero" to refer to this property.

Note that our definition of an SSP includes a finite set of search operators $\{O_1, \dots, O_k\}$, some of which may be unavailable in some states. We have made this choice purely for reasons of notational convenience. The theory we present generalizes immediately to the case in which there is an infinite number of search operators, but the number of operators applicable to any particular non-goal state is finite.

3 Control Lyapunov Functions

Lyapunov methods originated in the study of the stability of systems of differential equations. These methods were borrowed and extended by control theorists. The techniques of Lyapunov analysis are now a fundamental component of control theory and are widely used for the analysis and design of control systems [e.g., Vincent and Grantham 1997].

Lyapunov methods are tools for trying to identify a Lyapunov function for a control problem. In search terms, a Lyapunov function is most easily understood as a descent function (the opposite of a hill-climbing function) with no local minima except at goal states.

Definition 2 *Given an SSP, a control Lyapunov function (CLF) is a function $L : S \mapsto \Re$ with the following properties:*

1. *$L(s) \geq 0$ for all $s \in S$.*
2. *There exists $\delta > 0$ such that for all $s \notin G$ there is some search operator O_j such that $L(s) - L(Succ_j(s)) \geq \delta$.*

The second property asserts that at any non-goal state some search operator leads to a state at least δ down on the CLF. Since a CLF is non-negative, a descent procedure cannot continue indefinitely. Eventually it must reach a state where a δ step down on L is impossible; such a state can only be a goal state.

A CLF is a strong form of domain knowledge, but the benefits of knowing a CLF for a search problem are correspondingly strong. The existence of solutions is guaranteed, and (suboptimal) solutions can be constructed trivially. Numerous sources discuss methods for finding Lyapunov functions (e.g., Vincent and Grantham [1997], Krstić *et al.* [1995]). For many important problems and classes of problems, standard CLFs have already been developed. For example, linear and feedback linearizable systems are easily analyzed by Lyapunov methods. These systems include almost all modern industrial robots [Vincent and Grantham, 1997]. Path planning problems similarly yield to Lyapunov methods [Connolly and Grupen, 1993]. Many other stabilization and control problems, less typically studied in AI, have also been addressed by Lyapunov means, including: attitude control of ships, airplanes, and spacecraft; regulation of electrical circuits and engines; magnetic levitation; stability of networks or queueing systems; and chemical process control (see Levine [1996] for references). Lyapunov methods are relevant to many important and interesting applications.

The definition of a CLF requires that at least one search operator leads down by δ in any non-goal state. A stronger condition is that all search operators descend on the CLF:

Definition 3 *Given an SSP, the set of search operators descends on a CLF L if for any $s \notin G$ there exists at least one applicable search operator, and every applicable search operator O_j satisfies $L(s) - L(Succ_j(s)) \geq \delta$ for some fixed $\delta > 0$.*

4 CLFs and A*

In this section we establish two sets of conditions under which A* is guaranteed to find an optimal solution path in an SSP with an infinite state space. We then discuss several other search algorithms, and the relationship of CLFs to heuristic evaluation functions.

Theorem 1 *If there exists a CLF L for a given SSP, and either of the following conditions are true:*

1. *the set of search operators descends on L, or*
2. *the SSP's costs are bounded above zero,*

then A search will terminate, finding an optimal solution path from s_0 to G.*

Proof: Under condition 1, there are only a finite number of non-goal states reachable from a given start state s_0. The only way for an infinite number of states to be reachable is for them to occur at arbitrarily large depths in the search tree. But since every search operator results in a state at least δ lower on L and $L \geq 0$ everywhere, no state can be at a depth greater than $\lceil L(s_0)/\delta \rceil$. Since the CLF implies the existence of at least one solution and there are only a finite number of reachable states, it follows (e.g., from Pearl [1984], section 3.1), that A* must terminate and return an optimal solution path.

Alternatively, suppose conditions 2 holds. The CLF ensures the existence of at least one solution path. Let this path have cost C. Since each search operator incurs at least c_{low} cost, the f-value of a node at depth d is at least $d \cdot c_{low}$. A* will not expand a node with f-value higher than C. Thus, no node at depth $d > \lceil C/c_{low} \rceil$ will ever be expanded. This means A* must terminate and return an optimal solution. QED.

These results extend immediately to variants of A* such as uniform cost search or IDA*. Under condition 1, not only is the number of reachable states finite, but these states form an acyclic directed graph and all "leaves" of the graph are goal states. Thus, a great many search algorithms can safely be applied in this case. Under condition 2, depth-first branch-and-bound search is also guaranteed to terminate and return an optimal solution, if it is initialized with the bounding cost C. This follows from nearly the same reasoning as the proof for A*.

How do CLFs relate to heuristic evaluation functions? In general, a CLF seems a good candidate for a heuristic function. It has the very nice property that it can be strictly monotonically decreased as one approaches the goal. Contrast this, for example, to the Manhattan distance heuristic for the 8-puzzle [Russell and Norvig, 1995]. An 8-puzzle solution path typically goes up and down on the Manhattan distance heuristic a number of times, even though the ultimate effect is to get the heuristic down to a value of zero.

However, a CLF might easily fail to be admissible, overestimating the optimal cost-to-goal from some non-goal state. Notice that the definition of a CLF does not depend on the costs of search operators of an SSP. It depends only on S, G, and the successor function. CLFs capture information about the connectivity of the state space more than the costs of search operators.

A possible "fix" to the inadmissibility of a CLF is to scale it. If a CLF is multiplied by any positive scalar, defining a new function $L' = \alpha L$, then L' is also a CLF. If a given CLF overestimates the cost-to-goal for an SSP, then it is possible that a scaled-down CLF would not overestimate. We will not elaborate on the possibility of scaling CLFs to create admissible heuristics in this paper. We will, however, use the scalability of a CLF in the next section, where we discuss real-time search.

5 Real-Time Search

Lyapunov domain knowledge is especially well suited to real-time search applications. As mentioned before, one can rapidly generate a solution path by a simple descent procedure. A generalization of this would be to construct a path by performing a depth-limited search at each step to select the best search operator to apply next. "Best" would mean the search operator leading to a leaf for which the cost-so-far plus a heuristic evaluation is lowest. We will call this algorithm "repeated fixed-depth search" or RFDS.

Korf [1990] described this procedure in a paper on real-time search and, rightly, dismissed it because in general it is not guaranteed to produce a path to a goal state, even in finite state spaces. However, the procedure is appealing in its simplicity and appropriate for real-time search in that the search depth can be set to respond to deadlines for the selection of search operators. In this section we examine conditions under which this procedure can be guaranteed to construct a path to a goal state.

Theorem 2 *Given an SSP and a CLF, L, for that SSP, if the set of search operators descends on L then RFDS with any depth limit $d \geq 1$ and any heuristic evaluation function will terminate, constructing a complete path from s_0 to G.*

Proof: At each step, any search operator that is appended to the growing solution will cause a step down on L of at least δ, which inevitably leads to G. QED.

A more interesting and difficult case is when some of the operators may not be descending, so that the solution path may travel up and down on the CLF.

Theorem 3 *Given an SSP whose costs are bounded above zero and a CLF, L. Suppose that L is used as a heuristic evaluation of non-goal leaves, and that $L(s) - L(Succ_1(s)) \geq c_1(s)$ for all $s \notin G$. Then RFDS with any depth limit $d \geq 1$ will terminate, constructing a complete path from s_0 to G.*

Note that the theorem assumes that O_1 has the special property $L(s) - L(\text{Succ}_1(s)) \geq c_1(s)$. More generally, in each state there must be some search operator that satisfies this property. It is only for notational convenience that we have assumed that O_1 satisfies the property everywhere.

To prove Theorem 3, we require some definitions. Suppose RFDS generates a path $s_0, s_1, \ldots, s_N$, where s_N may or may not be a goal state but the other states are definitely non-goal. At the t^{th} step, $t \in \{0, \ldots, N\}$, RFDS generates a search tree to evaluate the best operator to select next. Let g_t be the cost of the search operators leading up to state s_t; let l_t be the leaf-state with the best (lowest) evaluation in the t^{th} search tree; let $c_{t,1}, \ldots, c_{t,n(t)}$ be the costs of the search operators from s_t to l_t; and let h_t be the heuristic evaluation of l_t—zero if the leaf corresponds to a goal state, and $L(l_t)$ otherwise. Define $f_t = g_t + \sum_{i=1}^{n(t)} c_{t,i} + h_t$.

Lemma 1 *Under the conditions of Theorem 3, $f_t \geq f_{t+1}$ for $t \in \{0, \ldots, N-1\}$.*

Proof: $f_t = g_t + \sum_{i=1}^{n(t)} c_{t,i} + h_t = g_{t+1} + \sum_{i=2}^{n(t)} c_{t,i} + h_t$ because s_{t+1} is one step along the path to the best leaf of iteration t. Thus, the inequality we need to establish is $\sum_{i=2}^{n(t)} c_{t,i} + h_t \geq \sum_{i=1}^{n(t+1)} c_{t+1,i} + h_{t+1}$. The right side of this inequality is simply the cost of some path from s_{t+1} to l_{t+1} plus a heuristic evaluation. The left side corresponds to a path from s_{t+1} to l_t, plus a heuristic evaluation.

First, suppose l_t is a goal state. The path from s_{t+1} to l_t will be included among those over which RFDS minimizes at the t^{th} step. Thus, in this case the inequality holds.

If l_t is not a goal state, then the search at iteration $t+1$ expands l_t. The evaluation of the path from s_{t+1} to $\text{Succ}_1(l_t)$ is

$$\begin{array}{ll} & \sum_{i=2}^{n(t)} c_{t,i} + c_1(l_t) + h(\text{Succ}_1(l_t)) \\ \leq & \sum_{i=2}^{n(t)} c_{t,i} + c_1(l_t) + L(\text{Succ}_1(l_t)) \\ \leq & \sum_{i=2}^{n(t)} c_{t,i} + L(l_t) \\ = & \sum_{i=2}^{n(t)} c_{t,i} + h_t. \end{array}$$

Thus, this path's cost meets the inequality, and since RFDS minimizes over that and other paths, the inequality must hold for the best path of iteration $t+1$. QED.

Proof of Theorem 3: Suppose RFDS runs forever without constructing a path to a goal state. From the previous lemma, $f_t \leq f_0$ for all $t \geq 0$. Since the SSP's costs are bounded above zero, $f_t \geq g_t \geq t \cdot c_{low}$. For sufficiently large t, this contradicts $f_t \leq f_0$. Thus RFDS does construct a path to goal. QED.

Theorem 3 requires a relationship between the decrease in a CLF caused by application of O_1 and the cost incurred by O_1. A given CLF may not satisfy this property, or one may not know whether the CLF satisfies the property or not. We propose two methods for generating a CLF guaranteed to have the property, assuming that applying O_1 to any non-goal state causes a decrease in the original CLF of at least δ, for some fixed $\delta > 0$.

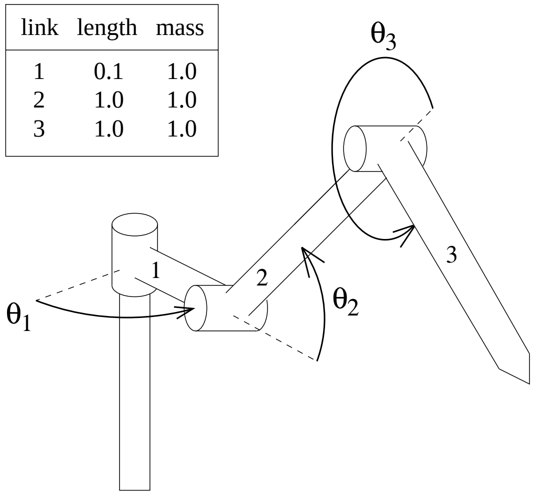
The first method is scaling, which is applicable when there is a maximum cost c_{high} that operator O_1 may incur when applied to any non-goal state. In this case, consider the new CLF $L' = \frac{c_{high}}{\delta} L$. For any $s \notin G$, $L'(s) - L'(\text{Succ}_1(s)) = (c_{high}/\delta) \cdot (L(s) - L(\text{Succ}_1(s))) \geq (c_{high}/\delta) \cdot \delta = c_{high} \geq c_1(s)$. Thus, L' is a CLF satisfying the conditions of Theorem 3.

A problem with this scheme is that one or both of the constants c_{high} and δ_1 may be unknown. An overestimate of the constant $\frac{c_{high}}{\delta_1}$ will produce a new CLF satisfying the theorem. If an overestimate cannot be made directly, then a scaling factor can be determined on-line, as RFDS runs. Suppose one defines $L' = \alpha L$ for any initial guess α. One can than perform RFDS, checking the condition $L'(s) - L'(\text{Succ}_1(s)) \geq c_1(s)$ every time O_1 is applied to some state. If the condition is violated, update the scaling factor, e.g., as $\alpha \leftarrow \frac{c_1(s)}{L'(s) - L'(\text{Succ}_1(s))} + \varepsilon$, for some fixed $\varepsilon > 0$. This update rule ensures that the guess α is consistent with all the data observed so far and will meet or exceed $\frac{c_{high}}{\delta_1}$ in some finite number of updates. If RFDS does not construct a path to a goal state by the time that a sufficient number of updates are performed, then Theorem 3 guarantees the completion of the path afterward.

A second method for generating a CLF that meets the conditions of Theorem 3 is to perform roll-outs [Tesauro and Galperin, 1996; Bertsekas *et al.*, 1997]. In the present context, this means defining $L'(s) =$ "the cost of the solution path generated by repeated application of O_1 starting from s and until it reaches G". The function L' is evaluated by actually constructing the path. The reader may verify that L' meets the definition of a CLF and satisfies the requirements of Theorem 3. Performing roll-outs can be an expensive method for evaluating leaves because an entire path to a goal state needs to be constructed for each evaluation. However, roll-outs have been found to be quite effective in both game playing and sequential control [Tesauro and Galperin, 1996; Bertsekas *et al.*, 1997].

6 Robot Arm Example

We briefly illustrate the theory presented above by applying it to a problem requiring the control of the simulated 3-link robot arm, depicted in Figure 1. The state space of the arm is $\Re^6$, corresponding to three angular joint positions and three angular joint velocities. We denote the joint position and joint velocity column vectors by θ and $\dot{\theta}$. The set G of goal states is $\{s \in S : ||s|| \leq 0.01\}$, which is a small hyper-rectangle of states in which the arm is nearly stationary in the straight-out horizontal configuration.



link	length	mass
1	0.1	1.0
2	1.0	1.0
3	1.0	1.0

Figure 1: Diagram of the 3-Link Robot Arm

The dynamics of mechanical systems such as a robot arm are most naturally described in continuous time. A standard model for a robot arm is [Craig, 1989]:

$$\frac{\mathrm{d}}{\mathrm{dt}} s = \frac{\mathrm{d}}{\mathrm{dt}} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ H^{-1}(\theta, \dot{\theta})(\tau - V(\theta, \dot{\theta}) - g(\theta)) \end{bmatrix}$$

where g represents gravitational forces, V represents Coriolis and other velocity-dependent forces, H is the inertia matrix, and τ is the actuator torques applied at the joints.

We develop a set of continuous time controllers—rules for choosing τ—-based on feedback linearization and linear-quadratic regulator (LQR) methods [e.g., Vincent and Grantham, 1997]. These controllers form the basis of the search operators described in the next section. Feedback linearization amounts to reparameterizing the control torque in terms of a vector u, where $\tau = Hu + V + g$. This puts the robot dynamics into the particularly simple linear form

$$\frac{\mathrm{d}}{\mathrm{dt}} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ u \end{bmatrix}$$

to which LQR control design is applicable. An LQR controller is a rule for choosing u, and thus τ, based on θ and $\dot{\theta}$. LQR design yields two main benefits: 1) a simple controller that can asymptotically bring the arm to any specified target state, 2) a Lyapunov function that can be used as a CLF for the problem of getting to a goal state. We define five basic controllers:

- C_1: An LQR controller with $\theta = (0,0,0)$ as the target configuration. This configuration is within G, so C_1 alone is able to bring the arm into G from any initial position. We use the associated Lyapunov function as a CLF for the SSP we are constructing: $L_{\text{arm}} = s^T Q s$, where Q is a symmetric positive definite matrix produced during the standard computations for the LQR design.
- C_2: Chooses τ as an LQR controller with target configuration $\theta = (0,0,0)$ except that u is multiplied by 2. This tends to cause higher control torques that bring the arm towards the target configuration faster.

- C_3: Similar to C_2, but u is multiplied by 0.5 instead of 2.
- C_4: An LQR controller that does not apply any torque to the first joint, but regulates the second two joints to configuration $(\theta_2, \theta_3) = (\pi/4, -\pi/2)$. This tends to fold the arm in towards the center, so it can swing around the first joint more easily.
- C_5: Similar to C_4, but regulating the second two joints to the configuration $(\theta_2, \theta_3) = (\pi/2, -\pi)$.

6.1 Search Operators

We construct two sets of five search operators. In the first set of search operators, Ops_1, each operator corresponds to one of the five controllers above. $\text{Succ}_j(s)$ is defined as the state that would result if the arm started in state s and C_j controlled the arm for $\Delta = 0.25$ time interval.

We define the cost of applying a search operator to be the continuous time integral of the quantity $||\theta(t)||^2 + ||\tau(t) - \tau_0||^2$, where $\theta(t)$ are the angles the arm goes through during the Δ-time interval, $\tau(t)$ are the joint torques applied, and τ_0 is the torque needed to hold the arm steady, against gravity, at configuration $\theta = (0,0,0)$. This kind of performance metric is standard in control engineering and robotics. The term $||\theta(t)||^2$ reflects the desire to move the arm to G; the term $||\tau(t) - \tau_0||^2$ penalizes the use of control torques to the extent that they differ from τ_0. Defined this way, the costs of all search operators are bounded above zero.

The second set of search operators, Ops_2, is also based on controllers C_1 through C_5, but the continuous-time execution is somewhat different. When a controller C_j, $j \in \{2, \ldots, 5\}$ is being applied for a Δ-time interval, the time derivative of L_{arm}, $\dot{L}_{\text{arm}}$, is constantly computed. If $\dot{L}_{\text{arm}}$ is greater than -0.1, the torque choice of C_1 is supplied instead. Under C_1, $\dot{L}_{\text{arm}} \leq \delta_1 < 0$ for some unknown δ_1. In this way, $\dot{L}_{\text{arm}}$ is bounded below zero for all the Ops_2 search operators, and thus they all step down L_{arm} by at least the (unknown) amount $\delta = \Delta \cdot \min(0.1, \delta_1)$ with each application. The costs for applying these search operators is defined in the same way as for Ops_1.

6.2 Experiments and Results

We tested A* and RFDS on the arm with both sets of search operators, averaging results over a set of nine initial configurations $\theta_0^T = (x, y, -y)$ for $x \in \{-\pi, -2\pi/3, -\pi/3\}$ and $y \in \{-\pi/2, 0, +\pi/2\}$. Theorem 1 guaranteed that A* would find an optimal solution within either set of search operators. We ran RFDS at a variety of depths and with three different heuristic leaf evaluation functions: zero (RFDS-Z), roll-outs (RFDS-R), and a scaled version of L_{arm} (RFDS-S). Because we did not know an appropriate scaling factor for L_{arm}, for each starting configuration we initialized the scaling factor to zero and updated it as discussed in Section 5, using $\varepsilon = 0.01$. All of the RFDS runs under Ops_2 were guaranteed to generate solutions by Theorem 2. For the RFDS-R and RFDS-S runs with Ops_1, Theorem 3 provided the guarantee of producing solutions.

The results are shown in Figures 2 and 3, which report average solution cost, nodes expanded by the search, and the amount of virtual time for which the robot arm dynamics were simulated during the search. Simulated time is closely related to the number of nodes expanded, except that the RFDS-R figures also account for the time spent simulating the roll-out path to the goal. The actual CPU times used were more than an order of magnitude smaller than the simulated times, and our code has not been optimized for speed. The "C_1 only" row gives the average solution cost produced by controlling the arm using C_1 from every starting point. Achieving this level of performance requires no search at all. A* with either set of search operators found significantly lower cost solutions than those produced by C_1. The A* solution qualities with Ops_1 and Ops_2 are very close. We anticipated that Ops_2 results might not be as good, because the paths are constrained to always descend on L_{arm}, whereas Ops_1 allows the freedom of ascending. For this problem, the effect is small. A* with Ops_2 involved significantly less search effort than with Ops_1, in terms of both the number of expanded nodes and the amount of arm simulation time required.

The RFDS-Z – Ops_1 combination is the only one not guaranteed to produce a solution, and indeed, it did not. Under Ops_2, RFDS-Z is guaranteed to produce a path to a goal for any search depth. At low search depths, the solutions it produced were worse than those produced by C_1. However, at greater depth, it found significantly better solutions, and with much less search effort than required by A*.

RFDS-R with either set of search operators produced excellent solutions at depth one. This result is quite surprising. Apparently the roll-out yields excellent information for distinguishing good search operators from bad ones. Another unexpected result is that greater search depths did not improve over the depth-one performance. Solution quality is slightly worse and search effort increased dramatically. At the higher search depths, RFDS-R required more arm simulation time than did A*. This primarily resulted from the heuristic leaf evaluations, each of which required the simulation of an entire trajectory to G.

RFDS-S produced good solutions with very little search effort, but greater search effort did not yield as good solutions as other algorithms were able to produce. RFDS-S seems to be the most appropriate algorithm for real-time search if very fast responses are needed. It produced solutions that significantly improved over those produced by C_1, using search effort that was less than one tenth of that of the shallowest roll-out runs, and less than one hundredth of the effort required by A*.

6.3 Discussion

The arm simulation results reflect some of the basic theory and expectations developed earlier in the paper. The algorithms that were guaranteed to find solutions did, and those that were not guaranteed to find solutions did not. These results depend on, among other things, our choice of the "duration" of a search operator, Δ. We ran the same set of experiments for four other choices of Δ: 1.0, 0.75, 0.5, and 0.1. Many of the qualitative results were similar, including the excellent performance of roll-outs with a depth-one search.

One difference was that for higher Δ, A* became more competitive with RFDS in terms of search effort. Con-

Algorithm	Sol'n Cost	Nodes Expanded	Sim. Time
C_1 only	435.0	0	0
A*	304.0	11757.4	14690.7
RFDS-Z depths 1-5	∞	∞	∞
RFDS-R depth 1	305.7	41.3	667.6
RFDS-R depth 2	307.1	158.6	2001.1
RFDS-R depth 3	307.1	477.9	5489.8
RFDS-R depth 4	307.1	1299.0	13214.6
RFDS-R depth 5	307.1	3199.6	27785.5
RFDS-S depth 1	354.3	32.9	49.0
RFDS-S depth 2	371.0	163.2	210.7
RFDS-S depth 3	359.4	675.3	848.8
RFDS-S depth 4	343.4	2669.4	3337.6
RFDS-S depth 5	334.4	9565.3	11952.1

Figure 2: Ops_1 Results

Algorithm	Sol'n Cost	Nodes Expanded	Sim. Time
C_1 only	435.0	0	0
A*	305.0	4223.3	3492.8
RFDS-Z depth 1	502.8	45.0	40.6
RFDS-Z depth 2	447.8	113.2	94.7
RFDS-Z depth 3	385.2	226.8	192.9
RFDS-Z depth 4	360.9	434.9	374.0
RFDS-Z depth 5	326.7	743.6	648.0
RFDS-R depth 1	306.4	38.2	479.8
RFDS-R depth 2	307.8	118.7	1424.1
RFDS-R depth 3	307.8	333.1	4047.0
RFDS-R depth 4	307.8	920.7	10085.3
RFDS-R depth 5	307.8	2359.6	21821.6
RFDS-S depth 1	355.3	33.0	28.2
RFDS-S depth 2	370.7	105.1	87.9
RFDS-S depth 3	359.2	324.3	284.3
RFDS-S depth 4	342.9	1019.6	920.9
RFDS-S depth 5	330.3	3011.4	2758.3

Figure 3: Ops_2 Results

versely, at $\Delta = 0.1$, A* exhausted the computer's memory and crashed. The complexity of A* generally grows exponentially with the length of the optimal solution. For RFDS, which iteratively constructs a solution, effort is linear in the length of the solution it produces. The complexity of RFDS is more governed by the search depth, which can be chosen independently. Another difference observed at $\Delta = 1.0$ is that RFDS-Z succeeded in producing solutions from all starting positions when the search depth was four or higher.

For all algorithms, solution quality was lower for higher Δ. This is simply because lower Δ allows finer-grained control over the trajectory that the arm takes. The freedom to choose Δ suggests an alternative means for performing real-time search: one could perform a sequence of A* searches, initially with a high Δ and then decrease Δ to find solutions at finer temporal resolutions.

7 Conclusion

We demonstrated how domain knowledge in the form of a Lyapunov function can help one analyze and formulate infinite-state search problems. Lyapunov methods guarantee the existence of solutions to a problem, and they can be used to show that both optimal and suboptimal/real-time search procedures will find those solutions. These results provide a theoretical basis for extending the range of problems to which heuristic search methods can be applied with a guarantee of results.

Acknowledgments

This work was funded by the National Science Foundation under Grant No. ECS-0070102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Theodore Perkins is also supported by a graduate fellowship from the University of Massachusetts Amherst.

References

[Bertsekas *et al.*, 1997] D. P. Bertsekas, J. N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 1997.

[Boone, 1997] G. Boone. Minimum-time control of the acrobot. In *1997 International Conference on Robotics and Automation*, pages 3281–3287, 1997.

[Connolly and Grupen, 1993] C. I. Connolly and R. A. Grupen. The applications of harmonic functions to robotics. *Journal of Robotics Systems*, 10(7):931–946, 1993.

[Craig, 1989] J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, 1989.

[Davies *et al.*, 1998] S. Davies, A. Ng, and A. Moore. Applying online search techniques to continuous-state reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 753–760, 1998.

[Korf, 1990] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, (42):189–211, 1990.

[Krstić *et al.*, 1995] M. Krstić, I. Kanellakopoulos, and P. Kokotović. *Nonlinear and Adaptive Control Design*. John Wiley & Sons, Inc., New York, 1995.

[Levine, 1996] W. S. Levine, editor. *The Control Handbook*. CRC Press, Inc., 1996.

[Pearl, 1984] J. Pearl. *Heuristics*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1984.

[Russell and Norvig, 1995] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1995.

[Tesauro and Galperin, 1996] G. Tesauro and G. R. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing: Proceedings of the Ninth Conference*. MIT Press, 1996.

[Vincent and Grantham, 1997] T. L. Vincent and W. J. Grantham. *Nonlinear and Optimal Control Systems*. John Wiley & Sons, Inc., New York, 1997.

Lyapunov-Constrained Action Sets for Reinforcement Learning

Theodore J. Perkins PERKINS@CS.UMASS.EDU
Andrew G. Barto BARTO@CS.UMASS.EDU
Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, 01003

Abstract

Lyapunov analysis is a standard approach to studying the stability of dynamical systems and to designing controllers. We propose to design the actions of a reinforcement learning (RL) agent to be descending on a Lyapunov function. For minimum cost-to-target problems, this has the theoretical benefit of guaranteeing that the agent will reach a goal state on every trial, regardless of the RL algorithm it uses. In practice, Lyapunov-descent constraints can significantly shorten learning trials, improve initial and worst-case performance, and accelerate learning. Although this method of constraining actions may limit the extent to which an RL agent can minimize cost, it allows one to construct robust RL systems for problems in which Lyapunov domain knowledge is available. This includes many important individual problems as well as general classes of problems, such as the control of feedback linearizable systems (e.g., industrial robots) and continuous-state path-planning problems. We demonstrate the general approach on two simulated control problems: pendulum swing-up and robot arm control.

1. Introduction

In many interesting minimum cost-to-target problems, the target, or goal, region is not a natural attractor in the dynamics of the system. Consider, for example, the acrobot—a two link pendulum which is to be swung to a near upright position (e.g. Sutton, 1996; Boone, 1997a; DeJong, 2000). To swing up the acrobot, a controller must counteract the effects of gravity and friction (if any), which favor a stationary, downward-hanging position. The fact that the goal is not an attractor in the dynamics is part of what makes this problem challenging for RL, despite the moderate dimensionality of its state space (4 dimensions) and action space (1 dimension, usually discretized into 2 or 3 actions). Conversely, when the goal is a natural attractor, the problem of learning to control the system may be significantly easier. The success of TD-gammon (Tesauro, 1994) may partly be due to the fact that games naturally tend to end; under almost any policy, pieces progress around the backgammon board and get home. Attracting goal states tend to shorten learning trials. This eases the temporal credit assignment problem and allows more trials, and thus more meaningful learning, per unit time.

In this paper, we study the possibility of making the goal of a control problem an attracting region by judicious selection of the actions made available to an RL agent. In particular, we propose to use Lyapunov methods, which have been developed primarily for the analysis of systems of (controlled) differential equations and for control design (e.g., Vincent & Grantham, 1997). A Lyapunov function constitutes a form of domain knowledge about a control problem. Such a function can be used to design or constrain the action choices of an RL agent so that they naturally lead the agent to the goal.

Expressing domain knowledge by manipulating the actions available to an RL agent has been explored before by a number of researchers (e.g. Maclin & Shavlik, 1996; Clouse, 1997; Parr, 1998; Precup, 2000; Dietterich, 2000). By focusing an agent's attention on actions already known or expected to be good, or by decomposing a problem into smaller subproblems, faster and better learning can be achieved. Our work can be viewed as seeking similar benefits, but we concentrate on a particular kind of domain knowledge, namely, Lyapunov functions. In addition to providing a beneficial heuristic bias, we seek to provide theoretical guarantees on the behavior of the RL agent. Lyapunov methods can be used to prove that an agent will reach a goal state on every trial and to give time bounds on how long trials can take. Such guarantees are particularly important for systems in which training time is at a premium or in which reasonable performance must be maintained during learning. Because Lyapunov-based performance guarantees follow from properties of the environment and of the actions available to an agent, they hold for any RL algorithm.

Although Lyapunov-based methods can ensure that an RL agent always reaches a goal state and protect against catastrophic behavior, they do not solve the optimal control problem. Typically, Lyapunov methods do not directly address the cost of controlling the system at all. Thus, it is sensible to combine Lyapunov methods with control methods, such as RL, that do attempt to minimize the cost incurred by the controller.

The combination of Lyapunov methods and RL has been studied before by Singh et al. (1994). Although specific to path-planning tasks and a particular kind of Lyapunov function, that work and the present paper have similar goals: robustness and more efficient learning.

2. Problem Definition

In this paper, we restrict attention to the control of deterministic dynamical systems. Lyapunov methods and RL are applicable to both deterministic and stochastic systems, but Lyapunov theory is considerably more complex in the stochastic case. The general ideas we present can be extended to the control of stochastic systems; this constitutes one direction for future work.

We use the following deterministic Markov decision process (MDP) framework to describe the environment of the RL agent. The environment has a state set S, which can be an arbitrary set. The set of goal states is denoted $G \subset S$; the set of non-goal states is denoted $\overline{G}$. In each non-goal state s, k actions are available to the agent. Each action a_j incurs some non-negative cost $c_j(s)$ and results in a successor state $s'_j(s)$.

A policy is a mapping $\pi : \overline{G} \mapsto \{1,\ldots,k\}$ which selects an action based on the current state. For any start state s_0, a policy π induces a trajectory: $s_1 = s'_{\pi(s_0)}(s_0)$, $s_2 = s'_{\pi(s_1)}(s_1)$, etc. A trajectory is either finite, ending at a goal state, or infinite. The value of a policy for a start state, s_0, is the discounted sum of action costs in the trajectory: $V^\pi(s_0) = \sum_i \gamma^i c_{\pi(s_i)}(s_i)$, where $\gamma \in [0,1]$ is the discount rate. We assume there exists a start state distribution S_0 on $\overline{G}$. The value of a policy is defined as its expected value across start states: $V^\pi = E_{s_0 \sim S_0} V^\pi(s_0)$. The goal of an RL agent is to find a policy that minimizes V^π.

Note that an RL agent is not explicitly required to reach a goal state. If this happens, it is as a side effect of minimizing cost. Proper use of Lyapunov functions can ensure that the RL agent reaches G on every trial.

3. Lyapunov Functions

To introduce the main idea of this paper, consider a very simple example. Figure 1a depicts a five by five "gridworld". Each square corresponds to a state of an RL agent's environment. The "S" square represents the start state, and the "G" square represents the goal state. The agent can move to any up-, down-, left-, or right-adjacent square. There is some unknown cost for entering each square.

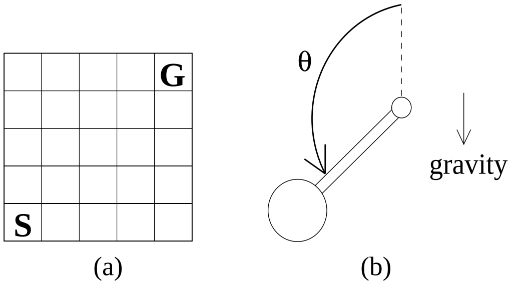


Figure 1. (a) Gridworld for Section 3. (b) Pendulum diagram for Section 4.

Suppose some unspecified RL agent is put into this environment. Knowing nothing about the agent—its learning algorithm, whether it learns on- or off-policy, its method of function approximation, its method of exploration—it is difficult to predict how the agent will behave. It is possible that it will quickly find an optimal path and follow it thereafter. On the other hand, it may not even reach the goal state once.

Now consider the case in which the RL agent is only allowed to move to up- or right-adjacent squares. Then the agent would, by necessity, reach the goal state on every trial in no more (and no less) than 10 time steps. Each action that the agent takes reduces its Manhattan distance to goal by one, leading it eventually to the goal state. The Manhattan distance of the agent to goal is a simple example of a Lyapunov function. This kind of guarantee on goal achievement, deriving from the actions available to RL agent and not the RL algorithm itself, is the main idea we pursue in this paper. But we address more interesting and difficult classes of problems than this simple gridworld.

Lyapunov functions are often thought of as generalized "energy" or "potential" functions. They take large values away from the goal, and decrease monotonically towards the goal. There are many different definitions of "Lyapunov function," depending on the context in which it is applied. We use the following:

Definition *Given a deterministic MDP, a* Lyapunov function *is a function $L : S \mapsto \Re$ such that* (i) $L(s) > 0$ *for all states $s \in \overline{G}$, and* (ii) *there exists $\delta > 0$ such that for all states $s \in \overline{G}$ there exists an action a_j such that either $s'_j(s) \in G$ or $L(s) - L(s'_j(s)) \geq \delta$.*

That is, a Lyapunov function is positive for any non-goal state, and in any non-goal state some action either leads immediately to the goal or to a state at least δ lower on the function. If the second condition holds for all actions, i.e., the "exists an action a_j" is replaced by "for all actions a_j", then we will say the MDP is descending on L. This condition can be achieved by proper selection, or restriction, of the actions available to an agent and has both theoretical and practical benefits. The main theoretical benefit is formalized by the following:

Theorem *If a deterministic MDP is descending on a Lyapunov function L, then any trajectory starting from $s_0 \in \overline{G}$ must enter G in no more than $\lceil L(s_0)/\delta \rceil$ time steps.*

Proof Suppose that for some $s_0 \in \overline{G}$ an agent could perform a sequence of $\lceil L(s_0)/\delta \rceil$ actions and never enter G. Since each action leads to a successor state where L is lower by at least δ, the non-goal state at the end of the action sequence would have L-value no more than $L(s_0) - \delta \lceil L(s_0)/\delta \rceil \leq 0$. But no non-goal states have $L \leq 0$, so we have a contradiction.

A Lyapunov function constitutes a strong form of domain knowledge, and certainly is not always available. In principle, Lyapunov functions exist for any problem in which the goal is reachable (Vincent & Grantham, 1997). Finding a Lyapunov function in analytic form for a given problem is, however, something of an art. Numerous sources discuss methods for finding Lyapunov functions (e.g., Vincent & Grantham, 1997; Krstić et al., 1995). For many important problems and classes of problems, standard Lyapunov functions have already been developed. For example, feedback linearizable systems are easily analyzed by Lyapunov methods. These systems include almost all modern industrial robots (Vincent & Grantham, 1997). At least two major types of Lyapunov functions can be applied to continuous-state path planning problems (Rimon & Koditschek, 1992; Connolly & Grupen, 1993). Other concrete and abstract control problems are discussed in, e.g., Kalman & Bertram (1960), Anderson & Grantham (1989), Krstić et al. (1995), and Vincent & Grantham (1997).

4. Pendulum Swing-up and Balance Task

Boone (1997a; 1997b) and DeJong (2000), among others, have discussed the role of mechanical energy in controlling pendulum-type systems. The standard tasks are either to swing the pendulum's end point above some height (swing-up) or to swing the pendulum up and balance it in a vertical position (swing-up and balance). In either task, the goal states have greater total mechanical energy (potential plus kinetic) than the start state, which is typically the hanging-down rest position. Thus, any controller that solves one of these tasks must somehow impart a certain amount of energy to the system. Our first demonstration problem is a swing-up and balance task for a single-link pendulum.

4.1 Pendulum Problem Definition

Figure 1b depicts the pendulum. Its state is given by its angular position, $\theta \in [-\pi, \pi]$, and its angular velocity, ω. The pendulum's continuous-time dynamics are given by

$$\dot{\theta} = \omega, \quad \dot{\omega} = \sin(\theta) + u,$$

where the sine term is due to the acceleration of gravity, and u is a control torque applied at the fulcrum of the pendulum. These equations assume a mass one, length one pendulum. The magnitude of the control torque is bounded by some $\mu > 0$. We assume μ is relatively small, so that the pendulum cannot be directly driven to the balanced position. Instead, the solution involves swinging the pendulum back and forth repeatedly, pumping energy into it until it can swing upright.

The mechanical energy of the pendulum is $\mathrm{ME}(\theta, \omega) = 1 + \cos(\theta) + \frac{1}{2}\omega^2$. When the pendulum is upright and stationary, $\mathrm{ME} = 2.0$. For any other state with $\mathrm{ME} = 2.0$, if u is taken to be zero, the pendulum will naturally swing upright and asymptotically approach the vertical, stationary state $(\theta = 0, \omega = 0)$. The pendulum swing-up and balance problem for the single-link pendulum can therefore be reduced to the problem of achieving any state with $\mathrm{ME} = 2.0$. We take $G = \{(\theta, \omega) \mid \mathrm{ME}(\theta, \omega) = 2.0\}$. The cost incurred by any action is 1.0, making this a minimum-time control problem.

We define four actions for controlling the pendulum. Each action corresponds to applying a continuous-time control law to the pendulum for a time interval of unit duration, or less if the pendulum reaches G before the interval ends. In the experiments reported below, optimal policies are approximately 15 steps long and random policies take, on average, between 25 and 100 steps to reach G, depending on which action set is used.

The first two actions we define correspond to constant-torque controls laws: a_1 corresponds to the torque $-\mu$ and a_2 corresponds to the torque $+\mu$. The second two actions are based on an "energy-ascent" control law, EA. To a first approximation, EA applies maximum allowed torque in the direction of current pendulum motion. This has the effect of maximizing the time derivative of mechanical energy, which is just $\frac{\mathrm{d}}{\mathrm{dt}}\mathrm{ME} = \omega \cdot u$. However, the rule is made somewhat more complex by the need to avoid potential equilibrium points where gravity and $\pm\mu$ torque cancel out:

$$\mathrm{EA}(u, \varepsilon_\theta, \varepsilon_\omega) = \begin{cases} \mathrm{sign}(\theta)u & \text{if } \omega = 0 \\ \frac{1}{2}\mathrm{sign}(\omega)u & \text{if } (0 < \omega \leq \varepsilon_\omega \text{ and} \\ & \quad (\theta \in [\theta_3 - \varepsilon_\theta, \theta_3) \text{ or} \\ & \quad \theta \in [\theta_4 - \varepsilon_\theta, \theta_4)\ )) \\ & \text{or } (-\varepsilon_\omega < \omega < 0 \text{ and} \\ & \quad (\theta \in (\theta_1, \theta_1 + \varepsilon_\theta] \text{ or} \\ & \quad \theta \in (\theta_2, \theta_2 + \varepsilon_\theta]\ )) \\ \mathrm{sign}(\omega)u & \text{else} \end{cases}$$

where $\mathrm{sign}(x) = 1$ if $x \geq 0$ and -1 otherwise, $u > 0$, ε_θ and

ε_ω are small positive constants, and $\theta_1, \ldots, \theta_4$ are equilibrium points of $\pm\mu$ with gravity: $\theta_1 = \arcsin(\mu)$, $\theta_2 = \pi - \theta_1$, $\theta_3 = \theta_1 - \pi$, $\theta_4 = -\theta_1$. The complex condition in the middle case checks for the pendulum having low velocity and being near one of the equilibrium points. When that condition holds, the torque $\frac{1}{2}\text{sign}(\omega)u$ is used instead of $\text{sign}(\omega)u$. This still increases ME, but there is no danger of being stuck approaching an equilibrium point with this smaller torque. Action a_3 corresponds to $\text{EA}(\mu, 0.1, 0.1)$ and a_4 corresponds to $\text{EA}(\frac{1}{2}\mu, 0.1, 0.1)$. Both a_3 and a_4 can be shown to increase the mechanical energy of the pendulum by at least some amount $\delta > 0$ regardless of the pendulum's state. (We omit proof due to space considerations.)

These four actions are organized into three different action sets: $\text{A}_{\text{const}} = \{a_1, a_2\}$, $\text{A}_{\text{EA}} = \{a_3, a_4\}$, and $\text{A}_{\text{all}} = \{a_1, a_2, a_3, a_4\}$. Because a_3 and a_4 both cause a δ step up on ME, the function $L(\theta, \omega) = 2 - \text{ME}(\theta, \omega)$ defines a Lyapunov function for the MDP induced by those two actions. Further, that MDP is descending on L, so the theorem of Section 3 applies, ensuring that an agent restricted to use these actions will always reach G. There is no such guarantee for agents using either of the other two action sets, although one might expect A_{all} to bias an RL agent towards the goal because it includes the two EA-based actions.

4.2 Q-Learning Experiments on the Pendulum

To represent action-values, we discretized the state space region $[-\pi, \pi]^2$ into uniform bins by splitting both dimensions into B intervals. We report results for $B = 2^0, 2^1, 2^2, \ldots, 2^8$. The total number of bins thus ranged from 1 to $2^{16} = 65,536$. For each bin and each action, we maintained an action-value. For each choice of B and each action set, we ran 30 independent runs of Q-learning (Watkins, 1989). For the kth update of an action value, we used the learning rate parameter $1/\sqrt{k}$. We found this outperformed various constant learning rates as well as the choice $1/k$. Each run contained 10,000 learning trials. The start state for every trial was $(\theta, \omega) = (-\pi, 0)$. Action selection was $\varepsilon-$greedy with $\varepsilon = 0.1$, and ties were broken randomly (Sutton & Barto, 1998). Trials were terminated if they took longer than 900 time steps. After each learning trial we performed a test trial, controlling the pendulum according to the current greedy policy with no exploration and no learning.

The results we report correspond to a choice of $\mu = 0.224$. We obtained qualitatively similar results for other choices of μ, but this choice magnified the differences between the results for the three action sets, making comparisons easier. The discount rate, γ, was 1.

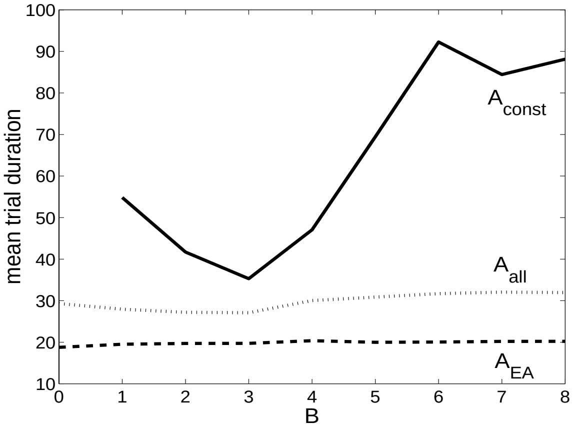


Figure 2. Early learning performance on the pendulum

4.3 Pendulum Results

One of the primary effects of having the energy-based actions a_3 and a_4 was to improve the initial performance of the learner. Figure 2 displays the mean number of steps to goal (equivalently, total cost incurred) during the first 10 learning trials, averaged over the 30 independent runs. Learning trials were significantly shorter for agents with action sets A_{EA} or A_{all} than for agents that with A_{const}. The energy-based actions strongly biased these agents to reach the goal. Agents that used A_{EA}, which guaranteed that the mechanical energy increased on every step, further outperformed the A_{all} agents. These effects were strongest at the higher values of B. When the state space was divided into a large number of bins, all the agents behaved nearly randomly because of the random tie-breaking rule and the improbability of revisiting bins. Trial times under A_{EA} were practically unaffected by the state space binning resolution. Note that there is no data point for A_{const} at $B = 0$, because almost all of those learning trials timed out. Using only the policies "$-\mu$ always" and "$+\mu$ always", it is virtually impossible swing up the pendulum, except by a lucky sequence of random exploratory actions.

Figure 3 displays the final performance under the different action sets, as measured by the mean duration of the last 100 test trials. One clear trend is that learning was most successful at the intermediate resolutions, around $B = 5$ or 6. There, and at $B = 4$ and $B = 7$ as well, A_{const} agents performed the best. The final performances for the three action sets were: 14.51 ± 0.00 for A_{const} at $B = 5$, 14.65 ± 0.05 for A_{all} at $B = 5$, and 15.44 ± 0.01 for A_{EA} at $B = 6$. It appears that the restriction of A_{EA} to descend on the Lyapunov function does limit performance. No A_{EA} run was shorter than 15 time steps. That A_{all} did not do as well as A_{const} was surprising, since the former can represent any policy that the latter can. The difference is slight, however, for intermediate values of B. At

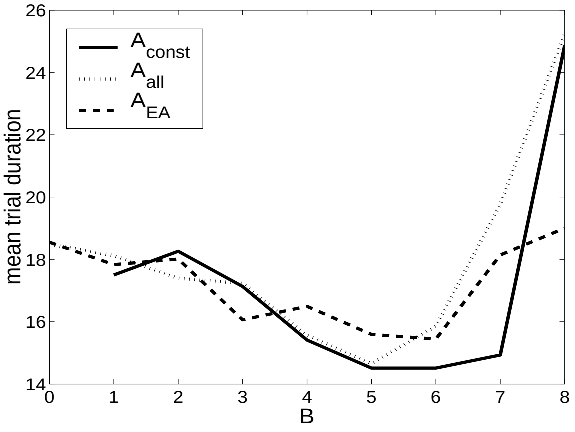


Figure 3. Final test performance on the pendulum

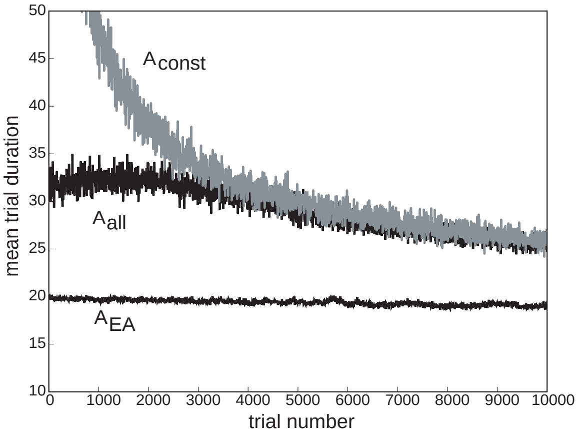


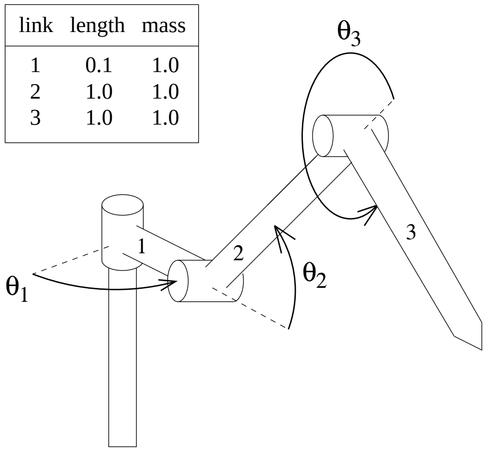
Figure 4. Learning curves for $B = 8$ on the pendulum

higher values of B, learning had not asymptoted by the end of the 10,000 trials, as is seen clearly in Figure 4. There are $2^{2B} \times$ (Number of actions) free parameters in the action value functions, so it is unsurprising that learning was slow for high B.

Lyapunov-based constraints made a big difference in the worst-case behavior of the learning agents. Over all runs, the longest learning trial under A_{EA} only took 33 time steps, compared to 136 time steps for A_{all} and 547 time steps for A_{const} $(B \geq 1)$. Under A_{all} and A_{const}, the longest test trials timed out after 900 time steps. Thus, Lyapunov-based constraints can play an important role in keeping trials short and in maintaining performance during learning.

5. Robot Arm Control

The second problem we consider is controlling a simulated 3-link robot arm, depicted in Figure 5. The arm is a special case of a feedback linearizable system, an important class of systems to which Lyapunov methods are applicable.



link	length	mass
1	0.1	1.0
2	1.0	1.0
3	1.0	1.0

Figure 5. Diagram of the 3-link arm

5.1 Arm Problem Definition

The state space of the arm is $\Re^6$, corresponding to three angular joint positions and three angular velocities. We denote the joint position and joint velocity column vectors by Θ and $\dot{\Theta}$. We define $G = \{s \in S : ||s|| \leq 0.01\}$. This is a set of low-velocity states near to a fully-extended horizontal configuration. The start state distribution is uniform over the nine configurations $\Theta_0 = (x, y, -y)^T$ for $x \in \{-\pi, -2\pi/3, -\pi/3\}$ and $y \in \{-\pi/2, 0, +\pi/2\}$, with zero velocity.

The dynamics of the arm are modeled by:

$$\frac{\mathrm{d}}{\mathrm{dt}} \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} = \begin{bmatrix} \dot{\Theta} \\ H^{-1}(\Theta, \dot{\Theta})(\tau - V(\Theta, \dot{\Theta}) - \boldsymbol{G}(\Theta)) \end{bmatrix},$$

where H is the inertia matrix, τ is the vector of actuator torques applied at the joints, V is a vector modeling Coriolis and other velocity-dependent forces, and $\boldsymbol{G}$ is a vector modeling gravitational forces (Craig, 1989).

As with the pendulum, each of the RL agents' actions corresponds to running a continuous-time controller for a time interval of unit duration or until the arm reaches G. This makes optimal trajectories around 10 steps long under the actions we define below. The continuous-time controllers for the arm are based on feedback linearization and linear-quadratic regulator (LQR) methods (e.g., Vincent & Grantham, 1997).

A simple approach to feedback linearization in the robot manipulator case is to reparameterize the control torque in terms of a vector u so that $\tau = Hu + V + \boldsymbol{G}$. This puts the robot dynamics into the linear form $\frac{\mathrm{d}}{\mathrm{dt}}[{}^{\Theta}_{\dot{\Theta}}] = [{}^{\dot{\Theta}}_{u}]$. Once the dynamics are expressed in linear form, LQR design methods are applicable. LQR design produces continuous-time controllers for the arm, and also provides the Lyapunov function which constrains one of the action sets we define below.

LQR methods result in a linear control law of the form $u = -M\left[\begin{smallmatrix}\Theta-\Theta_0\\ \dot{\Theta}\end{smallmatrix}\right]$, where Θ_0 is a desired target configuration. If the control law is applied indefinitely, it causes the state to asymptotically approach the target position. The matrix M results from the numerical solution of a matrix equation. We compute LQR controllers to minimize the cost functional $\int_{t=0}^{\infty} ||\Theta(t)||^2 + ||u||^2 dt$, where $\Theta(t)$ is the position component of the trajectory generated by the LQR controller from any starting configuration $\Theta(0)$. The resulting matrix M does not depend on the starting configuration. We define five LQR-based controllers:

C_1 An LQR controller for the feedback linearized arm with $\Theta = (0,0,0)$ as the target configuration. This configuration is within G, so C_1 alone is able to bring the arm into G from any initial configuration.

C_2 An LQR controller with target configuration $\Theta = (0,0,0)$ except that u is multiplied by 2. This tends to cause larger control torques that bring the arm towards the target configuration faster.

C_3 An LQR controller with target configuration $\Theta = (0,0,0)$ except that u is multiplied by 0.5. This tends to cause smaller control torques that bring the arm towards the target configuration more slowly.

C_4 An LQR controller that regulates towards the target configuration $(\Theta_1, \pi/4, -\pi/2)$ where Θ_1 is the current position of joint 1. This tends to fold the arm in towards the center, so it can swing around the first joint more easily.

C_5 Similar to C_4, but regulating the second two joints to the configuration $(\Theta_2, \Theta_3) = (\pi/2, -\pi)$.

The first action set for the arm, $\mathrm{A}_{\mathrm{LQR}}$, has five actions, corresponding to the five controllers above.

A benefit of LQR design is that it produces a Lyapunov function as a side effect. Associated with each LQR controller is a quadratic Lyapunov function for the problem of getting to a small region around the controller's target configuration. In general, this takes the form $[(\Theta - \Theta_0)\ \dot{\Theta}]W[(\Theta - \Theta_0)\ \dot{\Theta}]^T$, where W is a symmetric positive definite matrix. Since C_1's target point is within G, we can use its Lyapunov function, $L = [\Theta\ \dot{\Theta}]W_1[\Theta\ \dot{\Theta}]^T$, to design a second action set.

Control law C_1 causes the arm to descend continuously on L. It is easy to show that it produces $\frac{\mathrm{d}}{\mathrm{dt}}L \leq -\delta < 0$ for some δ in any non-goal state (Vincent & Grantham, 1997). However, the other control laws do not necessarily descend on L. To produce a set of controllers that descend on L, we use the following transformation. For any controller $C_j,\ j \in \{2,3,4,5\}$ we define the L-descending version of the controller to be the one that chooses u as:

$$u = \begin{cases} C_j(s) & \text{if } L \geq 0.1 \text{ and that choice} \\ & \text{produces } \frac{\mathrm{d}}{\mathrm{dt}}L \leq -0.1 \\ C_1(s) & \text{otherwise.} \end{cases}$$

In other words, if controller C_j is not descending fast enough (or at all) on L or if the arm is already near the goal, then the control choice of C_1 is supplied instead. The second action set for the arm, $\mathrm{A}_{\text{L-desc}}$, contains five actions, corresponding to C_1 and the L-descending versions of $C_2, \ldots, C_5$. $\mathrm{A}_{\text{L-desc}}$ induces an MDP which is descending on L, and thus satisfies the theorem in Section 3.

When an action is selected, the cost incurred is the continuous-time integral, until the next action choice, of the quantity $c = ||\Theta(t)||^2 + ||\tau(t) - \tau_0||^2$. τ_0 is the torque needed to hold the arm steady in the horizontal configuration. This is a standard cost function used in control theory. The first term penalizes the agent to the extent that the arm is away from the goal configuration. The second term penalizes the use of high torques, not counting the torque τ_0, which is unavoidable. This is not a minimum-time cost function. It also differs from the cost function used in the LQR design because it penalizes $\tau - \tau_0$ instead of u. This cost function comes closer to penalizing the whole control effort applied, rather than just penalizing the control effort applied to the feedback linearized system.

5.2 Q-Learning Experiments on the Arm

Due to the choice of start states and G, we can restrict attention to the region of state space: $\Theta \in [-4,1] \times [-2,2]^2$, $\dot{\Theta} \in [-3,3]^3$. To represent action values, the region was divided into bins, with B equal intervals along each dimension. We report results for B =1, 3, 5, 7, 9, and 11. The total number of bins thus ranged between 1 and $11^6 = 1,771,561$. For each bin we maintained action-values, which were updated according to the Q-learning update rules (Watkins, 1989). For each choice of B and both action sets, we performed 20 independent runs of 10,000 learning trials. Action selection was $\varepsilon-$greedy with $\varepsilon = 0.1$. Trials were terminated if they took longer than 500 time steps. After every 25th learning trial, a test trial was run from each of the nine starting points, to evaluate the current policy. The learning rate parameter for the kth update of an action-value was $1/\sqrt{k}$. The discount rate, γ, was 1.

5.3 Arm Results

Figure 6 displays the mean cost of the first 100 learning trials for the two action sets, averaged across the 20 inde-

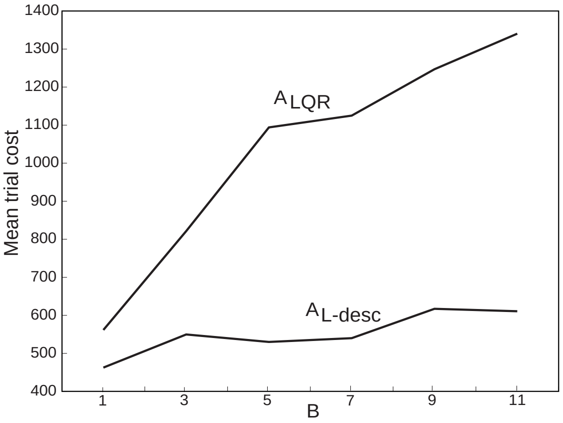


Figure 6. Early learning performance on the arm

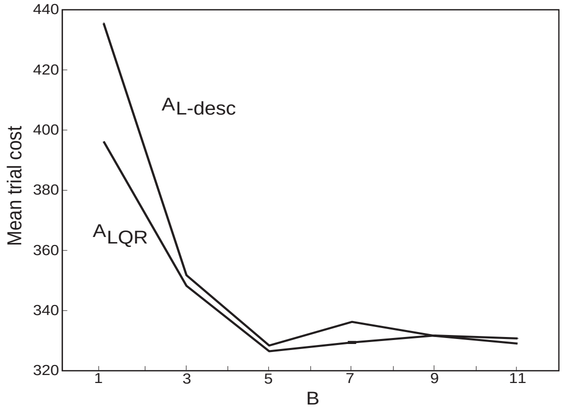


Figure 7. Final test performance on the arm

pendent runs. $A_{L\text{-desc}}$ produced significantly better initial performance than A_{LQR}. Under either action set, performance decreased significantly as the resolution of the state space binning increases. These observations are similar to the results obtained with the pendulum.

Figure 7 shows the value of the final policies discovered under each action set, as measured by the mean trial cost during the last 10 testing periods. There was little difference in asymptotic performance under the two action sets, except at $B = 1$. It appears that good policies descend on L, so that a restriction to descend on L costs little or nothing in terms of long-term performance. This is plausible, considering that the $||\Theta||^2$ component of action costs penalizes movement away from G.

Although final performance was good for both action sets, performance under A_{LQR} was sometimes quite poor during learning. The longest learning trial took 244 time steps and incurred a cost of 15,421. Some test trials timed out. By comparison, the worst learning or test trial under $A_{L\text{-desc}}$ took only 15 time steps and incurred a cost of 2,644; both

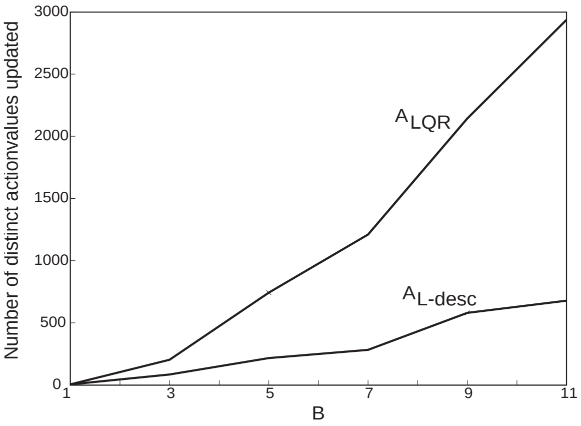


Figure 8. Mean memory usage for arm learning

statistics are an order of magnitude better than the figures for A_{LQR}. As with the pendulum, Lyapunov constraints played an important role in maintaining good performance during learning.

Another benefit of the Lyapunov-based actions was that they focused learning on a much smaller portion of state space than was explored by agents using action set A_{LQR}. Figure 8 plots the mean number of distinct action-values updated by the end of 10,000 learning trials. At most choices of B, the $A_{L\text{-desc}}$ agents updated approximately one-fifth as many distinct action-values as agents using A_{LQR}. This effect was not observed in the pendulum experiments. Reducing the portion of the state-action space explored by learning can have important implications if one uses function approximators with memory requirements that are proportional to this factor. For example, in a simple binned state space representation, one can save memory by allocating space for an action value only when it is updated for the first time. Reducing the number of states and actions explored by an agent can increase memory savings, allowing finer resolution representations or permitting easier scaling to higher dimensional problems.

6. Conclusion

Lyapunov methods allow the design of controllers that are guaranteed to bring a system to a goal state. However, Lyapunov methods do not directly address the cost of controlling the system. By formulating the actions available to an RL agent so that they descend on a Lyapunov function, one can use learning to minimize costs while retaining the strong theoretical and practical benefits of Lyapunov methods. In theory, one can guarantee that the RL agent reaches a goal state on every trial. In the two examples presented in this paper, we illustrated how Lyapunov-based actions can greatly improve a Q-learner's initial performance and help maintain good performance during learning.

Manipulating the action formulation of an RL agent is certainly not the only way to use Lyapunov domain knowledge. Ng et al. (1999) proposed a general method for changing the reward function of an MDP to encourage movement toward a goal as measured by some "potential" function. Lyapunov functions seem excellent candidates for the potential functions in such a scheme. Combining our approach with theirs might yield even better results. Alternatively, in a Bayesian framework such as the one proposed by Dearden et al. (1998), Lyapunov domain knowledge might be used to set priors about the values of actions or about which actions are more likely to be optimal ones.

Acknowledgments

This work was funded by the National Science Foundation under Grant Nos. ECS-9980062 and ECS-0070102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Theodore Perkins is also supported by a graduate fellowship from the University of Massachusetts Amherst. We thank Doina Precup and Michael Rosenstein for comments on earlier drafts of this paper.

References

Anderson, M. J., & Grantham, W. J. (1989). Lyapunov optimal feedback control of a nonlinear inverted pendulum. *Journal of Dynamic Systems, Measurement, and Control*.

Boone, G. (1997a). Efficient reinforcement learning: Model-based acrobot control. *1997 International Conference on Robotics and Automation* (pp. 229–234).

Boone, G. (1997b). Minimum-time control of the acrobot. *1997 International Conference on Robotics and Automation* (pp. 3281–3287).

Clouse, J. A. (1997). *On integrating apprentice learning and reinforcement learning*. Doctoral dissertation, University of Massachusetts Amherst.

Connolly, C. I., & Grupen, R. A. (1993). The applications of harmonic functions to robotics. *Journal of Robotics Systems*, *10*, 931–946.

Craig, J. J. (1989). *Introduction to robotics: Mechanics and control*. Addison-Wesley.

Dearden, R., Friedman, N., & Russell, S. (1998). Bayesian q-learning. *Proc. AAAI-98, Madison, Wisconsin*. AAAI Press.

DeJong, G. (2000). Hidden strengths and limitations: An empirical investigation of reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 215–222). Morgan Kaufmann.

Dietterich, T. G. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Kalman, R. E., & Bertram, J. E. (1960). Control system analysis and design via the "second method" of lyapunov ii: Discrete-time systems. *Transactions of the ASME: Journal of Basic Engineering*, 394–400.

Krstić, M., Kanellakopoulos, I., & Kokotović, P. (1995). *Nonlinear and adaptive control design*. New York: John Wiley & Sons, Inc.

Maclin, R., & Shavlik, J. W. (1996). Creating advice-taking reinforcement learners. *Machine Learning*, *22*, 251–281.

Ng, A. Y., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 278–287). Morgan Kaufmann.

Parr, R. (1998). *Hierarchical control and learning for markov decision processes*. Doctoral dissertation, University of California, Computer Science Division, Berkely.

Precup, D. (2000). *Temporal abstraction in reinforcement learning*. Doctoral dissertation, University of Massachusetts Amherst.

Rimon, E., & Koditschek, D. E. (1992). Exact robot navigation using artificial potential functions. *IEEE Transactions on Robotics and Automation*, *8*, 501–517.

Singh, S. P., Barto, A. G., Grupen, R., & Connolly, C. (1994). Robust reinforcement learning in motion planning. *Advances in Neural Information Processing Systems 6* (pp. 655–662). Cambridge, MA: MIT Press.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8* (pp. 1038–1044). Cambridge, MA: MIT Press.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, Massachusetts: MIT Press/Bradford Books.

Tesauro, G. J. (1994). Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*, 215–219.

Vincent, T. L., & Grantham, W. J. (1997). *Nonlinear and optimal control systems*. New York: John Wiley & Sons, Inc.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Doctoral dissertation, Cambridge University, Cambridge, England.

Learning and Approximate Solutions to Partially Observable Markov Games for UAVs and UGVs.

Shankar Sastry(sastry@eecs.berkeley.edu)

Joint with Jin Kim, Omid Shakernia, David Shim, Rene Vidal, Hoam Phong, Peter Ray, Ron Tal and Andrew Ng, Michael Jordan.

Abstract

Deterministic pursuit-evasion games on finite graphs have been relatively well-studied, and there has been an attempt to abstract the region in which the game takes place to a finite graph. However, when the environment is unknown a priori, the "map-learning" phase often precedes it, which is time-consuming and computationally very hard even for the simplest environment. In this research, we formulate pursuit-evasion games involving unmanned aerial vehicles (UAVs) and unmanned ground vehicles (UGVs) in a probabilistic framework and use reinforcement learning and approximate dynamic programming to obtain approximate solutions with the satisfactory performance. We are extending this result to games involving active evaders or obstacles to compute the pursuit policies that are (sub-)optimal in a sense of minimizing the expected time to find the evader or increasing the possibility of finding the computation of finding the evader in a given finite time interval.

Adaptive Dynamic Programming for Dynamic Resource Allocation Problems

Warren B. Powell
Department of Operations Research and Financial Engineering
Princeton University
powell@princeton.edu

Abstract

Dynamic resource allocation problems arise in a variety of settings in transportation and logistics, including the management of fleets of vehicles, drivers, and the freight being moved. These can be modeled as multistage optimization problems, but they are characterized by high dimensional state vectors, information vectors and decision vectors. We outline a strategy for solving these three j"curses of dimensionality" by combining a nonstandard form of the optimality recursion with separable, nonlinear approximations of the value function.

Adaptive Dynamic Programming for Multistage Stochastic Resource Allocation

Warren B. Powell
Gregory Godfrey
Katerina Papadaki
Michael Spivey
Huseyin Topaloglu

Department of Operations Research and Financial Engineering
Princeton University, Princeton, NJ 08544

February, 2000

Abstract

We introduce the concept of continuous functional approximations for the solution of multistage linear and integer programs arising in the context of stochastic resource allocation. Two classes of functional approximation are considered: linear, and piecewise linear, separable. We show that for the nonlinear case, it is important to set up the optimality recursion using the concept of an incomplete state variable, which dramatically simplifies the algorithm. We propose specific updating strategies for the functional approximations, and compare our strategies to techniques drawn from discrete dynamic programming (including neuro-dynamic programming), and multistage linear programming (especially nested Benders decomposition). Finally, we illustrate the techniques in the context of four different problem classes: a batch dispatching process, single commodity flow problems, integer multicommodity flow problems, and the dynamic assignment problem.

Multistage decision processes arise in a variety of applications where information arrives over time, forcing new decisions to be made in response to this information. For the purposes of this paper, it is useful to divide this work into two broad classes: problems with scalar (or very low dimension) decision variables, and those with vector decision variables, possibly of very high dimension. Problems with scalar decision variables include the extensive literature on option pricing, optimal control of batch service queues and optimal stopping problems. These problems generally lend themselves to analytical or numerical solutions based on dynamic programming or control theory.

Our focus in this paper is on the second class of problems, which involves vector-valued decision variables with possibly high dimensionality. These problems arise in multiproduct inventory and distribution, fleet management, routing and scheduling of vehicles and crews, personnel management, asset allocation and dynamic traffic assignment. The most common approach in practice is to solve these problems using classical math programming algorithms on deterministic instances of the problem, either using myopic on-line algorithms (Gendreau, Guertin, Potvin & Taillard (1999), Hoogeveen & Vestjens (2000), Hall, Schulz, Shmoys & Wein (1997)) or deterministic rolling-horizon techniques (Lundin & Morton (1975), Bhaskaran & Sethi (1987), Bes & Sethi (1988)). There has been growing interest in the research community in the use of stochastic programming techniques (see Sen & Higle (1999) for a nice recent review of the field), although the practical application of this work has focused primarily in the area of financial asset allocation (see, for example, Consigli & Dempster (1998) and Mulvey & Vladimirou (1992)). Financial problems often exhibit complex stochastic properties, forcing the use of scenarios sampled from past history.

Our interest is in problems which exhibit a compact state variable, which indicates that dynamic programming would be a natural framework. In the past, however, dynamic programming has focused on problems with discrete state and decision variables, and the algorithms have not scaled well to even medium-sized problems (see, for example, the superb reference text Puterman (1994)). Recent books (Bertsekas & Tsitsiklis (1996), Sutton & Barto (1998)) have brought attention to the promise of approximation methods in the context of forward dynamic programming algorithms. These techniques, referred to in the literature as "neuro-dynamic programming" or "reinforcement learning," help to offset the classic "curse of dimensionality" problem, but do not completely avoid it. For example, our interest is in problems which exhibit vector-valued decision variables with large (or infinite) feasible regions. Our approach builds on the work of Bertsekas & Tsitsiklis (1996), and offers specific methods for solving the problems posed by vector-valued decisions. Consistent with these techniques, our approximation methods are designed to provide good solutions; proofs of optimality will be limited to special cases.

This paper has grown out of a series of articles which have used a new approach for solving multistage stochastic optimization problems, each in the context of a specific application: vehicle dispatching (Papadaki & Powell (2001*a*)) and multi-product lot sizing (Papadaki & Powell (2001*b*)), dynamic networks (Godfrey & Powell (1999)), integer, multicommodity flow problems (Topaloglu & Powell (2000)) and the dynamic assignment problem (Spivey & Powell (2000)). Here, we present this approach in a general setting, and place it in the context of other methods for solving multistage stochastic optimization problems. We propose an approach that scales well to very large problems, including problems with vector-valued decisions of high dimensionality. We argue that dynamic programming suffers not from a single curse of dimensionality, but actually three curses: the state space, the outcome space, and the action space. We propose a method that avoids all three curses, in part through the use of specially chosen functional approximations of the value function, and in part through the introduction of an *incomplete* state variable, which captures the effect of prior decisions, but does not capture all the information required to make a decision. The interaction between the choice of the value function approximation and the structure of the problem is explored by comparing the structure of each of the four problems considered above.

The contribution of this paper is to merge the principles of approximation methods in dynamic programming, represented by the work of Bertsekas & Tsitsiklis (1996), with multistage linear programming. There have been two lines of research in multistage linear programming: scenario methods extended to multistage problems (see, for example, Mulvey & Vladimirou (1991*a*)), and nested Bender's decomposition (Birge (1985), Pereira & Pinto (1991), Infanger & Morton (1996), Chen & Powell (1999*a*)). For multistage problems scenario methods have the effect of taking what can already be a difficult problem, and require us to solve a problem that is magnified by the number of scenarios we have sampled to represent the future. Although specialized algorithms have been developed for this problem class (Ruszczynski (1993)) for some problem classes the resulting problem is simply intractable, especially when integrality is required. Nested Bender's decomposition is closest to our view of the world because it decomposes a problem into a sequence of single period problems using approximations of the recourse function. We are primarily interested in techniques based on Monte Carlo sampling (which are effectively multistage extensions of Higle

& Sen (1991)). For several of the problems we consider, we also use dual information from a stage $t+1$ problem to approximate a value function to be used in stage t. The concept of using dual information to approximate the impact of a decision made in the future was also used in Adelman & Nemhauser (1999) for a remnant cutting stock problem. This problem provided considerable special structure which allowed the dual of the second stage problem to be solved once, after which the first stage cutting stock problem could be solved.

Algorithms based on nested Bender's decomposition produce sequences of linear programs, but otherwise do not capture the structure of the types of problems we consider. In particular, we consider problems where the history of prior decisions is captured through a resource state vector which appears as a right-hand-side constraint. A central theme of this paper is the design of functional approximations which strike a balance between accurately capturing the future impact of decisions made now, and offering sufficient structure so that the one-period optimization problems (which may be quite large in practice) are computationally tractable.

Section 1 presents our basic problem as a sequential decision process. Here we introduce the concept of an incomplete state variable and show how the optimality equations of dynamic programming are modified. Then, section 2 describes the solution strategy. Fundamental is the choice of value function approximations that retain appropriate structural properties. Section 3 describes the use of two basic classes of functions: linear in the state variable, and nonlinear but separable in the state variable. Section 4 compares our approach to techniques drawn from dynamic programming and multistage stochastic programming. Finally, section 5 shows how these techniques can be applied to four important problem classes: aging and replenishment problems (which arise in inventory problems); dynamic network flow problems; integer, multicommodity network flow problems; and the dynamic assignment problem. This final section represents a summary of work presented in papers which address each of these problems in depth. Our presentation puts the discussion of these problems into a common framework, and illustrates the interaction between problem structure and approximation method.

1 Fundamentals

Our interest is in solving a class of problems which, in a deterministic setting, might be formulated as:

$$\max_{x \in \mathcal{X}} \sum_{t \in \mathcal{T}} c_t(x_t) \tag{1}$$

subject to:

$$\begin{aligned} A_t x_t - B_{t-1} x_{t-1} &= \hat{R}_t && (2) \\ D_t x_t &\leq u_t && (3) \\ x_t &\geq 0 && (4) \end{aligned}$$

where the matrices A_t and B_t handle system dynamics, D_t captures possible coupling constraints, u_t is the upper bound on flows, and $\hat{R}_t$ models the arrival of new resources to be managed. x_t is the decision vector (which may be continuous, discrete or mixed), and $c_t(x_t)$ is a one period contribution function to be maximized. $\mathcal{T} = \{0, 1, \ldots, T-1\}$ represents a set of discrete time points over which our system is defined.

This formulation is not the most general, but it will serve well as an illustration. Our interest is in problems that exhibit a natural state variable. For this purpose, replace equation (2) with the two equations:

$$\begin{aligned} A_t x_t - R_t &= \hat{R}_t && (5) \\ R_t - B_{t-1} x_{t-1} &= 0 && (6) \end{aligned}$$

The variable R_t plays the role of a resource state variable (hence our use of the notation R_t). In this formulation, the effect of all prior decisions is captured by R_t.

We would like to solve this problem in the presence of a sequential information process. In these problem classes, the information available to make decisions evolves over time. Optimization problems in the presence of sequential information processes have been widely studied by a number of different communities who have each evolved their own vocabularies. In our presentation, we are going to focus on problems that arise in resource management, where the decision variables are vectors of possibly very high dimensionality. For this reason, we have adopted a notational system that simplifies the relationship with math programming, since we are going to need the optimization algorithms from this community to solve our problems. Given that we are using x_t as our decision variable, it seems more natural (and more consistent with the operations research community) to use S_t as our state variable, and we use the notation of a value function $V_t(S_t)$ instead of a recourse function (stochastic programming) or cost-to-go function (control theory). We use the important concept of a policy (from dynamic programming) but we give it a specific interpretation that makes it very natural in the context of math programming.

We begin in section 1.1 by setting up the basic notation for our information process. Section 1.2 defines the state variable, which we do in a somewhat nontraditional manner. Section 1.3 describes our decision function and system dynamics, and introduces the concept of a policy as a characterization of a functional approximation for the value function. Finally, section 1.4 gives the objective function and briefly discusses competing techniques for solving this problem class.

1.1 Sequential information processes

Let $(\Omega, \mathcal{H}, \mathcal{P})$ be a probability space with elementary outcome $\omega \in \Omega : w = (\omega_0, \omega_1, \ldots, \omega_T)$. We let W_t be a random variable representing the information arriving at time t, with outcome $W_t(\omega) = \omega_t$. Then, $\{W_t\}_{t=0}^T$ is our stochastic information process. The history of our process is given by:

$$H_t = \{W_0, W_1, \ldots, W_t\} \tag{7}$$

Let $\mathcal{H}_t$ be the $\sigma-$algebra generated by H_t, where $\mathcal{H} = \mathcal{H}_T$.

We distinguish two types of exogenous information:

$W_t =$ (W_t^r, W_t^p), where:

$W_t^r =$ The arrival of new *resources* to the system. For mnemonic purposes, we use the notation $\hat{R}_t = W_t^r(\omega)$.

$W_t^p =$ Information arriving that determines system *parameters*, with sample outcome $\omega_t^p = W_t^p(\omega)$.

For our purposes, we can think of W^p as information about parameters such as contributions c and system dynamics (represented by the matrices A, B, D and u). $W_t^r = \hat{R}_t$ represents new resources arriving to the system. The distinction between information about parameters and resources is important because the decisions x_t act directly on the resources, which then appear in a right hand side constraint (as in equation 2). We may write $c_t(\omega), A_t(\omega)$ and $\hat{R}_t(\omega)$ when we wish to refer to a specific realization of parameters and arrivals.

1.2 State variables

Central to our modeling approach is the careful definition of the sequence of states, actions and information. We have found that, as a result of the structure of our problem, it helps if we adopt a slightly different notation than is commonly used for stochastic optimization problems. Section 1.2.1 outlines the basic ideas in a general way, and section 1.2.2 describes the state variables for our specific problem class.

1.2.1 Some basic concepts

Dynamic systems evolve through a sequence of states, actions and outcomes. These occur in a sequence that is typically written:

$$\{S_0, x_0, \omega_0, S_1, x_1, \omega_1, S_2, x_2, \ldots, \omega_{t-1}, S_t, x_t, \ldots\} \tag{8}$$

Here, it is assumed that S_t contains all the information needed to make decison x_t. For reasons that emerge later, we have adopted a somewhat different notation. Instead of S_t, we use the notation S_t^+ to represent the *complete* state variable, which contains all the information needed to make a decision. We then use the notation S_t to represent an *incomplete* state variable, which captures all the relevant history required to make a decision, but excludes the information that arrived in time period t. In equation (8), ω_t is the information arriving in time period t to be used in time period $t+1$. We prefer to index ω_t by the time period in which the information is actually used. Thus, our sequence of states, actions and outcomes looks like:

$$\{S_0^+, x_0, S_1, \omega_1, S_1^+, x_1, S_2, \omega_2, S_2^+, x_2, \ldots, S_t, \omega_t, S_t^+, x_t, \ldots\} \tag{9}$$

We could have used the notation S_t^- to represent our incomplete state variable, allowing S_t to continue to play the role of the more conventional state variable. Our choice is driven by notational parsimony. We formulate our equations primarily using the incomplete state variable, while we never actually use the complete state variable in our algorithms. We prefer not to carry around the additional superscript. We prefer to index the new information to be used in time period t by t instead of $t-1$ primarily because this produces notation that is more common in the context of deterministic problems. Since we are working on problems which are typically solved in their deterministic form, we felt that it was more appropriate to create notation that was consistent with this literature.

The importance of the incomplete state variable becomes clear in section 2. When we use the complete state variable, we end up with the standard optimality equations:

$$V_t^+(S_t^+) = \max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{V_{t+1}^+(S_{t+1}^+)|S_t^+\right\} \tag{10}$$

When we use the incomplete state variable, our optimality equations take the form:

$$V_t(S_t) = E\left\{\max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t) + V_{t+1}(S_{t+1})|S_t\right\} \tag{11}$$

Equation (11) is much easier to work with than equation (10) since we can drop the expectation and solve the problem for a single sample realization ω_t. If we were to drop the expectation in (10) and solve it for a single sample realization, we would be solving for x_t given ω_{t+1}, which violates standard measurability (or nonanticipativity) conditions.

1.2.2 State variables for dynamic resource management

We assume that we are working on problems that lend themselves to a natural state variable. Interestingly, standard textbooks on dynamic programming do not provide formal definitions of a state variable. In our work, it is important to distinguish between a *resource state* and an *information state*. Strictly speaking, state variables are always information states, but for many problems in operations research, resource states and information states are equivalent. For our work, we do not require them to be equivalent, but we do make the assumption that we are only considering decisions which impact the resource state.

To be precise, we first define:

$a =$ The attribute vector of a resource being managed.

$\mathcal{A} =$ The space of possible resource attribute vectors.

$R_{at} =$ The number of resources with attribute a at time t, before new arrivals have been added.

$\hat{R}_{at} =$ The number of new arrivals in time period t, which we assume can be acted on in time period t.

$$\begin{aligned} R^+_{at} &= \text{The total number of resources which can be acted on in time period } t, \\ &= R_{at} + \hat{R}_{at}. \end{aligned}$$

We refer to $\mathcal{A}$ as the *resource attribute space*, which is the set of possible attributes a resource might have, and R_t as the *incomplete resource state vector* (it is incomplete because it excludes the new arrivals for time period t).

In addition to the arrival of information about resources (captured by $W^r(\omega) = \hat{R}$), we also have information arriving about parameters. We let:

$\rho_t =$ A vector of parameters containing all information at time t about system parameters (such as elements of the matrices A and B, or the contribution c).

We assume that the vector ρ_t is updated to reflect new information using:

$$\rho_t \quad \leftarrow \quad U^p(\rho_{t-1}, \omega^p_t) \tag{12}$$

Equation (12) implies that ρ_t is a *sufficient statistic*, in that it captures all the information required to update model parameters given the latest information captured in ω^p_t.

For our problem, the complete (information) state is given by:

$$S^+_t \quad = \quad \{R^+_t, \rho_t\} \tag{13}$$

In our problem class, we assume that prior decisions $\{x_{t'}\}_{t'<t}$ have an impact only on the (incomplete) resource vector R_t. Later, we use these properties to build functional approximations around R_t alone.

1.3 Decision functions and system dynamics

We now have the foundation to describe our decision process. In deterministic models, it is enough to write out a single optimization problem (as in equations (1) - (4)) to choose a set of decisions.

For stochastic problems, we have to model the time staging of information. For our purposes, it is useful to define:

$X_t^\pi(S_t^+) =$ The decision function, which performs the mapping: $X_t^\pi : \mathcal{S}_t^+ \mapsto \mathcal{X}_t$, where $\mathcal{X}_t$ is the set of possible decisions.

$\Pi =$ The family of functions (or *policies*) for making decisions, where $X_t^\pi, \pi \in \Pi$ is a member of this family.

In discrete dynamic programming, "policies" are often viewed as lookup tables where for a given state s there is an associated action x. Our decision functions serve the same role, but ours generally require solving an optimization problem. Thus, we prefer the view that we are trying to find the best function which will perform this operation for us.

Complementary to the decision function is the *transfer function*. In control theory, the transfer function is typically denoted by f_t. To avoid possible confusion with conventional notation for objective functions, we prefer to write:

$$S_{t+1}^+ = M_t(S_t^+, x_t, \omega_{t+1}) \tag{14}$$

For obvious reasons, the random variable S_{t+1}^+ may be written as $S_{t+1}^+(S_t^+, x_t, \omega_{t+1})$ when the dependence on $(S_t^+, x_t, \omega_{t+1})$ needs to be clear. Otherwise, we assume the dependence to be implicit.

We refer to M_t as the *modify function* (it can also be thought of as the "model" since it captures all the physics of our problem). We find, then, that the decision function and the modify function work in parallel:

$$X_t^\pi : \mathcal{S}_t^+ \mapsto \mathcal{X}_t \tag{15}$$

$$M_t : \mathcal{S}_t^+ \times \mathcal{X}_t \times \Omega_{t+1} \mapsto \mathcal{S}_{t+1}^+ \tag{16}$$

These two functions describe the evolution of information in our system.

1.4 The objective function

We can now provide a formal statement of our problem. Let:

$c_t(S_t^+, x_t) =$ The total contribution generated at time t when the system is in (information) state S_t^+ and we make decision x_t.

Our problem is to solve:

$$\max_{\pi \in \Pi} E \left\{ \sum_{t \in \mathcal{T}} c_t(S_t^+, X_t^\pi) \right\} \tag{17}$$

where the system dynamics are governed by equations (15) - (16). It is important to emphasize that even if we do not write it explicitly, the decision function is a function only of the state variable S_t^+.

The challenge in solving equation (17) is, of course, finding the right function $X_t^\pi(S_t^+)$. For our classes of applications, the decision variables x_t can be large vectors, and solving deterministic instances of the problem, as represented by (1) - (4), can be very difficult. Finding a decision for *each* state S_t^+, then, can appear to be hopelessly intractable. Standard approaches to these problems include the following:

1) Myopic procedures - These include the entire class of "on-line" algorithms, where problems are solved, perhaps to optimality, using only the information known at time t. No attempt is made to capture the impact of decisions at time t on the future.

2) Rolling horizon procedures - Here, we forecast the future (typically using expectations of random quantities) and solve a sequence of problems of the form (1) - (4) over the time periods t to $t + \tau^{ph}$ (where τ^{ph} is the planning horizon). The output of this process is the decision to be made at time t.

3) Stochastic programming - Instead of generating a single, point forecast of the future (as we did with rolling horizon procedures) we generate a set of potential outcomes (scenarios) which we might represent using $\hat{\Omega}$, giving possible events over the period from t to $t+\tau^{ph}$. We would then use the techniques of stochastic programming to determine the right decision to make at time t (see Birge & Louveaux (1997), Infanger (1994), Kall & Wallace (1994)).

4) Dynamic programming - We exploit the structure of the state variable to use the power of the optimality equations to solve the problem (see Puterman (1994)).

All four approaches suffer from major limitations. Myopic procedures can be highly suboptimal for many applications (in particular, nonstationary problems). Rolling horizon procedures are the most widely used tool in practice (particularly for the problem classes we consider here), but effectively ignore the presence of uncertainty. Stochastic programming methods struggle with multistage problems, and often turn what may be already a large optimization problem into one that is much larger. Classical techniques in dynamic programming generally do not even scale to small-sized problems.

In the next section, we outline a computationally tractable solution approach based on the principles of dynamic programming but which avoids all the major limitations of discrete dynamic programming techniques.

2 The solution approach

Solving standard optimality recursions encounters the standard curse of dimensionality. In section 2.1, we present the basic optimality recursion, and show that there are actually three "curses" of dimensionality that arise in the use of dynamic programming. Section 2.2 outlines the strategy for avoiding the three curses of dimensionality, which depends partly on formulating the problem around a special version of the state variable, and partly on the use of carefully constructed value function approximations. Finally, section 2.3 provides two versions of the basic algorithm that we use.

2.1 The dynamic programming recursion

It is well known that problems of the form (17) can be solved via Bellman's optimality equations (see, for example, Puterman (1994)). The optimality equations are given by:

$$V_t^+(S_t^+) \quad = \quad \max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{ V_{t+1}^+(S_{t+1}^+(S_t^+, x_t, W_{t+1})) | S_t^+ \right\} \tag{18}$$

where $\mathcal{X}_t$ captures the feasible region for time period t. We use the notation V_t^+ to emphasize the dependence of the value function on the complete state variable. It is very common to write the optimality equations in terms of a one-step transition matrix. Let:

$p(s'|s, x) =$ The probability the system will be in state s' given that it is in state s and we take action x.

We can then write the optimality equations in their more familiar form (for dynamic programming):

$$V_t^+(S_t^+) = \max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t) + \sum_{s' \in \mathcal{S}_{t+1}^+} p(s'|S_t^+, x_t) V_{t+1}^+(s') \tag{19}$$

The field of dynamic programming has largely evolved around solving the optimality equations using a backward recursion. Assuming that $\mathcal{S}_{t+1}^+$ is discrete, this requires using a known function $V_T^+(S_T^+)$, and then finding each function $V_t^+(S_t^+)$ for all outcomes $S_t^+ \in \mathcal{S}_t^+$ by moving backward in time (a thorough review of these techniques is given in Puterman (1994)). Obviously S_t^+ has to be discrete for this approach to work. However, even if it is discrete, if S_t^+ is a vector, then we encounter the well-known "curse of dimensionality" in dynamic programming which has limited its application to practical problems.

In our problem setting, there are, in fact, three curses of dimensionality:

1) The size of the state space: If S_t^+ is a vector, then the number of possible states grows exponentially as the dimensionality of the state variable grows. This is the classical curse of dimensionality associated with dynamic programs.

2) The size of the outcome space: The expectation in equation (18) requires a summation over the outcomes in Ω. Again, if ω_t is a vector, then this summation is over an exponentially large number of elements. This problem is hidden if we use the version given by equation (19), where the expectation is required to calculate the one-step transition matrix (which is impossible to find for realistic problems). However, even if we did have the one-step transition matrix, equation (19) still requires a summation over the entire sample space.

3) The size of the feasible region: Classical dynamic programming algorithms generally assume that it is possible to search over every possible decision (or action, in the parlance of dynamic programming). This is needed because the value function $V_{t+1}(S_{t+1})$ does not generally have any structure. When x is a vector, the feasible region (again, assuming it is discrete) once again becomes extremely large.

All three of these issues arise because of a dimensionality problem, but in each one, it involves a different vector. Textbook dynamic programming problems are illustrated using problems where the state variable, the exogenous random variable and the decision variable are all scalars. If we are going to use dynamic programming effectively, we have to avoid all three curses of dimensionality.

2.2 Avoiding the three curses of dimensionality

Our solution approach is to use forward dynamic programming techniques in the same general class as those described in Bertsekas & Tsitsiklis (1996) and Sutton & Barto (1998). In this approach, we step forward in time, evaluating the value of being in states that we actually visit, rather than all possible states. Forward dynamic programming approaches partially avoid the first curse of dimensionality by eliminating the need to calculate the value function for every possible state (as we see shortly, these techniques do not entirely avoid the problem with large state spaces). In this section, we show how we can avoid all three curses of dimensionality.

Inherent in forward dynamic programming methods is the lack of an exact value function (as we see below, these are not very useful anyway for our problem class). As a result, we start by replacing the value function with an approximation which we denote $\hat{V}_t^{+}(S_t^{+})$, giving us an equation of the form:

$$\tilde{V}_t^{+}(S_t^{+}) = \max_{x_t \in \mathcal{X}_t} c_t(S_t^{+}, x_t) + E\left\{\hat{V}_{t+1}^{+}(S_{t+1}^{+}(S_t^{+}, x_t, W_t)) | S_t^{+}\right\} \tag{20}$$

The biggest problem with this equation is the expectation, which typically involves summing over an exponentially large outcome space. Using the one-step transition matrix as illustrated in equation (19) does not solve the problem; the expectation is implicit in the calculation of the probabilities, and this equation still requires summing over the state space.

We could avoid the expectation if we simply dropped it and used a single sample estimate (in contrast with the "small sample" suggested earlier). This would mean solving (for compactness, we let $S_{t+1}^{+} = S_{t+1}^{+}(S_t^{+}, x_t, W_{t+1})$):

$$\tilde{V}_t^{n+}(S_t^{+}) = \max_{x_t \in \mathcal{X}_t} c_t(S_t^{+}, x_t) + \hat{V}_{t+1}^{n+}(S_{t+1}^{+}) \tag{21}$$

The problem with this approach is that it determines x_t using ω_{t+1} which violates the measurability of the decision function X_t^{π}. In effect, we are choosing x_t given what will happen in time period $t+1$.

We circumvent this problem by rewriting the optimality equations using the incomplete state variable. For this, we need the following result:

Proposition 2.1 *The optimality equations in terms of the incomplete state variable are given by:*

$$\begin{aligned} V_t(S_t) &= E\left\{\max_{x_t\in\mathcal{X}_t} c_t(S_t^+, x_t) + V_{t+1}(S_{t+1})|S_t\right\} \\ &= E\left\{\max_{x_t\in\mathcal{X}_t} c_t(S_t, W_t, x_t) + V_{t+1}(S_{t+1})|S_t\right\} \end{aligned} \tag{22}$$

Proof: We start by writing $S_t^+ = (S_t, W_t)$. Substituting this into equation (18) gives:

$$V_t^+(S_t, W_t) = \max_{x_t\in\mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{V_{t+1}^+(S_{t+1}, W_{t+1})|S_t, W_t\right\} \tag{23}$$

In the equation above, the maximization operator fixes the value of x_t. Given the value of x_t, information about S_t and W_t gives the information about S_{t+1}. Then (23) can be written as:

$$V_t^+(S_t, W_t) = \max_{x_t\in\mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{V_{t+1}^+(S_{t+1}(S_t, x_t, W_t), W_{t+1})|S_{t+1}(S_t, x_t, W_t), S_t, W_t\right\}$$

Given S_{t+1}, W_{t+1} is independent of W_t and S_t. Writing $S_{t+1} = S_{t+1}(S_t, x_t, W_t)$, the equation above now reduces to:

$$V_t^+(S_t, W_t) = \max_{x_t\in\mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{V_{t+1}^+(S_{t+1}, W_{t+1})|S_{t+1}\right\} \tag{24}$$

We take the conditional expectation of both sides given S_t:

$$E\left\{V_t^+(S_t, W_t)|S_t\right\} = E\left\{\max_{x_t\in\mathcal{X}_t} c_t(S_t^+, x_t) + E\left\{V_{t+1}^+(S_{t+1}, W_{t+1})|S_{t+1}\right\}\middle| S_t\right\} \tag{25}$$

Define:

$$V_t(S_t) = E\{V_t^+(S_t, W_t)|S_t\} \tag{26}$$

Substituting (26) into (25) gives us the desired result. □

A significant practical problem with either form of the optimality equations is the presence of the expectation. For our problem class, random variables are typically vectors (possibly of very high dimensionality) making it completely impossible to calculate an expectation. Our strategy is to drop the expectation and solve the problem for a particular realization ω_t:

$$V_t(S_t,\omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t(\omega_t)} c_t(S_t^+, x_t) + V_{t+1}(S_{t+1}(\omega_t)) \tag{27}$$

Next we exploit the assumption that $S_t^+ = (R_t^+, \rho_t)$ which means that $S_t = (R_t, \rho_{t-1})$. We replace the value function with an approximation $\hat{V}_t(R_t)$, which is a function only of R_t. We also do not write $\hat{V}_t(R_t)$ as an explicit function of ρ_t. This is an approximation which will also introduce errors. We then obtain, at iteration n:

$$\tilde{V}{}_t^n(R_t,\omega_t) \quad = \quad \max_{x_t(\omega_t) \in \mathcal{X}_t(\omega_t)} c_t(S_t^+, x_t) + \hat{V}_{t+1}^n(R_{t+1}(\omega_t)) \tag{28}$$

Once we solve (28), we then use $\tilde{V}{}_t^n(R_t)$ to update our approximation $\hat{V}_t^n(R_t)$, which we represent using:

$$\hat{V}_t^{n+1}(R_t) \quad \leftarrow \quad U^V(\hat{V}_t^n, \tilde{V}{}_t^n, R_t^n) \tag{29}$$

where $U^V()$ is the function which updates the value function approximation using the information gained from the most recent solution of (28), captured by the function $\tilde{V}{}_t^n(R_t)$ and the (resource) state R_t^n that we visited in iteration n at time t.

Of course, the solution that we obtain from solving this problem is only a sample realization, but our goal is not to find the optimal action (for a particular outcome) but rather to build the function $X_t^\pi(S_t^+)$ that can be used to find an action given a state $S_t^+ = (S_t, \omega_t)$. Our "policy" is to solve functions of the form:

$$X_t^\pi(S_t^+, \hat{V}_{t+1}) \quad = \quad \arg\max_{x_t \in \mathcal{X}_t(\omega_t)} c_t(S_t^+, x_t) + \hat{V}_{t+1}(R_{t+1}(\omega_t)) \tag{30}$$

Choosing the best policy, then, can be viewed as choosing the best function $\hat{V}$. The abstract notion of optimizing over the set of policies (as in equation (17)) can be stated in terms of finding the best functional approximation. The space of policies Π can now be characterized as the space of possible functional approximations.

We have now solved the problem of two of the three curses of dimensionality. The last one is the feasible region (action space). Recall that we are working on problems where, in general, the one period problem $\max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t)$ requires the machinery of optimization algorithms. This might be a linear program, a network, or an integer programming problem. We assume that the one-period problem is basically solvable (that is, we are not trying to solve NP-hard problems), but they will only remain solvable if we choose an approximation $\hat{V}_{t+1}(R_{t+1})$ that does not destroy the structure of the original problem. A discrete table-lookup form of value function (where for any vector R_{t+1} we can find $\hat{V}_{t+1}(R_{t+1})$) is usually of little value. For example, if $\max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t)$ is a network problem, a discrete representation of the value function destroys the network structure.

We solve the third curse of dimensionality, then, by requiring that we choose a functional form for the value function approximation so that the optimization problem in equation (28) is computationally tractable. It is especially nice when the structure of (28) is the same as the structure of $\max_{x \in \mathcal{X}} c_t(S_t^+, x_t)$, but this is not necessary. For example, the one period problem might be a network (with nice integrality properties); it is convenient, then, when the value function approximation does not destroy the network structure.

Choosing a good value function approximation is, of course, highly problem dependent. For this reason, section 3 illustrates the issues that arise in the development of value function approximations in the context of a series of classical optimization problems.

2.3 The general algorithm

We close our presentation of the basic strategy by summarizing the specific steps of the algorithm. To be a true algorithm, we would have to specify the form of $\hat{V}_t(R_t)$ and the updating scheme $U^V()$. We address these issues in section 3. Here, we outline the overall procedure.

There are two versions of the algorithm: a single pass version, where all the calculations take place during the forward pass, and a double pass version, where the updating of the value functions occurs during a separate backward pass. The single pass version is given in figure 1.

The double pass version of the algorithm is given in figure 2. The key difference is that we do not update the value function during the forward pass. Instead, we find all the decisions (for a given ω) and then use equation (31) to recompute $\tilde{V}_t^n\,(R_t)$ before updating $\hat{V}_t^n(R_t)$. We use the same sample outcome, and the same decision that was determined in the forward pass.

The forward pass version is, of course, easier to implement. In large scale applications, it may

```
STEP 0: Initialization:
    Initialize $\hat{V}_t^0, \quad t \in \mathcal{T}$.
    Set $n = 0$.
    Initialize $R^0$.
STEP 1: Do while $n \leq N$:
    Choose $\omega = (\omega_1^n, \omega_2^n, \ldots, \omega_T^n)$
    STEP 2: Do for $t = 0, 1, \ldots, T$:
        STEP 2a: Solve equation (28) to obtain $\tilde{V}_t^n\ (R_t)$ and $x_t^n$.
        STEP 2b: Compute $R_{t+1}^n = M_t(R_t^n, x_t^n, \omega_t^n)$.
        STEP 2c: Update the value function approximations, $\hat{V}_t^n$ using $U^V(\hat{V}_t^n, \tilde{V}_t^n, R_t^n)$.
STEP 3: Return the policy $X_t^\pi(S_t^+, \hat{V}^N)$.
```

Figure 1: Single pass version of the adaptive dynamic programming algorithm

be necessary to retain a lot of information that was generated in the solution of $\tilde{V}_t^n$ at time t during the forward pass, in order to effectively recompute the information needed in the backward pass to perform the updating (needless to say, this is problem dependent). However, we have found in some applications that the two-pass version dramatically improves the overall rate of convergence.

Our updating scheme is closest in spirit to that proposed by Bertsekas & Tsitsiklis (1996) [pages 269-275], although we differ in several critical aspects. The most important is that our use of the incomplete state variable allows us to use a single Monte Carlo sample to solve a single, deterministic optimization problem at each time period. By contrast, Bertsekas and Tsitsiklis assume that you will sample a number of forward trajectories as part of the updating process. For large-scale optimization problems, this is a significant computational savings. The use of forward and backward passes is, of course, unique to finite horizon problems. We compare our approach to neuro-dynamic programming as well as multistage stochastic programming in section 4.

```
STEP 0: Initialize $\hat{V}_t^0, \quad t \in \mathcal{T}$.
        Set $n = 0$.
        Initialize $R^0$.
STEP 1: Do while $n \le N$:
        STEP 2: Do for $t = 0, 1, \ldots, T-1$:
            STEP 2a: Choose a sample $\omega_t^n$.
            STEP 2b: Solve equation (28) to obtain $\tilde{V}_t^n\ (R_t)$ and $x_t^n$.
            STEP 2c: Compute $R_{t+1}^n = M(R_t^n, x_t^n, \omega_t^n)$
        STEP 3: Do for $t = T-1, T-2, \ldots, 1, 0$:
            STEP 3a: Recompute $\tilde{V}_t^n\ (S_t)$ using $\hat{V}_{t+1}^{n+1}$ and the decision $x_t^n$ from the forward pass:
                $\tilde{V}_t^n\ (S_t) = c_t(S_t^+, x_t^n) + \hat{V}_{t+1}^{n+1}(S_{t+1}(\omega_{t+1}^n))$    (31)
            STEP 3b: Update the value function approximations, $U^V(\hat{V}_t^n, \tilde{V}_t^n, R_t^n)$.
STEP4: Return policy $X_t^\pi(S_t^+, \hat{V}^N)$.
```

Figure 2: Double pass version of the adaptive dynamic programming algorithm

3 Approximating the value function

In our applications with vector-valued decision variables, the solution of a single subproblem $\tilde{V}_t^n$ (R_t^n, ω_t^n) will generally require the use of an optimization algorithm. For this reason, it is important that $\hat{V}$ be chosen so that the optimization problem is computationally tractable. It is generally the case, for example, that a discrete value function approximation is of no use, since we are not going to be able to evaluate each possible decision. Rather, we need to use value function approximations for $\hat{V}$ that provide sufficient structure that allow $\tilde{V}_t^n$ to be solved using a computationally tractable search algorithm.

The importance of retaining appropriate structural properties when formulating a value function approximation has received virtually no attention in the dynamic programming literature. The solution of vector-valued stochastic optimization problems has been addressed primarily in the stochastic programming literature, which completely ignores the use of state variables and the power of dynamic programming recursions.

The dynamic programming community has pursued two basic paths for estimating value functions: discrete, table-lookup functions, and parameterized functional approximations. A discrete, table-lookup function assumes that we have an estimate $\hat{V}(s)$ for each value of s. The most basic forward algorithms provide an updated estimate of $\hat{V}(s)$ each time state s is visited. More sophisticated strategies allow the updating of more than one state with each visit of a state. However, all of these strategies produce only a discrete value function which we are generally unable to use in the optimization of vector-valued optimization problems.

One of the limitations of table-lookup representations of value functions is that we are effectively estimating $|\mathcal{S}|$ parameters ($\hat{V}(s)$ for each element of $\mathcal{S}$). Not surprisingly, then, a number of authors have looked for methods which characterize the value function using fewer parameters. Beale, Forest & Taylor (1980) is one of the earliest to have suggested the use of response surface methods, whereby the value function is assumed to take the form of a regression equation with a few parameters. A more modern and thorough treatment of this approach is provided by Tsitsiklis & Van Roy (1997), who propose that an approximate value function may be

$$\hat{V}(s) \quad = \quad \sum_{k \in \mathcal{K}} \rho_k \phi_k(s) \tag{32}$$

where $\{\phi_k(s)\}_{k \in \mathcal{K}}$ is a set of *basis functions* weighted by the parameters $\{\rho_k\}_{k \in \mathcal{K}}$. The idea is that the cardinality of the set $\mathcal{K}$ would be fairly low, producing a much smaller set of parameters to be estimated. The parameters $\{\rho_k\}_{k \in \mathcal{K}}$ are estimated using regression techniques or neural-networks by generating a sample of estimates of the value function (see Bertsekas & Tsitsiklis (1996)). Of particular value is that we obtain an estimate of $\hat{V}(s)$ for states s that we have never visited. Of course, the challenge is deciding on the set of basis functions. Bertazzi, Bertsekas & Speranza (2000) experiment with basis functions that are linear or quadratic in the (resource) state variable, but consider only a fairly small problem where the issue of solving the one-period problem does not arise.

An overlooked feature of functional approximations is that they may provide us with the structure that we need to solve the optimization problem $\widetilde{V}$. This leaves us with the challenge of finding the right basis functions. For this purpose, we depend on the structure of our resource management problems. First, we assume that decisions x_t affect only the resource vector R_{t+1}. Second, we observe that the resource vector R_t plays the role of constraining the system (our decisions act on resources, and flow conservation limits our actions). As a result, the value functions tend to have properties such as concavity (when we are maximizing) and/or monotonicity. It is also the case that separable approximations are simply easier to work with, and tend to retain problem structure. For these reasons, we are particularly interested in two classes of functional approximations:

$$\hat{V}_t(R_t) = \sum_{a\in\mathcal{A}} v_{at}R_{at} \tag{33}$$

$$\hat{V}_t(R_t) = \sum_{a\in\mathcal{A}} \hat{V}_{at}(R_{at}) \tag{34}$$

We refer to equations of the form of (33) as *linear* because they are linear in the (resource) state variable (by contrast, Tsitsiklis & Van Roy (1997) refer to linear approximations which are linear in the parameters, which of course ours are). We refer to equations of the form of (34) as *nonlinear, separable* because they are nonlinear and separable in the resource state (but, they are not necessarily linear in the parameters). In fact, our nonlinear approximations can not be represented in the format implied by equation (32). For the remainder of this paper, linear approximations always refer to linear in the resource state, and nonlinear approximations always refer to approximations that are both nonlinear and separable in the resource vector. We let:

$\mathcal{V}^L$ = The space of all possible linear approximations (which is equivalent to the set of all vectors v).

$\mathcal{V}^{NLS}$ = The space of all possible nonlinear, separable approximations.

Specifying a linear approximation involves, of course, nothing more than determining the slope with respect to the resource vector. By contrast, there is a much richer variety of nonlinear functions we may choose from, even when we restrict our attention to separable approximations. In practice, linear approximations are much easier to work with. Not only are they easier to estimate and store (this can be an issue with some large-scale applications that arise in practice), they naturally produce monotone functions without the overhead of explicitly maintaining monotonicity. Of course, value functions, as a rule, are not linear, and we can expect that nonlinear approximations will perform better. In our work, we do not generally work with nonlinear functions of the form in (32); for example, a piecewise linear function does not fit this form.

Estimating linear or separable nonlinear functions is fairly easy. Let:

$\tilde{v}_t^n$ = A sample estimate of the slope of $\tilde{V}_t^n (R_t)$ at $R_t = R_t^n$.

Below, we illustrate problems where $\tilde{v}_t^n$ can be estimated using finite difference methods or from dual variables of linear programs. If we are estimating a linear approximation, then we would update our estimate of the slope using:

$$\hat{v}_t^{n+1} = (1-\alpha^n)\hat{v}_t^n + \alpha^n \tilde{v}_t^n \tag{35}$$

There are several methods for estimating separable, nonlinear functions. One of the simplest is to use the SHAPE algorithm of Cheung & Powell (2000). Here, we start with an appropriate nonlinear function $\hat{V}_{it}^0(R_{it})$. For example, for some problem classes (which include multistage linear programs) it is possible to show that the value function is concave (when we are maximizing). For this problem class, we might choose functions such as:

$$\begin{aligned}
\hat{V}^0(R) &= \rho_0 \left(1 - e^{-\rho_1 R}\right) \\
\hat{V}^0(R) &= \ln(R+1) \\
\hat{V}^0(R) &= -\rho_0 (R-\rho_1)^2
\end{aligned}$$

Given an initial function, the SHAPE algorithm updates the function at each iteration using:

$$\hat{V}^{n+1}(R) = \hat{V}^n(R) + \alpha^n \left(\tilde{v}^n - \nabla \hat{V}^n(R^n)\right) R \tag{36}$$

where $\nabla \hat{V}^n(R^n)$ is the gradient of $\hat{V}$ with respect to R. This algorithm has been shown to be optimal for two-stage problems with continuously differentiable value functions, but can be used as a heuristic in the context of multistage problems (for a closely related algorithm, see also Culioli & Cohen (1990)).

One limitation of the SHAPE algorithm is that it requires that we assume some initial functional form. The successive updates tilt this function, but do not change the shape. We have found that this can be a problem in the context of discrete applications where value functions tend to be piecewise linear, but may not follow a particularly nice shape. One solution to this is the CAVE algorithm (Godfrey & Powell (2001)). This algorithm works best when we have access to left and right derivatives, which we denote $\tilde{v}_t^{n+}$ and $\tilde{v}_t^{n-}$. If the problem is concave, we expect that $\tilde{v}_t^{n-} \geq \tilde{v}_t^{n+}$. However, because these are sample estimates, we encounter the problem of smoothing these sample estimates with our current estimate of $\hat{V}_t^n(R_t)$, since we may produce an updated function that violates concavity. The CAVE algorithm provides specific steps that ensure that the updated function $\hat{V}_t^{n+1}(R_t)$ are also concave. We represent the updating of $\hat{V}$ using the CAVE algorithm using:

$$\hat{V}^{n+1} \quad \leftarrow \quad U^{CAVE}(\hat{V}^n, \tilde{v}^{n-}, \tilde{v}^{n+}, R^n) \tag{37}$$

The CAVE algorithm can be used even without the left and right derivatives by simply letting $\tilde{v}^- = \tilde{v}^+ = \tilde{v}$. If we do have access to the left and right derivatives, we can also use a two-sided updating scheme for the SHAPE algorithm, as follows:

$$\hat{V}^{n+1}(R) \quad = \quad \begin{cases} \hat{V}^n(R) + \alpha^n \left(\tilde{v}^{n+} - \partial\hat{V}^{n+}(R^n)\right) R & R \geq R^n \\ \hat{V}^n(R) + \alpha^n \left(\tilde{v}^{n-} - \partial\hat{V}^{n-}(R^n)\right) R & R \leq R^n \end{cases} \tag{38}$$

If we use the two-sided updating process, then the initial choice of the approximating function becomes less important, as each update will actually change the shape of the function.

4 Comparison to other approaches

Now that we have our basic solution strategy in hand, it is useful to compare our method to two lines of investigation that have been pursued in the research literature: forward methods in dynamic programming, and multistage stochastic linear programming. We start in section 4.1 by describing forward methods in dynamic programming since these are most closely related to our own approach. Our problem class is closer to multistage stochastic linear programming, and in section 4.2 we contrast the basic technique behind a dynamic programming-based approach to that used in the stochastic programming literature. We close with a few remarks in section 4.3.

4.1 Forward methods in dynamic programming

We are aware of two general strategies for handling the problem of large state spaces: state space aggregation, and forward methods. Aggregation methods have often been proposed as a method for solving the curse of dimensionality problem in dynamic programmig (Bean, Birge & Smith (1987)). We do not consider state space aggregation methods in this paper because they produce table-lookup representations of value functions which cannot be used when the one-period problem involves optimizing vector-valued decisions over large feasible regions.

Considerably more promising are forward methods in dynamic programming, which depend on moving forward in time, using an approximation of the value function at each iteration (see Bertsekas & Tsitsiklis (1996) and Sutton & Barto (1998) for thorough discussions of these techniques). The most basic method for approximating the value function is to iteratively solve problems of the form (at iteration n):

$$\tilde{V}_t^n(S_t^+) = \max_{x_t \in \mathcal{X}_t} c_t(S_t^+, x_t) + \sum_{s' \in \mathcal{S}} p(s'|S_t^+, x_t)\hat{V}_{t+1}^n(s') \tag{39}$$

Note that we use the notation $\tilde{V}_t^n(R_t)$ to represent the left hand side of equation (39), whereas we use $\hat{V}_{t+1}^n(R_{t+1})$ on the right hand side. $\tilde{V}_t^n(R_t)$ is a form of place-holder, which we are going to use to update our value function approximation for time period t. We then update the value function $\hat{V}_t^n(S_t^+)$ using:

$$\hat{V}_t^{n+1}(S_t^+) \leftarrow (1-\alpha^n)\hat{V}_t^n(S_t^+) + \alpha^n \tilde{V}_t^n(S_t^+) \tag{40}$$

Equation (40) smooths the old estimate of the value of being in a particular state with an updated estimate, based on solving an approximation of the problem. The stepsize α^n should satisfy the standard conditions:

$$\begin{aligned} \sum_{n=0}^{\infty} \alpha^n &= \infty \\ \sum_{n=0}^{\infty} (\alpha^n)^2 &< \infty \end{aligned}$$

This technique requires starting with an approximation $\hat{V}_t^0(S_0^+)$. As a general rule, this approximation should be optimistic so that we do not ignore potentially good decisions. Proofs of convergence require assumptions that every state be visited infinitely often (the practical equivalent of this requirement is that we would require that every state have a strictly positive probability of being visited at each iteration), a condition that would generally not be met in practice (furthermore, the failure to visit each state with a strictly positive probability can also produce noticeable errors). As a result, the techniques have to be viewed as heuristics.

There are a number of variations of how this method would be implemented. A major weakness of classical forward methods arises when there is a large state space, since we have to sample each state a number of times to get a reasonable estimate of the value of the state. Forward methods partially overcome the problem of large state spaces since they focus on the states that are actually visited, but even this set of states can be extremely large.

Step 1 For all t set $\hat{V}_t^0(s) = c_t(s)$ for all states s. Let $n = 0$.

Step 2 For $t = 0, ..., T-1$ set the decision rule to be:

$$X_t^{\pi,n}(S_t^+) \quad = \quad \arg\max_{x \in \mathcal{X}_t} c_t(S_t^+, x) + E\left\{\hat{V}_{t+1}^n(S_{t+1}^+)|S_t^+\right\} \tag{41}$$

Step 3 Select a set of states $\hat{\mathcal{S}}_t^n \in \mathcal{S}_t^+$ at random. For each state $s \in \hat{\mathcal{S}}_t^n$, generate a trajectory of length τ starting at $s_0 = s$, using the decision rule $X_t^{\pi,n}$. Let $\tilde{\mathcal{S}}_t^n$ be the set of states from all the generated trajectories at time t. Calculate the discounted contribution $\tilde{c}_t^n(s)$ which is the contribution of the trajectory starting in state s at time t at iteration n. Then for each time t solve the following least squares optimization problem to find β_t^n:

$$\min_\beta \sum_{s \in \tilde{\mathcal{S}}_t^n} \sum_{m=1}^{|\hat{\mathcal{S}}_t^n|} \left\{\hat{V}_t^n(s, \beta_t^n) - \tilde{c}_t^n(s)\right\}^2 \tag{42}$$

where

$$\hat{V}_t^n(s, \beta_t^n) = \sum_{k \in \mathcal{K}} \beta_{kt}^n \phi_k(s)$$

and where $\phi_k(s)$ are predetermined functions of the state variable (called basis functions). Perform appropriate smoothing on the parameters β_t^n.

Step 4 Let $n := n + 1$, and go to step 2.

Figure 3: Sketch of a neuro-dynamic programming algorithm for a finite-horizon problem

Bertsekas & Tsitsiklis (1996) (see also Tsitsiklis & Van Roy (1997)) propose the use of neural networks to help overcome the problem of estimating the value of being in each state by instead estimating a statistical function of the form given in equation (32). Figure 3 provides an outline of a basic implementation of the neuro-dynamic programming concept adapted to a finite-horizon problem (see Bertazzi et al. (2000) for an illustration of this procedure to a problem in inventory management, also in the context of steady state problems).

Perhaps the biggest limitation of this strategy, as it has been presented by these authors, is that it depends on solving equations of the form given in equation (39). Even if we can develop a good approximation, the explicit or implicit use of the expectation encounters the second curse of dimensionality: the outcome space. In short, computing expectations (which includes finding one-step transition matrices) is, in general, computationally intractable. Bertsekas & Tsitsiklis

(1996) address this [see pages 266-267] by suggesting that the expectation can be approximated using simulation. Let $\hat{\Omega}_t$ be a sample of exogenous outcomes for time period t drawn from Ω_t. Equation (41) would then be replaced with:

$$X_t^{\pi,n}(S_t^+) = \arg\max_{x \in \mathcal{X}_t} c_t(S_t^+, x) + \sum_{\omega'_{t+1} \in \hat{\Omega}_{t+1}} \hat{V}_{t+1}^n(S_{t+1}(\omega'_{t+1}))p(\omega'_{t+1}) \tag{43}$$

If we use a discrete representation for $\hat{V}_{t+1}(S_{t+1})$ then the only way to solve equation (43) is to evaluate it for each possible value of $x \in \mathcal{X}_t$. This is computationally intractable for our problem class, where x is typically a vector, possibly of large dimension. But consider instead the case where $\hat{V}_{t+1}(S_{t+1})$ follows one of the two functional approximations that we propose (equations (33) and (34)). The linear case again offers special simplifications. Assume as we have been doing that $\hat{V}_{t+1}(S_{t+1}^+) = \hat{V}_{t+1}(R_{t+1}^+)$ where $R_{t+1}^+ = R_{t+1}(x_t) + \hat{R}_t$ (we write $R_{t+1}(x_t)$ explicitly as a function of x_t to emphasize this dependence). Using this relationship and a linear approximation means (43) becomes:

$$\begin{aligned} X_t^{\pi,n}(S_t^+) &= \arg\max_{x \in \mathcal{X}_t} c_t(S_t^+, x) + \sum_{\omega'_{t+1} \in \hat{\Omega}_{t+1}} \hat{v}_{t+1}^n \left(R_{t+1}(x_t) + \hat{R}_{t+1}(\omega'_{t+1}) \right) p(\omega'_{t+1}) \\ &= \arg\max_{x \in \mathcal{X}_t} c_t(S_t^+, x) + \hat{v}_{t+1}^n R_{t+1}(x_t) + \sum_{\omega'_{t+1} \in \hat{\Omega}_{t+1}} \hat{v}_{t+1}^n \hat{R}_{t+1}(\omega'_{t+1}) p(\omega'_{t+1}) \end{aligned} \tag{44}$$

The last term on the right hand side of (44) is not a function of x_t and can be dropped. The remaining problem is the first stage problem plus a linear term which never destroys problem structure. Thus, if we use a linear approximation, our method and that described in figure 3 differ only in how the value function is estimated (we do not solve equation (42)).

The difference in the methods is more pronounced if we use a separable, nonlinear value function approximation. In this case, equation (43) becomes:

$$X_t^{\pi,n}(S_t^+) = \arg\max_{x \in \mathcal{X}_t} c_t(S_t^+, x) + \sum_{\omega'_{t+1} \in \hat{\Omega}_{t+1}} \sum_{i \in \mathcal{I}} \hat{V}_{i,t+1}^n \left(R_{t+1}(x_t) + \hat{R}_{t+1}(\omega'_{t+1}) \right) p(\omega'_{t+1}) \tag{45}$$

If $\hat{V}_{i,t+1}^n(R_t^+)$ is written as a piecewise linear function, then equation (45) can also be written as a linear program (if $\max_{x \in \mathcal{X}_t} c_t(S_t^+, x)$ is a linear program). Beyond this, we have probably lost any convenient structure that may have existed in our original one-period problem. As a result, equation (45) can be computationally intractable for vector-valued problems.

There is a second issue arising in the solution of equation (45). Obviously, if $\hat{\Omega}_{t+1}$ has a single observation, then we violate measurability by allowing x_t to "see" ω_{t+1}. We would be choosing x_t based on an assumed outcome for time $t+1$, thereby biasing both our policy and the value function estimate it would produce. This problem is mitigated as $|\hat{\Omega}_{t+1}|$ is increased, but not entirely eliminated, and the price we have to pay is a larger optimization problem. For most problems in resource allocation, using $\hat{\Omega}_{t+1}$ with more than one sample would be extremely difficult computationally. Aside from the fact that some deterministic problems are extremely large, we also have the challenge of dealing with integer variables. Even a simple problem such as a pure network or assignment problem loses nice integrality properties when $\hat{\Omega}_{t+1}$ has more than one sample.

4.2 Multistage linear programming

Problems where the one-period problem $\max_{x \in \mathcal{X}_t} c_t(S_t^+, x)$ is a linear program are most commonly approached (in the research community) using the techniques of stochastic linear programming (in practice, the most common approach is to ignore the uncertainty altogether). Given the extensive literature in this area, we make no effort here to review the field, referring the reader instead to a series of excellent reviews and books (Sen & Higle (1999), Birge & Louveaux (1997), Infanger (1994) and Kall & Wallace (1994)). The point of departure from stochastic dynamic programming and multistage linear programming starts with equation (43). In dynamic programming, we assume the presence of a reasonably compact (resource) state variable R_t and use this to develop functional approximations. In stochastic linear programming, equation (43) is typically replaced with:

$$X_t^{\pi,n}(S_t^+) = \arg\max_{x_{\mathcal{T}_t} \in \mathcal{X}_{\mathcal{T}_t}} \sum_{\omega \in \hat{\Omega}} \sum_{t' \in \mathcal{T}_t} c_{t'} x_{t'}(\omega) p(\omega) \tag{46}$$

where:

$\mathcal{T}_t =$ The set of time periods constituting a planning horizon at time t.

$x_{\mathcal{T}_t}(\omega) = \{x_{t'}(\omega)\}_{t' \in \mathcal{T}_t}, \omega \in \hat{\Omega}$

$\mathcal{X}_{\mathcal{T}_t} =$ The feasible region for the vector $x_{\mathcal{T}_t}$.

As we have written it, solving equation (46) means that we are choosing x_t given an entire vector of outcomes specified by choosing ω. In fact, we have allowed ourselves to choose a value of x_t for *each* ω, which is the same as making decisions given the future. To avoid this behavior, we add to our basic constraints $\mathcal{X}_{\mathcal{T}_t}$ the *nonanticipativity* constraints:

$$x_t(\omega) - \bar{x}_t \quad = \quad 0 \quad \forall \omega \in \hat{\Omega}_{\mathcal{T}_t} \tag{47}$$

Equation (47) simply requires that $x_t(\omega)$ be the same for all ω. For multistage problems, we should also impose similar restrictions for $x_{t'}, t' = t+1, t+2, \ldots$ conditioned on the history of the process up to time t' (see Mulvey & Vladimirou (1991*b*), Glockner & Nemhauser (2000)). However, if we wish to get a reasonable number of scenarios starting at time t' for a given history up to time t', then the number of scenarios required to even approximately capture a multistage problem grows exponentially with the planning horizon. A common approximation is to simply enforce nonanticipativity for the first time period and then model all subsequent time periods as a single information stage. Thus, decisions at time $t' > t$ are allowed to see the future information stream. We are not aware of any careful study of the errors introduced by this approximation.

Equation (46) implies that we are solving a linear program that is approximately $|\hat{\Omega}|$ times larger than the original one-period program. For the problem class that we are considering, such an optimization problem can be *very* difficult, if not completely intractable (see Mulvey & Ruszczynski (1995) for work on specialized algorithms for this problem class). For example, our basic problem may consist of networks where integer solutions are desirable. The nonanticipativity constraint typically destroys any imbedded network structure, making the presence of integrality constraints problematic (see Louveaux & van der Vlerk (1993), Laporte & Louveaux (1993) for examples of work involving integer variables in stochastic programming).

Scenario methods in stochastic linear programming have proven effective for problems in financial risk management where there are complex correlations not only across random variables at the same time period, but also across time periods (Mulvey & Vladimirou (1992)). These problems typically exhibit network structure, but do not require integer variables. The deterministic versions of these problems are also typically not very large, so the problems lend themselves quite well to this approach.

A method that is closer to our view of the world is nested Bender's decomposition using Monte Carlo sampling. Benders decomposition was first proposed for stochastic programming by Van Slyke & Wets (1969), and extended to multistage problems by Birge (1985). These methods both work on a fixed set of scenarios, and are computationally intractable in the multistage setting. A significant breakthrough in this area of research is the development of stochastic decomposition (Higle & Sen

(1991)) which uses Monte Carlo sampling to generate sequences of relatively small linear programs (avoiding the problems of scenario methods). In this method, we would solve sequences of problems that look like:

$$\min_{x_0} \quad c_0^T x_0 + z$$

subject to:

$$\begin{aligned} A_0 x_0 &= b_0 \\ z + \beta(\omega)^T x_0 &\geq \alpha(\omega), \quad \forall \;\; \omega \in \hat{\Omega}^n \\ x_0 &\geq 0 \end{aligned}$$

where $\hat{\Omega}^n$ is the set of scenarios that have been sampled by iteration n, and $\alpha(\omega)$ and $\beta(\omega)$ are coefficients that were estimated for scenario ω from the dual of the second stage problem.

This basic idea has been extended by several authors to multistage problems (Pereira & Pinto (1991), Archibald, Buchanan, McKinnon & Thomas (1999), Infanger & Morton (1996)). We are not aware of a proof of convergence of nested Benders for multistage. Chen & Powell (1999*b*) give a proof of convergence for a nested Benders algorithm which assumes a finite (although possibly quite large) sample space. Experimental evidence for this approach is limited but promising. Particularly attractive is the property that we are solving sequences of reasonably small, single period problems. Of course, the method is restricted to continuous multistage linear programs, whereas we often encounter integer problems. Also, nested Benders does not take advantage of the compactness of the resource state variable, which is typically of much lower dimensionality than the decision vector x.

4.3 Remarks

Dynamic programming methods work best for problems where there is a natural state variable. When we use functional approximations, we do not require the state space to be small, but we do assume that the dimensionality of the vector R_t is not "too large." For example, let $R_t = \{R_{at}, a \in \mathcal{A}\}$ where a is an "attribute vector" that describes a resource. If a is a vector, the space $\mathcal{A}$ may be extremely large, and this would complicate the process of finding either separable, nonlinear approximations, or perhaps even linear approximations. The technique has been applied to a problem where the vector a had six dimensions, producing a resource state space $\mathcal{A}$ with over a million elements (Powell, Shapiro & Simao (2000)), but the algorithm took advantage of the fact that only a few thousand of these elements were actually used.

It is normal in dynamic programming to assume that the problem must also possess a Markovian structure. This is true for exact solutions, but our method never made this assumption, since the sampling from Ω may take into account complex correlations between stages. We did, however, approximate the value function by formulating approximations purely around the resource variable. Errors will be introduced by ignoring the evolution of other forms of information, but errors are also introduced in stochastic linear programming when nonanticipativity constraints in future time periods are ignored. Extensive experimental work would be needed to evaluate the effects of these different approximations.

5 Applications

Structural properties are best illustrated in the context of specific problems. In the sections that follow, we illustrate the use of value function approximations in the context of a range of problems that have been widely studied in operations research. Each section is based on the work of another paper which explores in depth technical issues that arise in the context of the specific application. Here, our focus is putting all the problems in the context of a single algorithmic strategy and illustrating the effect of different approximation strategies on the underlying problem structure.

We begin in section 5.1 by presenting an aging and replenishment process which arises in the context of basic inventory and batch processing systems (see Papadaki & Powell (2001*b*) for a more complete discussion of this problem in the context of a single link dispatching problem). We use this simple problem, which is the only one we can solve to optimality (in its stochastic form), to illustrate the importance of retaining problem structure. Next, section 5.2 considers a single commodity (dynamic) network flow problem (see Godfrey & Powell (1999)), where we illustrate the effectiveness of nonlinear value function approximations. Section 5.3 then considers integer, stochastic multicommodity network flow problems (see Topaloglu & Powell (2000)), where we show that linear approximations produce pure network subproblems that are especially easy to solve (with integer solutions), whereas nonlinear approximations are somewhat harder. Finally, section 5.4 describes a dynamic assignment problem, where we have to assign resources to tasks over time (see Spivey & Powell (2000) for a description of this problem class). In this setting, we show that a particular nonseparable approximation can produce tractable subproblems.

Our focus throughout this section is to illustrate the importance of exploiting problem structure and the challenges that this poses when we design a functional approximation. The first application below is a classical discrete dynamic program. All of the rest are different types of multistage stochastic linear or integer programs.

5.1 Aging and replenishment processes

Aging and replenishment processes arise in a variety of applications where there is a nonnegative stochastic process that drifts exogenously to the right (left) with controls that shift the process to the left (right). Perhaps the most common instance of this problem class is inventory management (which has an exogenous leftward shift as demand depletes inventory), but other applications arise in areas such as machine maintenance. The problem is interesting because in its simplest form, it is easy to solve to optimality. We then study our different approximation strategies and show that the most basic forward algorithm does not work if we do not properly exploit the structure of the problem. Then we show that both the linear and nonlinear approximations work quite well, even on a problem that is neither convex nor concave (although it is monotone).

The basic aging and replenishment process can be described as:

$R_t =$ The (resource) state of the system at time t.

$D_t =$ The exogenous (rightward) drift.

$x_t =$ The endogenously controlled (leftward) shift.

$c^s_t(S_t) =$ The contribution from being in (information) state S.

$c^c_t(S_t, x_t) =$ The one-period contribution from applying control x.

If this were an inventory problem, there would be a leftward drift (as product is consumed) and a rightward shift (as product is replenished). $c^s(S)$ would be the holding cost (applied to product left in the system at the end of the time period) and $c^c(S, x)$ would capture the ordering cost (typically a fixed cost for placing an order, and a variable cost that reflects the amount of the order). In this setting, we would normally minimize costs rather than maximize contribution.

The dynamics of our system are written simply as:

$$R_{t+1} \quad = \quad R_t + D_t - x_t$$

where we assume that $x_t \geq R_t + D_t$. The information state variable is $S_t^+ = (R_t, D_t)$, and the incomplete state variable, as before, is $S_t = R_t$ (there is no other information process involved here). The one-period contribution function is:

$$c_t(S_t, x_t) \quad = \quad c^s(S_t) + c^c(S_t, x_t)$$

The objective function is given by equation (17), and the optimality equations are written simply as:

$$V_t(S_t) \quad = \quad E\left\{\max_{x_t \geq 0} c_t(S_t, x_t) + V_{t+1}(S_{t+1})|S_t\right\} \tag{48}$$

This is a very simple problem to solve since, in its most basic form, S_t, D_t and x_t are all scalars. Equation (48) can be easily solved using the standard backward recursion of dynamic programming. Of course, real problems involve multiple products with some combination of product substitution or joint production economies, in which case the state, outcome and decision variables are all vectors.

The simple scalar problem, however, is useful for illustrating the importance of structural properties when we approximate the value function. We start by considering a version of an aging and replenishment process that arises in the context of a simple vehicle dispatching vehicle where we seek to minimize dispatch and holding costs. In this setting, let:

$D_t =$ Number of customers arriving in time period t to the terminal.

$K =$ The capacity of the vehicle.

$c^h =$ The cost of holding a customer.

$c^d =$ The cost of dispatching the vehicle.

$z_t =$ 1 if the vehicle is dispatched, and 0 otherwise.

The shift x_t is given by:

$$x_t \quad = \quad \begin{cases} \min(K, R_t + D_t) & \text{if } z_t = 1 \\ 0 & \text{Otherwise} \end{cases}$$

Thus, our real decision variable is z_t rather than x_t.

It is easy to find the optimal solution to this problem using classical backward dynamic programming techniques. The algorithms described in figures 1 and 2 were then applied to the problem. The techniques in references such as Bertsekas & Tsitsiklis (1996) and Sutton & Barto (1998) are described exclusively in the context of (stationary) infinite horizon problems, but we feel that the algorithms we have described reasonably capture the spirit of the basic approach represented by forward dynamic programming in the context of time-dependent problems.

Basic forward dynamic techniques, used to estimate a discrete value function, do not capture any structural properties of the problem. It is possible to show (see Papadaki & Powell (2001*b*)) that $V_t(R_t)$ is monotone in R_t (a similar result is given in Puterman (1994) for the case where $K = \infty$). It is not necessarily the case that the approximation $\hat{V}_t^n(R_t)$ will be monotone in R_t, even as $n \rightarrow \infty$. The reason is that we cannot guarantee that each state will be visited infinitely often.

It is possible, however, to apply a forward algorithm, but to then enforce a structural property such as monotonicity. To do this, we first initialize $\hat{V}^0(R_t) = 0$ for $R_t \geq 0$. Next let $\tilde{V}_t^n (R_t^n)$ be the solution to equation (28), giving us an updated estimate of $\hat{V}_t^n(R_t)$ for $R_t = R_t^n$. Normally, we would then update the value of state R_t^n using:

$$\hat{V}_t^{n+1}(R_t) = \begin{cases} (1-\alpha^n)\hat{V}_t^n(R_t^n) + \alpha^n \tilde{V}_t^n (R_t^n) & \text{if } R_t = R_t^n \\ \hat{V}_t^n(R_t) & \text{Otherwise} \end{cases} \tag{49}$$

Of course, there is no guarantee that $\hat{V}_t^{n+1}(R_t)$ will be monotone in R_t after applying equation (49). We can accomplish this using a simple two-stage process. There are several ways we can enforce monotonicity. For illustrative purposes, we use the simplest:

```
If $\hat{V}_t^{n+1}(R_t^n) > \hat{V}_t^{n+1}(R_t^n + 1)$ (right side violation), then:
    Let $r = R_t^n$.
    Do while $\hat{V}_t^{n+1}(r) > \hat{V}_t^{n+1}(r+1)$ and $r < r^{max}$:
        $\hat{V}_t^{n+1}(r+1) = \hat{V}_t^{n+1}(r)$
        $r = r + 1$
If $\hat{V}_t^{n+1}(R_t^n) < \hat{V}_t^{n+1}(R_t^n - 1)$ (left side violation), then:
```

```
Let $r = R^n_t$.
Do while $\hat{V}^{n+1}_t(r) < \hat{V}^{n+1}_t(r-1)$ and $r < r^{max}$:
    $\hat{V}^{n+1}_t(r-1) = \hat{V}^{n+1}_t(r)$
    $r = r - 1$
```

We tested three algorithms for solving this basic aging and replenishment problem: 1) The optimal solution (using backward dynamic programming), 2) forward DP using a double pass algorithm, and 3) forward DP (using a two-pass algorithm) enforcing monotonicity. The algorithms were tested on a battery of 100 problems described in Papadaki & Powell (2001*b*). The aggregate results are shown in table 1, where the approximate algorithms are summarized in terms of the percentage over the optimal objective function. The results indicate that when we enforce monotonicity, the accuracy of the approximation improves dramatically.

Method:	no	no	no	monotone	monotone	monotone
Hold/dispatch	Number of iterations					
cost	(500)	(1000)	(2000)	(500)	(1000)	(2000)
$c^h > c^d/K$	0.329	0.284	0.211	0.076	0.038	0.012
$c^h \simeq c^d/K$	0.143	0.136	0.121	0.058	0.042	0.022
$c^h < c^d/K$	0.054	0.048	0.046	0.035	0.026	0.022
Average	0.176	0.156	0.127	0.056	0.036	0.019

Table 1: (from Papadaki and Powell (2000))

The problem with the basic forward algorithm is that we do not ensure that each state is visited with a nonzero probability. As a result, it is possible to visit a state, obtain (because of statistical error) a high estimate of the cost of being in a state, and then (because the cost of being in the state is so high) never visit the state again.

This set of experiments illustrates the importance of retaining basic structural properties in the approximation of the value function. However, as we pointed out earlier, discrete approximations will never be useful in the solution of vector-valued decision problems since we are not generally going to be able to do a discrete search of the feasible region. Instead, we prefer to use functional approximations of the form given in equations (33) and (34). As described in section 3, we need a sample estimate of the slope of the value function. For this problem class, it is necessary to do this

Method:	linear	linear	linear	linear	nls	nls	nls	nls	DWF-
Hold/dispatch	Number of iterations								TC
cost	(25)	(50)	(100)	(200)	(25)	(50)	(100)	(200)	
$c^h > c^d/K$	0.077	0.060	0.052	0.050	0.227	0.114	0.052	0.025	0.774
$c^h \simeq c^d/K$	0.048	0.033	0.023	0.024	0.068	0.037	0.021	0.014	0.232
$c^h < c^d/K$	0.030	0.022	0.017	0.016	0.030	0.019	0.015	0.014	0.063
Average	0.052	0.038	0.031	0.030	0.108	0.057	0.029	0.017	0.356

Table 2: Fractional error of total cost with respect to the optimal cost (from Papadaki and Powell (2000))

using a finite difference:

$$\tilde{V}_t^n = \tilde{V}_t^n (R_t^n + 1) - \tilde{V}_t^n (R_t^n) \tag{50}$$

Both the linear and nonlinear functional approximations were compared to a myopic, “dispatch when full” heuristic. Under this heuristic, a vehicle was dispatched when it was full, or after it had waited time τ. Under the assumption that τ could be optimized to produce the best results, we ran a series of simulations to choose the best value of τ for *each* dataset. The complete runs are reported in Papadaki & Powell (2001*a*), but are summarized in table 2. All the problems were solved to optimality using classical dynamic programming methods. We first observe that nonlinear produces the best results when a large number of iterations are used to train the functions, but linear works better when there are fewer iterations. This behavior can probably be explained by the need to estimate more parameters (the slopes of individual segments of the function) which produces more statistical error. Overall, however, the results are quite good, and significantly outperform a carefully constructed myopic heuristic. Furthermore, they are better than a standard forward algorithm, even when we use the monotonicity property when estimating the function (note that we used far more iterations when we tested the discrete forward algorithm). This result is unexpected, and suggests that our ability to estimate a simple linear slope is better than our ability to estimate the value of being in a particular state.

These results are quite impressive, but we are solving a problem approximately which can be solved optimally. What is significant about this work is that it can be used to provide high quality solutions to vector-valued aging and replenishment processes. Instead of a state space of 50 or 100, we can use it to solve problems where the state space is 10^{100}.

5.2 Network flow problems

For our next problem, we consider the problem of a multistage dynamic network. For the purposes of this paper, it is sufficient to use a relatively simple network where the time required to move from one node to another is always one time period. We assume that we want to maximize the total contribution generated over the time horizon. With this problem class, we show that a particular piecewise-linear value function retains the network structure of the problem and produces results that are much better than the linear approximation.

Our problem is to determine the flows x_{ijt} of resources between locations $i, j \in \mathcal{I}$ over time subject to three types of randomness: the resources entering the system to be managed, the cost of moving flow, and the upper bounds that restrict the movement of flow from one location to the next. Thus, our exogeneous information process would be $W_t = (\hat{R}_t, c_t, u_t)$.

The optimality equations for this problem are given by:

$$V_t(R_t) \quad = \quad E\left\{ \max_{x_t \in \mathcal{X}_t} c_t x_t + V_{t+1}(R_{t+1}) | R_t \right\} \tag{51}$$

where the feasible region $\mathcal{X}_t = \mathcal{X}_t(R_t, \omega)$, given R_t and ω, is the set of decisions x_t that satisfy:

$$\sum_{j \in \mathcal{I}} x_{ijt} \quad = \quad R_{it} + \hat{R}_{it} \quad \forall i \in \mathcal{I} \tag{52}$$

$$\sum_{i \in \mathcal{I}} x_{ijt} - R_{j,t+1} \quad = \quad 0 \qquad \forall j \in \mathcal{I} \tag{53}$$

$$x_{ijt} \quad \leq \quad u_{ijt} \qquad \forall i, j \in \mathcal{I} \tag{54}$$

$$x_{ijt} \quad \geq \quad 0 \qquad \forall i, j \in \mathcal{I} \tag{55}$$

Following our basic strategy, we write:

$$\tilde{V}_t^{n} (R_t, \omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t} c_t x_t + \hat{V}_{t+1}^n (R_{t+1}) \tag{56}$$

where all the variables on the right hand side of (56) are a function of ω.

If we use a linear value function approximation, we can combine equations (33) with the constraint in (53) and, after a few standard manipulations, show that (56) reduces to:

$$\tilde{V}_t^{n} (R_t, \omega_t) \quad = \quad \max_{x \in \mathcal{X}_t} \sum_{i,j \in \mathcal{I}} (c_{ijt} + \hat{v}_{j,t+1}^n) x_{ijt} \tag{57}$$

This, of course, is a linear network problem which is easily solved to optimality using a range of general or special purpose solvers. Let $\tilde{v}_{it}$ be the dual variable corresponding to the constraint (52). We can then use equation (35) to produce a smoothed estimate of the slope of the value function.

This logic is extremely easy to implement using a single pass algorithm. The double-pass version introduces some complexity since the dual variable obtained from solving equation (57) is a function of $\hat{V}^n_{t+1}$. In the backward pass, it is necessary to recompute the dual as a function of $\hat{V}^{n+1}_{t+1}$. In the context of multistage optimization problems, linear value function approximations always retain the structure of the original problem. For this problem class, a separable nonlinear function also produces a network subproblem. If we use a nonlinear, separable value function approximation, equation (56) becomes:

$$\tilde{V}_t^{n+1}(R_t, \omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t} c_t x_t + \sum_{i \in \mathcal{I}} \hat{V}^n_{i,t+1}(R_{i,t+1}) \tag{58}$$

which is a nonlinear network problem. It is easy to establish that the real value function $V_t(R_t)$, is piecewise linear and concave. For this reason, it is best to use a piecewise linear function as the starting function in the SHAPE algorithm, or to use CAVE to produce a piecewise linear approximation. When $\hat{V}^n_t(R_t)$ is piecewise linear, separable and concave, then equation (58) produces a linear network with integer data (see figure 4). Thus, not only is it solvable with a fast network algorithm, but it also produces integer solutions, which is important in some applications. One special property of network problems is that it is relatively easy to obtain explicit left and right derivatives ($\tilde{v}^+$ and $\tilde{v}^-$). Since each subproblem is a pure network, we can solve two flow augmenting path problems to obtain the desired derivatives (see Powell (1989)). The value function will be piecewise linear concave, but even within a single dataset, the functions may have very different shapes.

When this algorithm is applied to deterministic datasets, the results are near optimal solutions (99.99 percent of optimality or better), with very fast and stable convergence. The CAVE algorithm was also applied to stochastic problems, where it was compared to a classical rolling horizon procedure. The results are shown in table 3, which show that the nonlinear value function approximation significantly outperforms the deterministic rolling horizon procedure. The results become more dramatic as the number of locations increases, reflecting the sensitivity of a deterministic model to an increasing number of decisions.

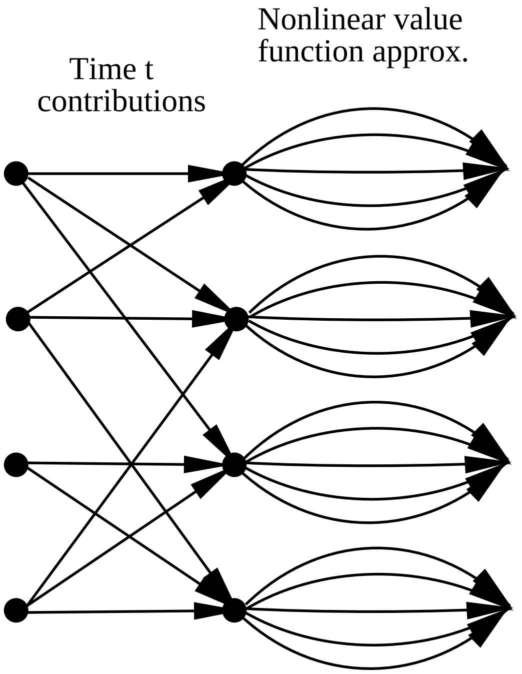


Figure 4: Illustration of one-period network with separable, nonlinear value function approximation

		Percentage of Posterior Bound	
Number of Locations	**Number of Resources**	**Rolling horizon**	**Stochastic using CAVE**
20	100	92.2%	96.3%
20	200	96.3%	97.8%
20	400	96.6%	98.1%
40	100	81.0%	90.5%
40	200	90.7%	96.2%
40	400	92.6%	96.8%
80	100	66.3%	82.1%
80	200	81.4%	93.3%
80	400	84.8%	94.5%

Table 3: Comparison of nonlinear approximation using CAVE to a deterministic rolling horizon procedure, for stochastic problems with different numbers of locations and resources.

5.3 Integer multicommodity flow problems

We next consider the example of a multicommodity flow problem. The techniques and approach are the same as for the single commodity case, but the implications of the dynamic programming approximations for this problem class are quite interesting. First, we show that the use of linear approximations implies that we are solving sequences of network problems, which naturally produce integer solutions. Next, we show that nonlinear approximations produce sequences of integer multicommodity network flow problems where the LP relaxation does *not* guarantee integrality. Thus, this problem is a nice illustration of how different functional approximations can significantly affect the structure of the subproblems that we are solving.

The notation for our problem is basically the same as for the single commodity case, with the exception that we let $k \in \mathcal{K}$ be our set of commodities, and we add a superscript k whenever we need to designate the commodity class. Our algorithmic strategy requires that we solve subproblems of the form:

$$\tilde{V}{}^n_t\,(R_t, \omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t} \sum_{k \in \mathcal{K}} \sum_{i,j \in \mathcal{I}} c^k_{ijt} x^k_{ijt} + \hat{V}^n_{t+1}(R_{t+1}(\omega_t)) \tag{59}$$

where the feasible region $\mathcal{X}_t = \mathcal{X}_t(R_t, \omega_t)$, given R_t and ω_t, is given by:

$$\begin{aligned}
\sum_{j \in \mathcal{I}} x^k_{ijt} &= R^k_{it} + \hat{R}^k_{it} \quad \forall i \in \mathcal{I}, \quad k \in \mathcal{K} && (60)\\
\sum_i x^k_{ijt} - R^k_{j,t+1} &= 0 \qquad\qquad\ \ \forall j \in \mathcal{I}, \quad k \in \mathcal{K} && (61)\\
\sum_{k \in \mathcal{K}} x^k_{ijt} &\leq u_{ijt} \qquad\quad\ \ \forall i, j \in \mathcal{I} && (62)\\
x^k_{ijt} &\geq 0 \qquad\qquad\ \ \forall i, j \in \mathcal{I} \ \ \forall k \in \mathcal{K} && (63)
\end{aligned}$$

A linear value function approximation for this problem class implies that:

$$\hat{V}_t(R_t) \quad = \quad \sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{I}} \hat{v}^k_{it} R^k_{it} \tag{64}$$

Substituting (64) for time $t+1$ into (59) and rearranging terms produces:

$$\tilde{V}{}^n_t\,(R_t, \omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t} \sum_{k \in \mathcal{K}} \sum_{i,j \in \mathcal{I}} \left(c^k_{ijt} + \hat{v}^{k,n}_{j,t+1} \right) x^k_{ijt} \tag{65}$$

Solving (65) subject to (60) - (63) produces a pure network problem which naturally yields integer solutions. We update the estimate of the slope $\tilde{v}^{k,n}_{it}$ using the dual of constraint (60), letting $\hat{v}^{k,n+1}_{it}$ be the resulting smoothed estimate. Thus, a linear value function approximation produces sequences of pure network subproblems, each of which are relatively small (because they involve only one time period). For the hardest problem classes (one example is the management of locomotives for railroads), this can be a valuable feature.

Now consider what happens when we introduce a nonlinear (separable) value function.

$$\tilde{V}_t^{\,n}(R_t, \omega_t) \quad = \quad \max_{x_t \in \mathcal{X}_t} \sum_{k \in \mathcal{K}} \sum_{i,j \in \mathcal{I}} c^k_{ijt} x^k_{ijt} + \sum_{k \in \mathcal{K}} \sum_{j \in \mathcal{I}} \hat{V}^{k,n}_{j,t+1}(R^k_{j,t+1}) \tag{66}$$

The use of a nonlinear value function, which is a function of $R^k_{j,t+1}$, prevents us from decomposing the function by the origin of the flow. As a result, we are now faced with an integer, multicommodity network flow problem. Fortunately, it is relatively small (since it is only one time period), and experimental evidence suggests that the LP relaxation produces integer solutions over 99 percent of the time (when fractional solutions arise, a few branches quickly produces the optimal integer solution). This is not the case when solving deterministic approximations over an extended planning horizon.

Finding and updating nonlinear value functions follows the same basic strategies that we used for the pure network case. However, we can no longer find left and right derivatives by solving flow augmenting path subproblems, as was discussed in the pure network case. Thus, we can use the one-sided SHAPE update (equation (36)), but not the two-sided update in equation (38). We can use the CAVE algorithm, but instead of using left and right derivatives (see equation (37)), we have to use the dual variable for equation (60) in place of $\tilde{v}^+$ and $\tilde{v}^-$.

Table 4 summarizes the extensive experimental results in Topaloglu & Powell (2000). It compares linear and nonlinear approximations to the results obtained using rolling horizon procedures. All the results are expressed as a percentage of the posterior optimal solution. Three sets of results are given. The first varies the number of locations. The second compares the results for four different sets of substitution matrices (these specify the reward received when resource type k is used to serve demand type l. The substitution matrix gives the amount of the task reward that was received when commodity k was used to serve a demand of type l. All the problems assumed the same number of commodities and demand types, and it was always assumed that if $k = l$, then the full reward was received. Finally, runs were conducted with three levels of resources, where 100 resources produced a tightly constrained problem (a substantial portion of the demands could not be covered), 200 was fairly balanced and 400 represented a generous allocation of resources. The results demonstrate that nonlinear approximations work quite well, significantly better than both linear approximations or rolling horizon procedures.

Percent of posterior optimal solution			
Problem	Linear	Nonlinear	Rolling horizon
Results of stochastic runs with varying number of locations			
10 locations	86.14	96.96	93.17
20 locations	78.65	93.28	86.84
40 locations	74.13	92.21	86.89
Results of stochastic runs with varying compatibility patterns			
Sub. matrix I	78.65	93.28	86.84
Sub. matrix II	80.59	95.40	90.87
Sub. matrix III	74.83	91.51	82.66
Sub. matrix IV	84.23	97.12	93.74
Results of stochastic runs with varying numbers of resources			
100 res.	74.19	84.87	76.81
200 res.	78.65	93.28	86.84
400 res.	84.41	96.51	91.67

Table 4: Performance of linear and nonlinear value function approximations against a deterministic rolling horizon procedure

5.4 The dynamic assignment problem

The last problem we consider is the dynamic assignment problem. Here we assume that we have a sequence of arrivals of resources to be assigned to tasks. Each resource and task has a unique set of attributes that determine the contribution from assigning a resource to a task. Resources and tasks arrive randomly over time. At each time period, we may solve an assignment problem involving the resources and tasks that are available at that point in time. Our solution, however, should consider the possibility of assigning a resource to a task that may only arrive in the future (or, holding a task for a resource that may only arrive in the future).

This problem is interesting because all the decisions are discrete. We again consider the properties of a linear value function, but since all the flows are (0,1), nonlinear functions are not particularly useful. Instead, we introduce the use of a particular nonseparable function that retains problem structure.

We describe our problem using the following notation:

$\mathcal{R}_t =$ The set of resources that are available to be assigned to a task at time t that were not assigned in the previous time period.

$\hat{\mathcal{R}}_t =$ The set of resources that first become available at time t.

$\mathcal{R}_t^+ =$ The complete set of resources available to be assigned in time period t.

The set $\mathcal{R}_t$ is the *incomplete* resource vector. Similarly, tasks are defined using:

$\mathcal{L}_t =$ The set of tasks that are available to be assigned to a resource at time t that were not assigned in the previous time period.

$\hat{\mathcal{L}}_t =$ The set of tasks that first become available at time t.

$\mathcal{L}_t^+ =$ The complete set of tasks available to be assigned in time period t.

Thus, $\omega_t = (\hat{\mathcal{R}}_t, \hat{\mathcal{L}}_t)$ represents the information arriving im time period t. Assignment costs and decisions are given by:

$c_{rlt} =$ The cost of assigning resource r to task l at time t.

$$x_{rlt} = \begin{cases} 1 & \text{If resource } r \text{ is assigned to task } l \text{ at time } t \\ 0 & \text{Otherwise} \end{cases}$$

We assume that once a resource and task have been coupled (assigned to each other) they vanish from the system. A resource or task that has not otherwise been assigned is assumed to stay in the system (or held). For this reason, we define:

$\mathcal{R}_t^a =$ The set of resources that are assigned at time t.

$= \{r | \sum_{l \in \mathcal{L}_t} x_{rlt} = 1\}$

$\mathcal{L}_t^a =$ The set of tasks that are assigned at time t.

$= \{l | \sum_{r \in \mathcal{R}_t} x_{rlt} = 1\}$

Our system dynamics can now be expressed using:

$$\begin{aligned} \mathcal{R}_t^+ &= \mathcal{R}_t \bigcup \hat{\mathcal{R}}_t \\ \mathcal{R}_{t+1} &= \mathcal{R}_t^+ \setminus \mathcal{R}_t^a & (67) \\ \mathcal{L}_t^+ &= \mathcal{L}_t \bigcup \hat{\mathcal{L}}_t \\ \mathcal{L}_{t+1} &= \mathcal{L}_t^+ \setminus \mathcal{L}_t^a & (68) \end{aligned}$$

We can solve this problem using the same basic equations given in (28), but as before, the challenge is finding an appropriate form for $\hat{V}$. We are particularly interested in integer solutions, so it seems reasonable to find a class of approximations that retains the basic structure of solving sequences of assignment problems. Once again, we find that a linear value function approximation accomplishes this, meaning that we have to solve sequences of problems of the form:

$$\begin{aligned}
\tilde{V}{}_t^n(\mathcal{R}_t,\mathcal{L}_t,\omega_t) &= \max_{x_t} \sum_{r\in\mathcal{R}_t^+(\omega)} \sum_{l\in\mathcal{L}_t^+(\omega)} c_{rlt}x_{rlt} + \sum_{r\in\mathcal{R}_t^+(\omega)} \hat{v}_{r,t+1}(1-\sum_{l\in\mathcal{L}_t^+(\omega)} x_{rlt}) \\
&\quad + \sum_{l\in\mathcal{L}_t^+(\omega)} \hat{v}_{l,t+1}(1-\sum_{r\in\mathcal{R}_t^+(\omega)} x_{rlt}) \qquad (69)
\end{aligned}$$

subject to:

$$\sum_{r\in\mathcal{R}_t^+(\omega)} x_{rlt} \leq 1 \quad \forall l \in \mathcal{L}_t^+(\omega) \qquad (70)$$

$$\sum_{l\in\mathcal{L}_t^+(\omega)} x_{rlt} \leq 1 \quad \forall r \in \mathcal{R}_t^+(\omega) \qquad (71)$$

The linear value function approximation used in equation (69) assumes that there is a value $v_{r,t+1}$ if a resource r is not assigned in period t, and is therefore "added" to the pool of resources in period $t+1$. Similarly, $v_{l,t+1}$ captures the value of holding a task until the next time period. It is straightforward to show that (69) is equivalent to solving:

$$\tilde{V}{}_t^n(\mathcal{R}_t,\mathcal{L}_t,\omega_t) = \max_{x_t} \sum_{r\in\mathcal{R}_t^+} \sum_{l\in\mathcal{L}_t^+} (c_{rlt} - v_{r,t+1} - v_{l,t+1})x_{rlt} + K \qquad (72)$$

where K is a constant term that is not a function of x_t. Solving (72) subject to equations (70)-(71) is, of course, a simple assignment problem. We can estimate $v_{r,t}$ and $v_{l,t}$ by using the dual variables of (70) and (71) and smoothing them as we did earlier.

This simple linear approximation does, however, introduce a potentially significant source of error. It assumes that the value of coupling a resource and a task is equal to the value of the resource plus the value of the task. It is not hard to create problems where this is not a very good approximation. A particular resource and task may have a very high value (negative cost) of being coupled to each other that is much higher than the value of the resource and the value of the task individually. We can capture this property by using a nonseparable value function approximation. This means replacing (72) with:

$$\tilde{V}{}_t^n(\mathcal{R}_t,\mathcal{L}_t,\omega_t) = \max_{x_t} \sum_{r\in\mathcal{R}_t^+} \sum_{l\in\mathcal{L}_t^+} (c_{rlt} - v_{rl,t+1})x_{rlt} + K \qquad (73)$$

Type of	Myopic	Resource	Resource and Task	Arc
experiment		Gradients	Gradients	Gradients
Deterministic	88.4	93.4	97.5	99.8
Stochastic	86.6	89.2	92.8	92.0

Table 5: Results of value function approximations for deterministic and stochastic experiments expressed as a percent of the posterior optimal solution.

As we see, this particular approximation also gives us a sequence of assignment problems, thereby retaining the essential problem structure that we are interested in.

Spivey & Powell (2000) compared four versions of the algorithm: myopic ($\hat{v}_{r,t} = 0, r \in \mathcal{R}_t^+, \hat{v}_{l,t} = 0, l \in \mathcal{L}_t^+$), resource gradients only ($\hat{v}_{l,t} = 0, l \in \mathcal{L}_t^+$), resource and task gradients (as in equation (69)), and arc gradients (as in equation (73)). The algorithms were applied to 20 different problems, for both deterministic and stochastic datasets. The results are summarized in table 5. The deterministic datasets show progressively better results as the value function approximation becomes more accurate. The runs with arc gradients almost always produced optimal solutions. For the stochastic datasets, the best results were obtained when we used both resource and task gradients; the arc gradients slightly underperformed these results. In all cases, the dynamic programming approximations of all forms produced results that were noticeably better than the myopic solutions.

6 Summary

This paper has shown how we can apply forward dynamic programming principles to large-scale, stochastic optimization problems, bringing closer together the fields of dynamic programming and multistage stochastic programming. The techniques are illustrated in the context of a broad class of resource management problems, where actions at time t impact the future through their impact on right hand side constraints. We use this structure to suggest specific linear and nonlinear approximations, and discuss the implications of these particular functional approximations on our ability to solve different classes of problems. Early numerical work on these problem classes indicate that this approach can work quite well, and offers a powerful new approach for solving problems in this class.

Our approach bridges neuro-dynamic programming techniques and multistage linear programming (although not all of our problems are multistage linear programs). Although a general comparison of the techniques was provided in section 4 it is useful to reinforce these differences, given the material covered in section 5. These include:

a) The use of the incomplete state variable, which allows us to use equation (28) overcomes the problem of having to even approximate the outcome space. With a suitable choice of functional approximation, our problems are typically no more difficult to solve than a greedy heuristic (myopic approximation).

b) The functional approximations are updated in equation (29) using gradient information (either from duals or finite differences). This logic exploits the structure of the problem in a way that is not captured using the estimation logic implicit in equation (42) or in stochastic programming (either scenario methods or nested Benders).

c) Resource allocation problems typically have an underlying polyhedral structure which we exploit. Neuro-dynamic programming is a more general technique, but does not exploit this structure.

d) In neuro-dynamic programming, the basis functions $\phi_k(s)$ are fixed, while the coefficients ρ_k must be estimated. In our work, the functional approximations might be piecewise linear with no preassumed shape, aside from basic properties such as concavity.

e) Where appropriate, we enforce structure such as monotonicity or concavity. Using basis functions, this would have to be enforced through a combination of an appropriate choice of function as well as imposing constraints on the coefficients.

f) Neuro-dynamic programming (Bertsekas & Tsitsiklis (1996)) and reinforcement learning (Sutton & Barto (1998)) are both presented in the context of stationary problems. Our approach is particularly well-suited to the types of nonstationary problems (in particular, finite horizon applications) which arise in resource management. The technique can also be applied in an approximate way to problems with non-Markovian structure.

Acknowledgement

This research was supported in part by grant AFOSR-F49620-93-1-0098 from the Air Force Office of Scientific Research. The authors would also like to acknowledge the comments and suggestions of Erhan Cinlar.

References

Adelman, D. & Nemhauser, G. (1999), 'Price-directed control of remnant inventory systems', *Operations Research* **47**(6), 889–898.

Archibald, T. W., Buchanan, C. S., McKinnon, K. I. M. & Thomas, L. C. (1999), 'Nested benders decomposition and dynamic programming for reservoir optimisation', *J. Oper. Res. Soc.* **50**, 468–479.

Beale, E., Forest, J. & Taylor, C. (1980), Multi-time period stochastic programming, *in* M. Dempster, ed., 'Stochastic Programming', Academic Press, pp. 387–402.

Bean, J., Birge, J. & Smith, R. (1987), 'Aggregation in dynamic programming', *Operations Research* **35**, 215–220.

Bertazzi, L., Bertsekas, D. & Speranza, M. G. (2000), Optimal and neuro-dynamic programming solutions for a stochastic inventory trasportation problem, Unpublished technical report, Universita Degli Studi Di Brescia.

Bertsekas, D. & Tsitsiklis, J. (1996), *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA.

Bes, C. & Sethi, S. (1988), 'Concepts of forecast and decision horizons: Applications to dynamic stochastic optimization problems', *Mathematics of Operations Research* **13**(2), 295–310.

Bhaskaran, S. & Sethi, S. (1987), 'Decision and forecast horizons in a stochastic environment : A survey', *Optimal Control Applications and Methods* **8**(1), 61–67.

Birge, J. (1985), 'Decomposition and partitioning techniques for multistage stochastic linear programs', *Operations Research* **33**(5), 989–1007.

Birge, J. & Louveaux, F. (1997), *Introduction to Stochastic Programming*, Springer-Verlag, New York.

Chen, Z.-L. & Powell, W. (1999*a*), 'A convergent cutting-plane and partial-sampling algorithm for multistage linear programs with recourse', *Journal of Optimization Theory and Applications* **103**(3), 497–524.

Chen, Z.-L. & Powell, W. B. (1999*b*), 'A convergent cutting-plane and partial-sampling algorithm for multistage stochastic linear programs with recourse', *Journal of Optimization Theory and Applications* **102**(3), 497–524.

Cheung, R. K.-M. & Powell, W. B. (2000), 'SHAPE: A stochastic hybrid approximation procedure for two-stage stochastic programs', *Operations Research* **48**(1), 73–79.

Consigli, G. & Dempster, M. A. H. (1998), 'Dynamic stochastic programming for asset-liability management', *Annals of Operations Research* **81**, 131–161.

Culioli, J.-C. & Cohen, G. (1990), 'Decomposition/coordination algorithms in stochastic optimization', *SIAM Journal of Control and Optimization* **28**, 1372–1403.

Gendreau, M., Guertin, F., Potvin, J. & Taillard, E. (1999), 'Parallel tabu search for real-time vehicle routing and dispatching', *Transportation Science* **33**, 381–390.

Glockner, G. D. & Nemhauser, G. L. (2000), 'A dynamic network flow problem with uncertain arc capacities: Formulation and problem structure', *Operations Research* **48**, 233–242.

Godfrey, G. A. & Powell, W. B. (1999), An adaptive, dynamic programming algorithm for integer, stochastic, dynamic resource allocation problems, Technical Report SOR-97-19, Department of Operations Research and Financial Engineering, Princeton University.

Godfrey, G. A. & Powell, W. B. (2001), 'An adaptive, distribution-free approximation for the newsvendor problem with censored demands, with applications to inventory and distribution problems', *Management Science.* (to appear).

Hall, L., Schulz, A., Shmoys, D. & Wein, L. (1997), 'Scheduling to minimize average completion time: Off-line and on-line approximation algorithms', *Mathematics of Operations Research* **22**, 513–544.

Higle, J. & Sen, S. (1991), 'Stochastic decomposition: An algorithm for two stage linear programs with recourse', *Mathematics of Operations Research* **16**(3), 650–669.

Hoogeveen, J. & Vestjens, A. (2000), 'A best possible deterministic on-line algorithm for minimizing delivery time on a single machine', *SIAM Journal on Discrete Mathematics* **13**, 56–63.

Infanger, G. (1994), *Planning under Uncertainty: Solving Large-scale Stochastic Linear Programs*, The Scientific Press Series, Boyd & Fraser, New York.

Infanger, G. & Morton, D. (1996), 'Cut sharing for multistage stochastic linear programs with interstate dependency', *Mathematical Programming* **75**, 241–256.

Kall, P. & Wallace, S. (1994), *Stochastic Programming*, John Wiley and Sons, New York.

Laporte, G. & Louveaux, F. (1993), 'The integer l-shaped method for stochastic integer programs wtih complete recourse', *Operations Research Letters* **13**(3), 133–142.

Louveaux, F. & van der Vlerk, M. (1993), 'Stochastic programming with simple integer recourse', *Mathematical Programming* **61**, 301–325.

Lundin, R. A. & Morton, T. E. (1975), 'Planning horizons for the dynamic lot size model: Zabel vs. protective procedures and computational results', *Operations Research* **23**(4), 711–734.

Mulvey, J. & Vladimirou, H. (1991*a*), 'Solving multistage stochastic networks: An application of scenario aggregation', *Networks* **21**, 619–643.

Mulvey, J. M. & Ruszczynski, A. J. (1995), 'A new scenario decomposition method for large-scale stochastic optimization', *Operations Research* **43**(3), 477–490.

Mulvey, J. M. & Vladimirou, H. (1991*b*), 'Solving multistage stochastic networks: An application of scenario aggregation', *Networks* **21**, 619–643.

Mulvey, J. M. & Vladimirou, H. (1992), 'Stochastic network programming for financial planning problems', *Management Science* **38**(8), 1642–1664.

Papadaki, K. & Powell, W. B. (2001*a*), Exploiting structure in adaptive dynamic programming algorithms for a stochastic batch service problem, CL-01-02, Princeton University.

Papadaki, K. & Powell, W. B. (2001*b*), A scalable adaptive dynamic programming problem for a batch production process, Report CL-01-01, Princeton University.

Pereira, M. & Pinto, L. (1991), 'Multistage stochastic optimization applied to energy planning', *Mathematical Programming* **52**, 359–375.

Powell, W. B. (1989), 'A review of sensitivity results for linear networks and a new approximation to reduce the effects of degeneracy', *Transportation Science* **23**(4), 231–243.

Powell, W. B., Shapiro, J. A. & Simao, H. P. (2000), An adaptive dynamic programming algorithm for the heterogeneous resource allocation problem, Technical Report CL-00-06, Department of Operations Research and Financial Engineering, Princeton University.

Puterman, M. L. (1994), *Markov Decision Processes*, John Wiley and Sons, Inc., New York.

Ruszczynski, A. (1993), 'Parallel decomposition of multistage stochastic programming problems', *Math. Programming* **58**(2), 201–228.

Sen, S. & Higle, J. (1999), 'An introductory tutorial on stochastic linear programming models', *Interfaces* **29**(2), 33–61.

Spivey, M. Z. & Powell, W. B. (2000), The dynamic assignment problem, Technical Report CL-00-03, Department of Operations Research and Financial Engineering, Princeton University.

Sutton, R. & Barto, A. (1998), *Reinforcement Learning*, The MIT Press, Cambridge, Massachusetts.

Topaloglu, H. & Powell, W. B. (2000), Dynamic programming approximations for time-staged integer multicommodity flow problems, Technical Report CL-00-02, Department of Operations Research and Financial Engineering, Princeton University.

Tsitsiklis, J. & Van Roy, B. (1997), 'An analysis of temporal-difference learning with function approximation', *IEEE Transactions on Automatic Control* **42**, 674–690.

Van Slyke, R. & Wets, R. (1969), 'L-shaped linear programs with applications to optimal control and stochastic programming', *SIAM Journal of Applied Mathematics* **17**(4), 638–663.

Linguistic Communication Between Components of DHP?

George Lendaris and Thaddeus Shannon
(lendaris@sysc.pdx.edu)

Abstract

Our context is the DHP version of Adaptive Critics, a Reinforcement Learning method that implements Approximate Dynamic Programming to design a control system that is (approximately) optimal with respect to criteria stated in a Utility (reward) function. Key components of the DHP system are: 1) Plant to be controlled, 2) Controller (Action) device, 3) Critic (Value function estimator), 4) Utility function, and 5) Plant Jacobian calculator. Neural Network (NN) function approximators are often used for the Controller and Critic components, and also for the Plant Jacobian calculator. We have (successfully) demonstrated use of Fuzzy function approximators for these three components, in various combinations, with NN function approximators used for the remaining ones.

The motivation for employing Fuzzy rather than NN components has been the ease of capturing *a priori* knowledge available to the human designer for the starting condition of the components, and, depending on the Fuzzy architecture used, ease of subsequent training. So far, however, we have always de-fuzzified the output of the Fuzzy components into crisp values for communication to other components of the DHP system. Such use of Fuzzy in the DHP context is of continuing research interest, however, in looking to the future at the Workshop, we will propose exploring the notion that the inter-component communication itself be expressed in a Fuzzy (linguistic), rather than a crisp, manner.

In the presentation, we will start with a description of the Fuzzy work accomplished to date for *intra*-component processing, and then focus on the potential use of linguistic variables for *inter*- component communication. This is not yet a fully developed proposal. At the Workshop, we will put early ideas we have developed on the table for interactive dialog, and let the “creative juices” flow. In the following, we briefly sketch the basic ideas so far developed.

To assist in developing our mental models, we have so far used the context of a human learning from a coach/mentor to enhance performance, be it in athletics, the performing arts, or art itself. A bulk of the coach’s contribution to the process is accomplished via linguistic communication.

Part of the motivation for this approach is based on an example described by Prof. Zadeh in some of his talks, which is roughly as follows: There are two experienced drivers in the same vehicle; one at the steering wheel is blindfolded; passenger uses his/her eyes (and any other useful sensory sub/system) and gives verbal instructions to the “driver”; the driver successfully performs the required tasks. In such a situation, the passenger performs the “intelligent act” of distilling (compressing) visual sensory data into linguistic commands. The person at the wheel performs the intelligent act of extracting relevant *information* from the verbal commands, and then taking corresponding actions. For an autonomous

vehicle, the equivalent of both intelligent acts would have to be accomplished via computing devices. One way to explore implementing such an autonomous vehicle will be to focus on each of the two intelligent tasks separately.

The above may be stylized as follows:

> Sensory Input =>/Intelligent Actor1/ =>Linguistic Command =>/Intelligent Actor2/ =>Control Command.

From the above example, it is easy to appreciate that linguistic communication has the attribute of being parsimonious. In the DHP context, endowing the Critic with an appropriate linguistic vocabulary would allow it to distill information about what could be a complex situation into a small message unit that gives qualitative information about the task. Simultaneously, of course, the Controller would have to be endowed with the ability to understand/interpret the same vocabulary.

The driver example above might give the impression that the amount of information needed for the communication between the passenger and driver is "small," as demonstrated by sufficiency of the verbal channel. However, care must be taken to assess how much "shared knowledge" the two actors have, implied in the stipulation that both drivers are experienced. What if one or both of the actors are novices in their respective roles? What are associated considerations related to development of internal linguistic world models by/for each actor? How will this translate to equivalent considerations in the DHP context -- in particular, relative to the degree of compression possible via the verbal communication?

To help us build up ideas, consider a human Reinforcement Learning context involving, say, a pole-vault athlete and a coach.

Athlete: Sensor data => Athlete => Physical action
Coach: Sensor data (different?) => Linguistic Instruction/Command
Athlete: "Listens" to Coach's linguistic feedback and then adjusts physical action(s).
[Does Athlete's interpretation of Coach's linguistic communication occur in some portion of the "sensory section" of Athlete's processing?] [Note, there is no "low level" feedback to Athlete about such things as timing, quantity of muscle contraction, etc.]

What assumptions do we develop about prior stored information in Athlete and/or Coach? E.g., what are implicit "system" requirements for Coach to be able to give to Athlete an instruction such as "think high", where Coach's intention is for Athlete to manifest a modified sequence of motor controls yielding an overall motion pattern that is "higher"?

Consider that there is *learning* on part of both, Coach and Athlete. The Coach learns how to get Athlete to develop desired results. How does Coach figure out how to use linguistic terms to evoke desired kinesthetic experience/expression? How does Athlete learn to interpret Coach's verbal instructions? How are kinesthetic experiences (muscle patterns, etc.) accessible via linguistics? There seems to be an axiomatic necessity for Sensory Inputs PLUS a Linguistic Model of the world, and perhaps an ability to Reason Linguistically. There are no doubt internal process models in both actors (both complex adaptive systems), and the *dynamics of the two systems are likely different*. Such ideas/questions will have their counterparts in the DHP context.

We consider that the training process may be indirect. For example, Coach may have Athlete do something to assist Athlete develop his/her internal model, rather than to be the end goal, per se. E.g., to have Athlete learn to have body upside down. Would Coach be better at this if Coach had done the task before personally (=> “shared” model aspect)?

For beginning athlete, there is issue of basic skill acquisition, such as internal model development and physical development (e.g., flexibility, musculature, etc. -- lower nervous system). For example, in the context of trapeze, if Athlete has prior experience as a gymnast, Couch could use higher level linguistic terms to instruct Athlete (both to teach new/refined concepts, and/or to evoke desired response). In the case where no shared linguistic model exists between Coach/Athlete, Coach is obliged to use lower level language, and perhaps other sensory modalities to establish shared higher-level linguistic-based models. E.g., Coach could demonstrate desired action for Athlete, to provide a visual model for Athlete, which Coach can subsequently access via linguistic communication. Another alternative is to video tape Athlete’s performance. Coach could then view tape with Athlete, and provide a linguistic model during tape viewing via verbally “labeling” observed motion patterns. What will equivalent considerations be in the DHP context?

While teaching a student driver, the teacher is obliged to use a different semantic level than for the experienced driver in the Zadeh example cited at the beginning. To design a controller using the DHP process, we will have to make substantial progress on the *learning*-related issues suggested above. The feeling here is that linguistic feedback may be even better (more parsimonious) for learning than for (just) performing. In the early stages of Athlete’s learning, the shared vocabulary may be small, with attendant constraints on Coach’s communications to Athlete. As shared vocabulary expands, “level” of communication (e.g., amount of compression) may rise. Coach is also learning in this process, and the associated “inner dialogue” may use a substantially different vocabulary from that used in communication to Athlete.

The Adaptive Critic process entails two learning loops (for the Controller/Action component, and for the Critic component). In the HDP version of ACs, a model is required only in the Controller training loop; the ADDHP version only uses a model in the Critic train loop; and the DHP version (the one to be pursued by us) requires a model in both loops. A key new consideration for DHP suggested by the above mental exercises relates to the use of a model for both training loops. In the past, the *same* model has been used for both training loops. Above considerations suggest use of *different* models in the two loops. Thus, we will be expanding the DHP method in two fundamental ways: 1) use of linguistic *inter*-component communication, and 2) use of *different* linguistic models for Critic and Controller training loops. Both aspects suggest exciting challenges for future research.

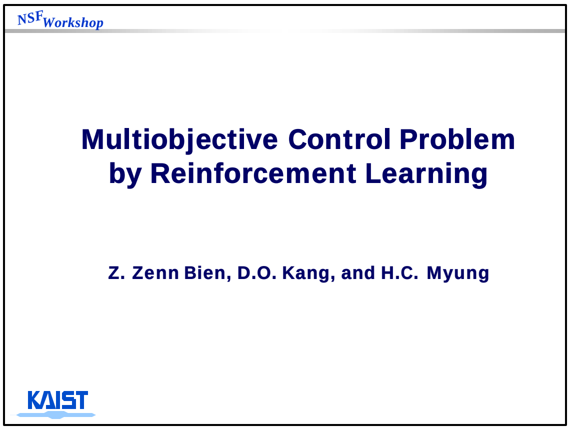
Multiobjective control problem by Reinforcement Learning

Z.Zenn Bien, D.O.Kang and H.C.Myung

(zbien@ee.kaist.ac.kr)

Abstract

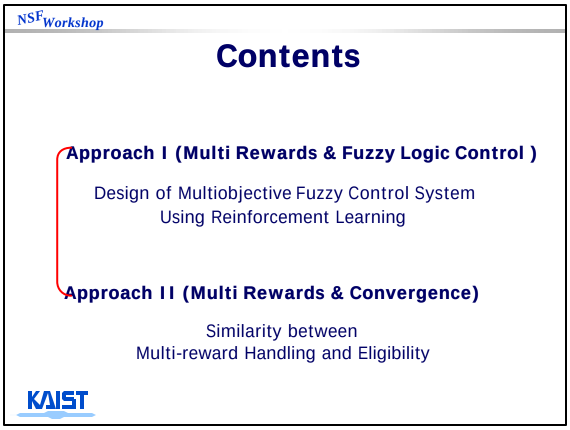
The problem of multiple-objective control for a plant with no a priori knowledge is studied in the framework of reinforcement learning. Two approaches are considered :
(1)Adoption of temporal difference learning in which a fuzzy inference subsystem is used for updating the evaluation function of the control system.
(2)Use of the objective eligibility notion to handle the problem of multi-rewards and convergence. And then the results are discussed in light of ADP.



NSF Workshop

Multiobjective Control Problem by Reinforcement Learning

Z. Zenn Bien, D.O. Kang, and H.C. Myung



NSF Workshop

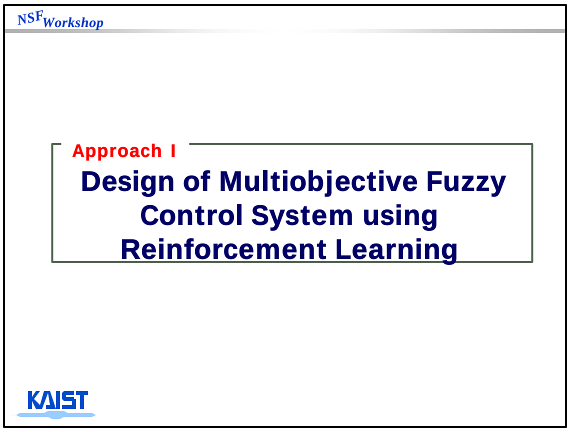
Contents

Approach I (Multi Rewards & Fuzzy Logic Control)

Design of Multiobjective Fuzzy Control System
Using Reinforcement Learning

Approach II (Multi Rewards & Convergence)

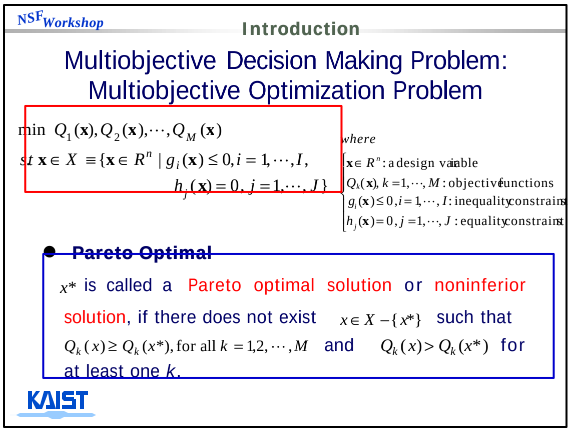
Similarity between
Multi-reward Handling and Eligibility



NSF Workshop

Approach I

Design of Multiobjective Fuzzy Control System using Reinforcement Learning



NSF Workshop

Introduction

Multiobjective Decision Making Problem: Multiobjective Optimization Problem

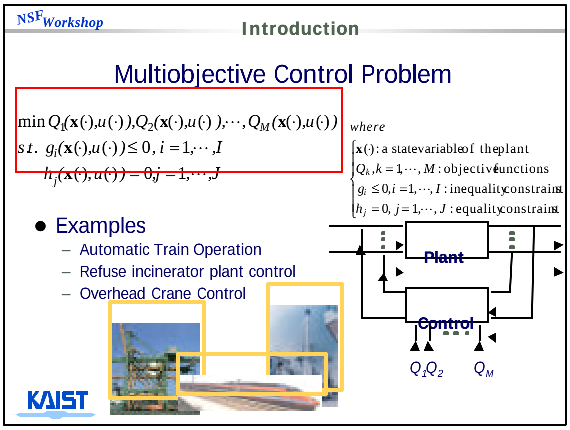
$$\min\ Q_1(\mathbf{x}), Q_2(\mathbf{x}), \cdots, Q_M(\mathbf{x})$$
$$s.t\ \mathbf{x} \in X \equiv \{\mathbf{x} \in R^n \mid g_i(\mathbf{x}) \le 0, i = 1, \cdots, I,\ h_j(\mathbf{x}) = 0,\ j = 1, \cdots, J\}$$

where

$$\begin{cases} \mathbf{x} \in R^n : \text{a design variable} \\ Q_k(\mathbf{x}),\ k = 1, \cdots, M : \text{objective functions} \\ g_i(\mathbf{x}) \le 0, i = 1, \cdots, I : \text{inequality constraints} \\ h_j(\mathbf{x}) = 0, j = 1, \cdots, J : \text{equality constraints} \end{cases}$$

- **Pareto Optimal**

x^* is called a Pareto optimal solution or noninferior solution, if there does not exist $x \in X - \{x^*\}$ such that $Q_k(x) \ge Q_k(x^*)$, for all $k = 1, 2, \cdots, M$ and $Q_k(x) > Q_k(x^*)$ for at least one k.



NSF Workshop

Introduction

Multiobjective Control Problem

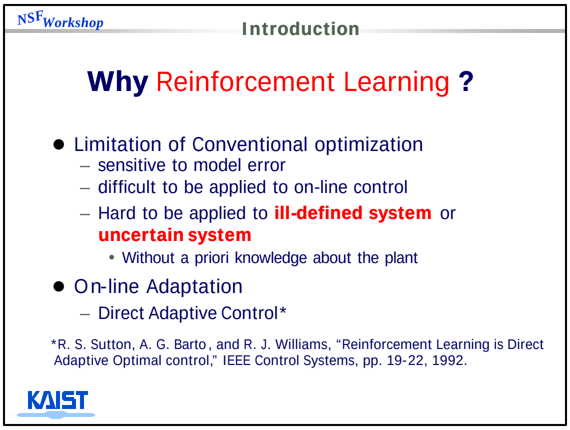
$$\min Q_1(\mathbf{x}(\cdot), u(\cdot)), Q_2(\mathbf{x}(\cdot), u(\cdot)), \cdots, Q_M(\mathbf{x}(\cdot), u(\cdot))$$
$$s.t.\ g_i(\mathbf{x}(\cdot), u(\cdot)) \le 0,\ i = 1, \cdots, I$$
$$h_j(\mathbf{x}(\cdot), u(\cdot)) = 0, j = 1, \cdots, J$$

where

$$\begin{cases} \mathbf{x}(\cdot) : \text{a state variable of the plant} \\ Q_k, k = 1, \cdots, M : \text{objective functions} \\ g_i \le 0, i = 1, \cdots, I : \text{inequality constraints} \\ h_j = 0,\ j = 1, \cdots, J : \text{equality constraints} \end{cases}$$

- Examples
  - Automatic Train Operation
  - Refuse incinerator plant control
  - Overhead Crane Control

Plant
Control
$Q_1 Q_2$
Q_M



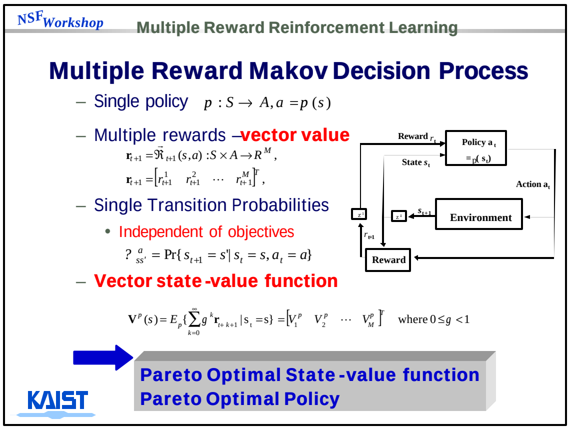
NSF Workshop

Introduction

Why Reinforcement Learning ?

- Limitation of Conventional optimization
  - sensitive to model error
  - difficult to be applied to on-line control
  - Hard to be applied to **ill-defined system** or **uncertain system**
    - Without a priori knowledge about the plant
- On-line Adaptation
  - Direct Adaptive Control*

*R. S. Sutton, A. G. Barto, and R. J. Williams, “Reinforcement Learning is Direct Adaptive Optimal control,” IEEE Control Systems, pp. 19-22, 1992.



NSF Workshop — Multiple Reward Reinforcement Learning

Multiple Reward Makov Decision Process

– Single policy $p : S \to A, a = p(s)$

– Multiple rewards –**vector value**

$\mathbf{r}_{t+1} = \vec{\Re}_{t+1}(s,a) : S \times A \to R^M ,$

$\mathbf{r}_{t+1} = \left[r_{t+1}^1 \quad r_{t+1}^2 \quad \cdots \quad r_{t+1}^M \right]^T ,$

– Single Transition Probabilities

- Independent of objectives

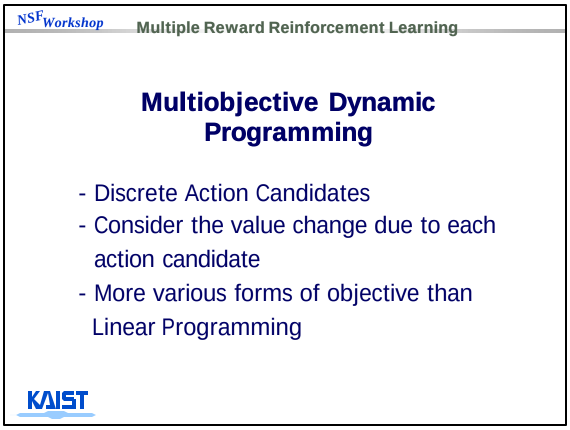
$? _{ss'}^{a} = \Pr\{ s_{t+1} = s' | s_t = s, a_t = a \}$

– **Vector state-value function**

$$\mathbf{V}^p(s) = E_p\{\sum_{k=0}^{\infty} g^k \mathbf{r}_{t+k+1} | s_t = s\} = \left[V_1^p \quad V_2^p \quad \cdots \quad V_M^p \right]^T \quad \text{where } 0 \le g < 1$$

Pareto Optimal State-value function
Pareto Optimal Policy

KAIST

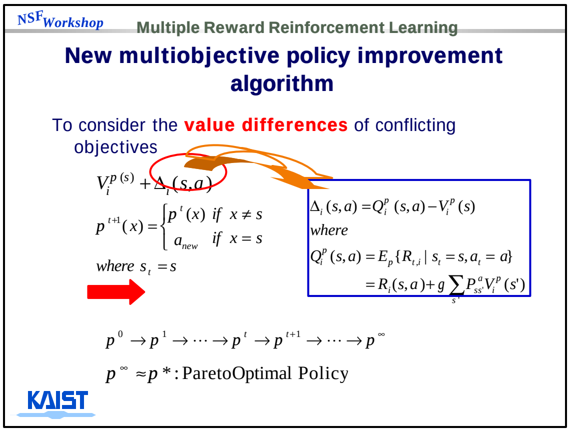


NSF Workshop — Multiple Reward Reinforcement Learning

Multiobjective Dynamic Programming

- Discrete Action Candidates
- Consider the value change due to each action candidate
- More various forms of objective than Linear Programming

KAIST



NSF Workshop — Multiple Reward Reinforcement Learning

New multiobjective policy improvement algorithm

To consider the **value differences** of conflicting objectives

$$V_i^{p(s)} + \Delta_i(s,a)$$

$$p^{t+1}(x) = \begin{cases} p^t(x) & if \ x \ne s \\ a_{new} & if \ x = s \end{cases}$$

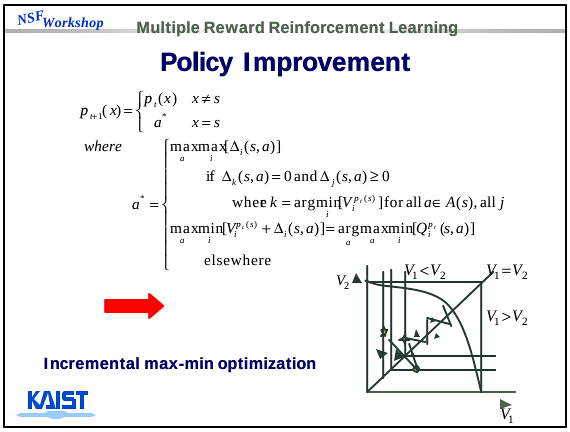
where $s_t = s$

$$\Delta_i(s,a) = Q_i^p(s,a) - V_i^p(s)$$
where
$$Q_i^p(s,a) = E_p\{R_{t,i} | s_t = s, a_t = a\}$$
$$= R_i(s,a) + g \sum_{s'} P_{ss'}^a V_i^p(s')$$

$$p^0 \to p^1 \to \cdots \to p^t \to p^{t+1} \to \cdots \to p^\infty$$

$$p^\infty \approx p^* : \text{ParetoOptimal Policy}$$

KAIST



NSF Workshop — Multiple Reward Reinforcement Learning

Policy Improvement

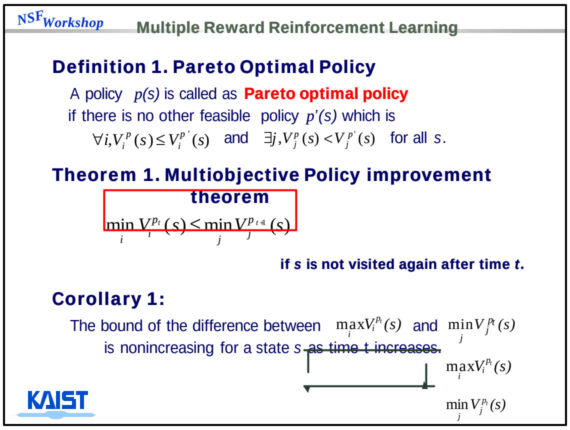
$$p_{t+1}(x) = \begin{cases} p_t(x) & x \ne s \\ a^* & x = s \end{cases}$$

where

$$a^* = \begin{cases} \max_a \max_i [\Delta_i(s,a)] \\ \quad \text{if } \Delta_k(s,a) = 0 \text{ and } \Delta_j(s,a) \ge 0 \\ \quad\quad \text{where } k = \arg\min_i [V_i^{p_t(s)}] \text{ for all } a \in A(s), \text{ all } j \\ \max_a \min_i [V_i^{p_t(s)} + \Delta_i(s,a)] = \arg\max_a \max_a \min_i [Q_i^{p_t}(s,a)] \\ \quad \text{elsewhere} \end{cases}$$

Incremental max-min optimization

KAIST



NSF Workshop — Multiple Reward Reinforcement Learning

Definition 1. Pareto Optimal Policy

A policy *p(s)* is called as **Pareto optimal policy** if there is no other feasible policy *p'(s)* which is

$\forall i, V_i^p(s) \le V_i^{p'}(s)$ and $\exists j, V_j^p(s) < V_j^{p'}(s)$ for all *s*.

Theorem 1. Multiobjective Policy improvement theorem

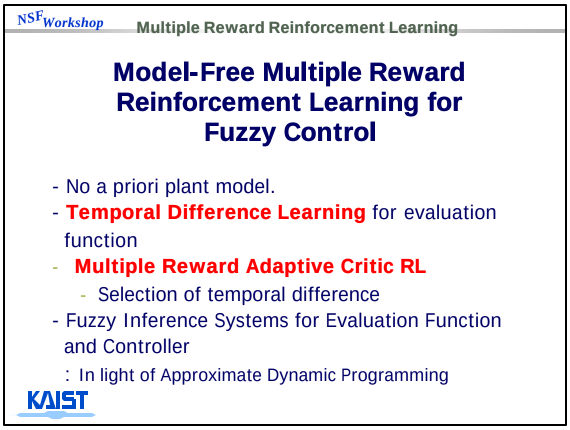
$$\min_i V_i^{p_t}(s) \le \min_j V_j^{p_{t+1}}(s)$$

if *s* is not visited again after time *t*.

Corollary 1:

The bound of the difference between $\max_i V_i^{p_t}(s)$ and $\min_j V_j^{p_t}(s)$ is nonincreasing for a state *s* ~~as time t increases.~~

KAIST



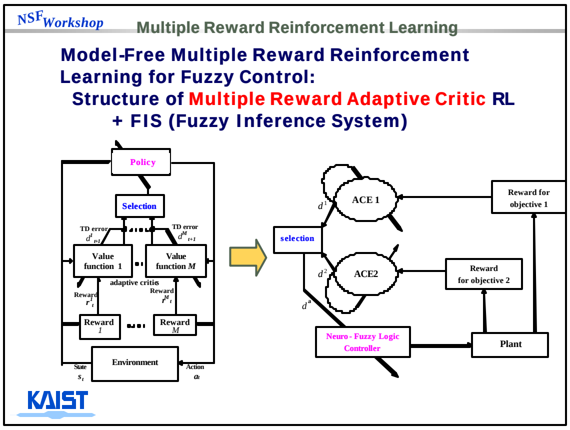
NSF Workshop — Multiple Reward Reinforcement Learning

Model-Free Multiple Reward Reinforcement Learning for Fuzzy Control

- No a priori plant model.
- **Temporal Difference Learning** for evaluation function
- **Multiple Reward Adaptive Critic RL**
  - Selection of temporal difference
- Fuzzy Inference Systems for Evaluation Function and Controller

: In light of Approximate Dynamic Programming

KAIST



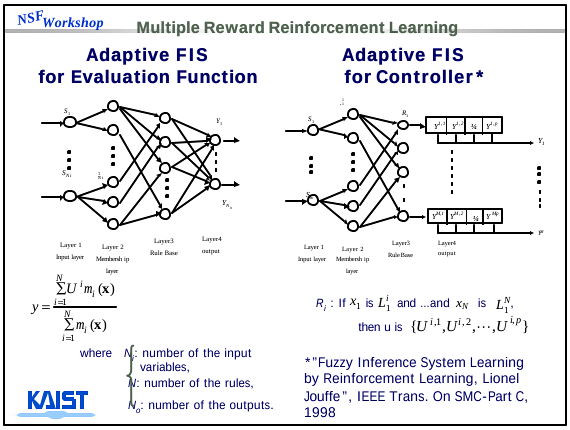
NSF Workshop

Multiple Reward Reinforcement Learning

Model-Free Multiple Reward Reinforcement Learning for Fuzzy Control:

Structure of Multiple Reward Adaptive Critic RL + FIS (Fuzzy Inference System)

KAIST



NSF Workshop

Multiple Reward Reinforcement Learning

Adaptive FIS for Evaluation Function

Adaptive FIS for Controller*

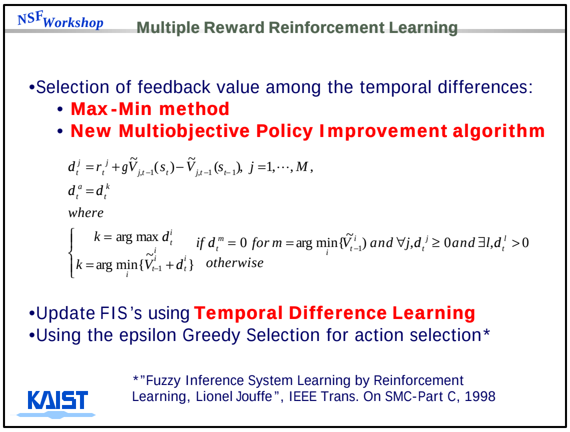
$$y=\frac{\sum_{i=1}^{N} U^i m_i(\mathbf{x})}{\sum_{i=1}^{N} m_i(\mathbf{x})}$$

where N_i: number of the input variables,
N: number of the rules,
N_o: number of the outputs.

R_i : If x_1 is L_1^i and ...and x_N is L_1^N,
then u is $\{U^{i,1}, U^{i,2}, \cdots, U^{i,p}\}$

*"Fuzzy Inference System Learning by Reinforcement Learning, Lionel Jouffe", IEEE Trans. On SMC-Part C, 1998

KAIST



NSF Workshop

Multiple Reward Reinforcement Learning

- •Selection of feedback value among the temporal differences:
  - • Max-Min method
  - • New Multiobjective Policy Improvement algorithm

$$d_t^j = r_t^j + g\tilde{V}_{j,t-1}(s_t) - \tilde{V}_{j,t-1}(s_{t-1}),\ j = 1,\cdots,M,$$

$$d_t^a = d_t^k$$

where

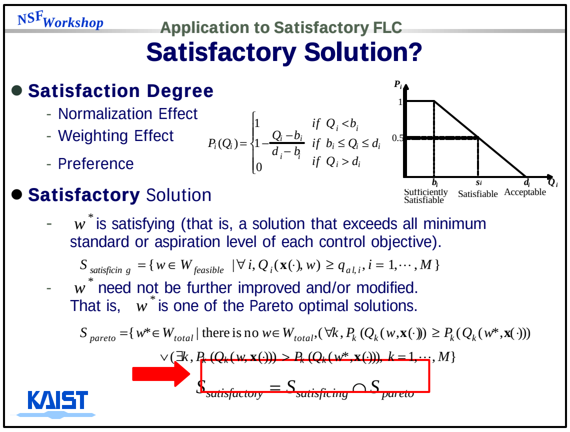
$$\begin{cases} k = \arg\max_i d_t^i & if\ d_t^m = 0\ for\ m = \arg\min_i(\tilde{V}_{t-1}^i)\ and\ \forall j, d_t^j \geq 0\, and\, \exists l, d_t^l > 0 \\ k = \arg\min_i\{\tilde{V}_{t-1}^i + d_t^i\} & otherwise \end{cases}$$

•Update FIS's using Temporal Difference Learning

•Using the epsilon Greedy Selection for action selection*

*"Fuzzy Inference System Learning by Reinforcement Learning, Lionel Jouffe", IEEE Trans. On SMC-Part C, 1998

KAIST



NSF Workshop

Application to Satisfactory FLC

Satisfactory Solution?

- ● Satisfaction Degree
  - Normalization Effect
  - Weighting Effect
  - Preference

$$P_i(Q_i) = \begin{cases} 1 & if\ Q_i < b_i \\ 1 - \dfrac{Q_i - b_i}{d_i - b_i} & if\ b_i \leq Q_i \leq d_i \\ 0 & if\ Q_i > d_i \end{cases}$$

- ● Satisfactory Solution
  - w^* is satisfying (that is, a solution that exceeds all minimum standard or aspiration level of each control objective).

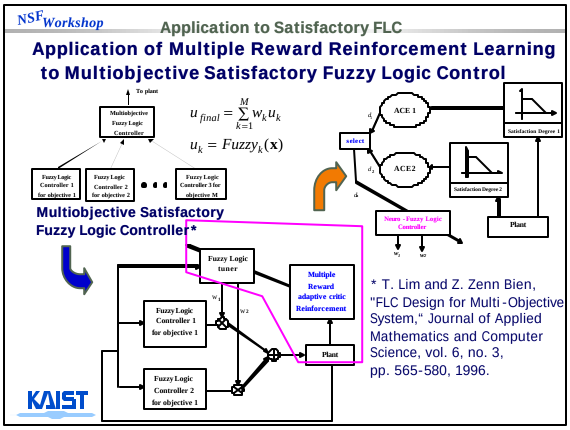
$$S_{satisficin\ g} = \{w \in W_{feasible}\ | \forall i, Q_i(\mathbf{x}(\cdot), w) \geq q_{al,i}, i = 1,\cdots,M\}$$

  - w^* need not be further improved and/or modified. That is, w^* is one of the Pareto optimal solutions.

$$S_{pareto} = \{w^* \in W_{total}\,|\,\text{there is no } w \in W_{total}, (\forall k, P_k(Q_k(w,\mathbf{x}(\cdot))) \geq P_k(Q_k(w^*,\mathbf{x}(\cdot)))$$
$$\vee(\exists k, P_k(Q_k(w,\mathbf{x}(\cdot))) > P_k(Q_k(w^*,\mathbf{x}(\cdot))), k = 1,\cdots,M\}$$

$$\boxed{S_{satisfactory} = S_{satisficing} \cap S_{pareto}}$$

KAIST



NSF Workshop

Application to Satisfactory FLC

Application of Multiple Reward Reinforcement Learning to Multiobjective Satisfactory Fuzzy Logic Control

To plant
Multiobjective Fuzzy Logic Controller
Fuzzy Logic Controller 1 for objective 1
Fuzzy Logic Controller 2 for objective 2
Fuzzy Logic Controller 3 for objective M

$$u_{final} = \sum_{k=1}^{M} w_k u_k$$

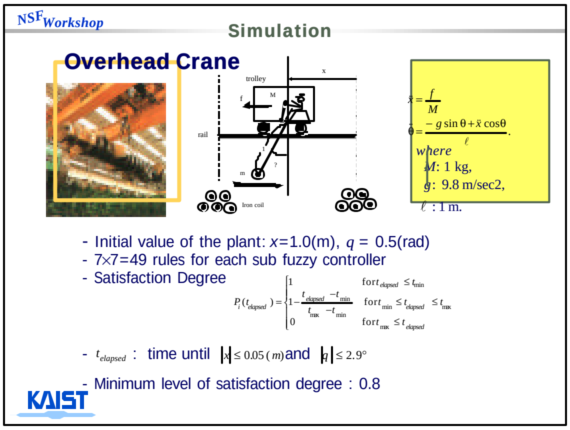
$$u_k = Fuzzy_k(\mathbf{x})$$

Multiobjective Satisfactory Fuzzy Logic Controller*

ACE 1
ACE2
d_1
d_2
d_a
select
Satisfaction Degree 1
Satisfaction Degree 2
Neuro - Fuzzy Logic Controller
Plant
w_1
w_2
Fuzzy Logic tuner
Multiple Reward adaptive critic Reinforcement
Fuzzy Logic Controller 1 for objective 1
Fuzzy Logic Controller 2 for objective 1
w_1
w_2
Plant

* T. Lim and Z. Zenn Bien, "FLC Design for Multi-Objective System," Journal of Applied Mathematics and Computer Science, vol. 6, no. 3, pp. 565-580, 1996.

KAIST



NSF Workshop

Simulation

Overhead Crane

$$\ddot{x} = \frac{f}{M}$$

$$\ddot{\theta} = \frac{-g\sin\theta + \ddot{x}\cos\theta}{\ell}.$$

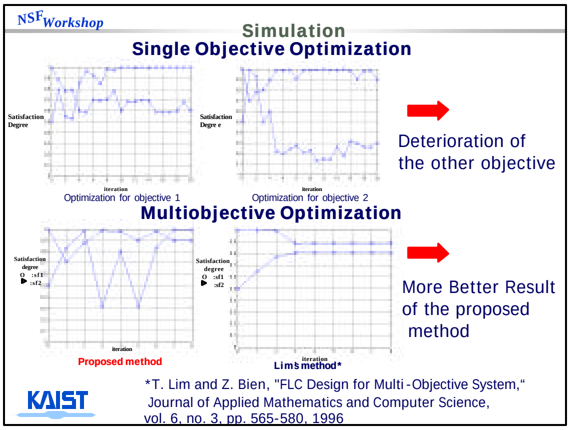
where
M: 1 kg,
g: 9.8 m/sec2,
ℓ : 1 m.

- Initial value of the plant: x=1.0(m), q = 0.5(rad)
- 7×7=49 rules for each sub fuzzy controller
- Satisfaction Degree

$$P_i(t_{elapsed}) = \begin{cases} 1 & for\, t_{elapsed} \leq t_{\min} \\ 1 - \dfrac{t_{elapsed} - t_{\min}}{t_{\max} - t_{\min}} & for\, t_{\min} \leq t_{elapsed} \leq t_{\max} \\ 0 & for\, t_{\max} \leq t_{elapsed} \end{cases}$$

- $t_{elapsed}$: time until $|x| \leq 0.05\,(m)$ and $|q| \leq 2.9°$
- Minimum level of satisfaction degree : 0.8

KAIST



NSF Workshop

Simulation

Single Objective Optimization

Optimization for objective 1

Optimization for objective 2

Deterioration of the other objective

Multiobjective Optimization

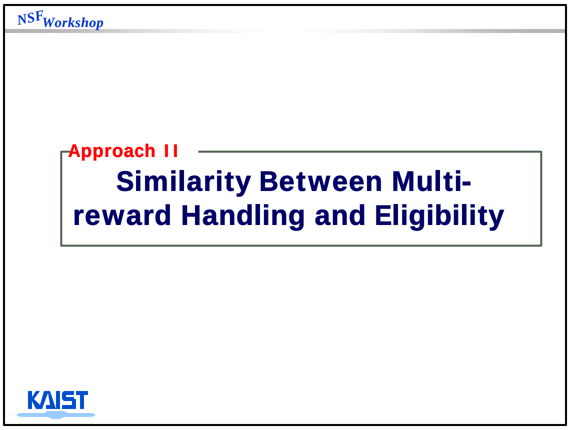
Proposed method

Lim's method*

More Better Result of the proposed method

*T. Lim and Z. Bien, "FLC Design for Multi-Objective System," Journal of Applied Mathematics and Computer Science, vol. 6, no. 3, pp. 565-580, 1996

KAIST

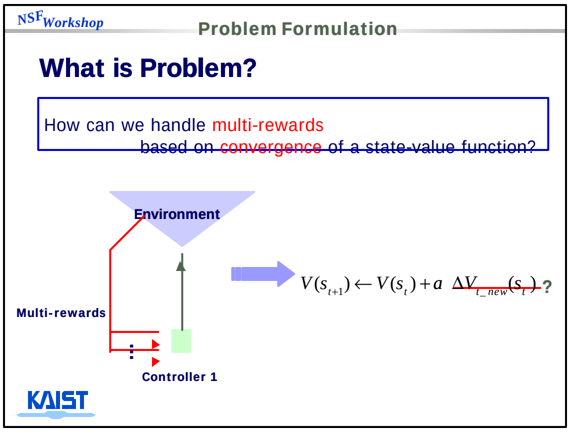


NSF Workshop

Approach II

Similarity Between Multi-reward Handling and Eligibility

KAIST



NSF Workshop

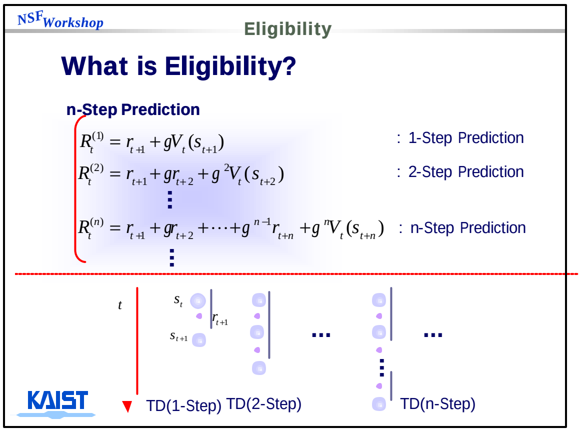
Problem Formulation

What is Problem?

How can we handle multi-rewards based on convergence of a state-value function?

$$V(s_{t+1}) \leftarrow V(s_t) + a\ \Delta V_{t\_new}(s_t)\ ?$$

KAIST



NSF Workshop

Eligibility

What is Eligibility?

n-Step Prediction

$$R_t^{(1)} = r_{t+1} + gV_t(s_{t+1})$$: 1-Step Prediction

$$R_t^{(2)} = r_{t+1} + gr_{t+2} + g^2V_t(s_{t+2})$$: 2-Step Prediction

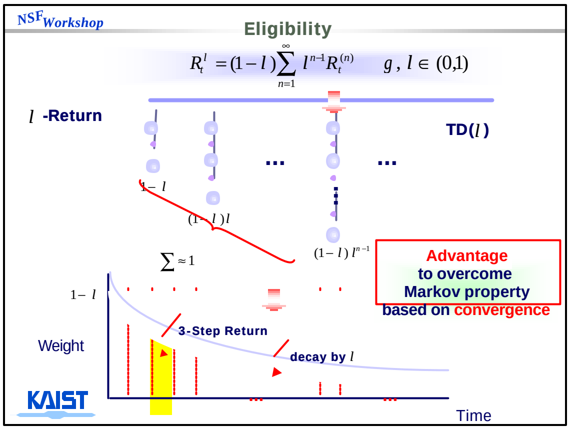
$$\vdots$$

$$R_t^{(n)} = r_{t+1} + gr_{t+2} + \cdots + g^{n-1}r_{t+n} + g^nV_t(s_{t+n})$$: n-Step Prediction

$$\vdots$$

TD(1-Step) TD(2-Step) TD(n-Step)

KAIST



NSF Workshop

Eligibility

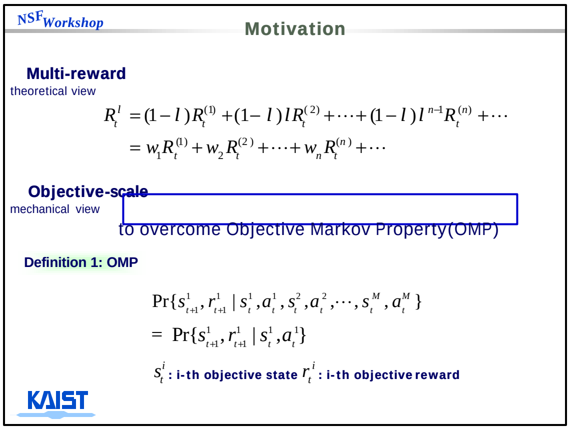
$$R_t^l = (1-l)\sum_{n=1}^{\infty} l^{n-1}R_t^{(n)} \qquad g,\ l \in (0,1)$$

l -Return

TD(l)

Advantage to overcome Markov property based on convergence

KAIST



NSF Workshop

Motivation

Multi-reward
theoretical view

$$R_t^l = (1-l)R_t^{(1)} + (1-l)lR_t^{(2)} + \cdots + (1-l)l^{n-1}R_t^{(n)} + \cdots$$
$$= w_1R_t^{(1)} + w_2R_t^{(2)} + \cdots + w_nR_t^{(n)} + \cdots$$

Objective-scale
mechanical view

to overcome Objective Markov Property(OMP)

Definition 1: OMP

$$\Pr\{s_{t+1}^1, r_{t+1}^1 \mid s_t^1, a_t^1, s_t^2, a_t^2, \cdots, s_t^M, a_t^M\}$$
$$= \Pr\{s_{t+1}^1, r_{t+1}^1 \mid s_t^1, a_t^1\}$$

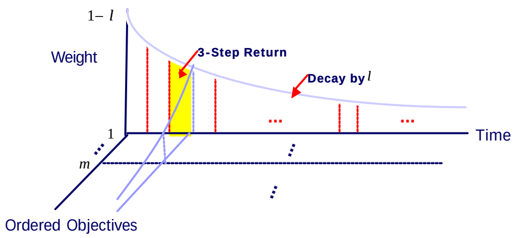
s_t^i : i-th objective state r_t^i : i-th objective reward

KAIST

Motivation

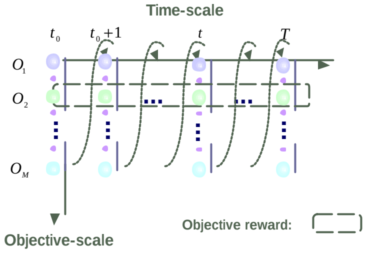
Definition 2: Objective Ordering

Ordering: **Objective Number ∝ 1 / Importance Degree**



Motivation

Combination
Time-scale and Objective-scale



TD Learning Method

Objective Prediction

$$\begin{cases} L_t^{(i)}(s_t^{(i+1)}) = \sum_{k=1}^{i} g^{k-1} r_{t+1}^{(i)} + g V_t^{(i+1)}(s_t^{(i+1)}) & 1 \le i < M \\ L_t^{(M)}(s_{t+1}^{(1)}) = \sum_{k=1}^{M} g^{k-1} r_{t+1}^{(i)} + g V_t^{(1)}(s_{t+1}^{(1)}) & i = M \end{cases}$$

$$\begin{cases} V_t^{(1)}(s_t^{(1)}) = r_{t+1}^{(1)} + \cdots + g^{M-1} r_{t+1}^{(M)} + g^{M} r_{t+2}^{(1)} + \cdots \\ \qquad = E_p\{\sum_{l=0}^{\infty} \sum_{k=1}^{i} g^{Ml+k-1} r_{t+l+1}^{(k)} \mid s_t^{(1)} = s^{(1)}\} \\ V_t^{(i)}(s_t^{(i)}) = r_{t+1}^{(i)} + \cdots + g^{M-i} r_{t+1}^{(M)} + g^{M-i+1} r_{t+2}^{(1)} + \cdots \end{cases}$$

$$= [V_t^{(1)}(s_t^{(1)}) - E_p\{\sum_{k=1}^{i-1} g^{k-1} r_{t+l+1}^{(k)} \mid s_t^{(1)} = s^{(1)}\}] / g^{i-1}$$

TD Learning Method

$$w = 1 / \sum_{i=1}^{M} (1-l) l^{i-1}$$

$$\Delta V_t(s_t^{(1)}) = w \sum_{k=1}^{M+1} (gl)^{k-1} d_k$$

Following Eligibility Strategy based on proof of convergence

$$\begin{cases} = w[(gl)^0\{r_t^{(1)} + gV_t^{(2)}(s_t^{(2)}) - V_t^{(1)}(s_t^{(1)}) / w\} \\ + (gl)^1\{r_t^{(2)} + gV_t^{(3)}(s_t^{(3)}) - V_t^{(2)}(s_t^{(2)})\} \\ \vdots \\ + (gl)^{M-1}\{r_t^{(M)} + gV_t^{(1)}(s_{t+1}^{(1)}) - V_t^{(M)}(s_t^{(M)})\} \\ - (gl)^M V_t^{(1)}(s_{t+1}^{(1)})] \end{cases}$$

mathematically provable for convergence

$$\begin{aligned} &= -V_t^{(1)}(s_t^{(1)}) \\ &\quad + w[(gl)^0\{r_t^{(1)} + gV_t^{(2)}(s_t^{(2)}) - glV_t^{(2)}(s_t^{(2)})\} \\ &\quad \vdots \\ &\quad + (gl)^{M-1}\{r_t^{(M)} + gV_t^{(1)}(s_{t+1}^{(1)}) - glV_t^{(1)}(s_{t+1}^{(1)})\}] \end{aligned}$$

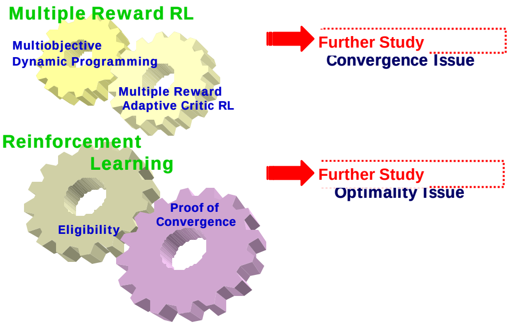
TD Learning Method

$$\begin{aligned} &= -V_t^{(1)}(s_t^{(1)}) + w[(1-l)\{r_t^{(1)} + gV_t^{(2)}(s_t^{(2)})\} \\ &\quad + (1-l)l\ \{r_t^{(1)} + gr_t^{(2)} + g^2 V_t^{(3)}(s_t^{(3)})\} \\ &\quad \vdots \\ &\quad + (1-l)\, l^{M-1}\{r_t^{(1)} + \cdots + g^{M-1} r_t^{(M)} + g^M V_t^{(1)}(s_{t+1}^{(1)})\}] \\ &= -V_t^{(1)}(s_t^{(1)}) + w \sum_{i=1}^{M} (1-l) l^{i-1} L_t^{(i)} \end{aligned}$$

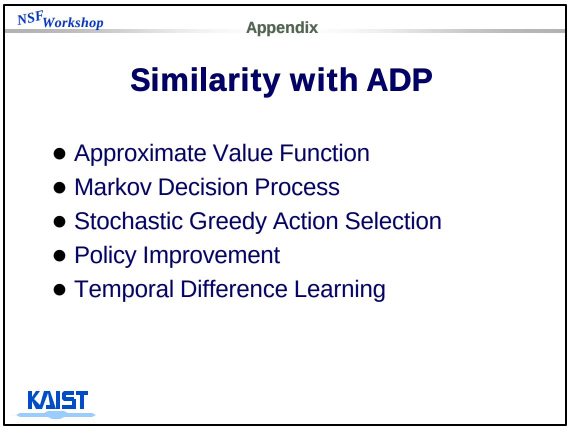
$$\Delta V_t^{(i)}(s_t^{(i)}) = \sum_{k=i}^{M} g^{k-1} r_t^{(k)} + \sum_{k=1}^{i-1} g^{M+k-1} r_{t+1}^{(k)} + g^{i} V_t^{(i)}(s_{t+1}^{(i)}) - V_t^{(i)}(s_t^{(i)}) \quad 2 \le i \le M$$

$$d_i = \begin{cases} r_t^{(i)} + gV_t^{(i+1)}(s_t^{(i+1)}) - V_t^{(i)}(s_t^{(i)}), & \text{for k} = 1, \\ r_t^{(k)} + gV_t^{(k+1)}(s_t^{(k+1)}) - V_t^{(k)}(s_t^{(k)}), & \text{for } 1 < \text{k} < \text{M}, \\ r_t^{(k)} + gV_t^{(1)}(s_{t+1}^{(1)}) - V_t^{(M)}(s_t^{(M)}), & \text{for k} = \text{M}, \\ -V_t^{(1)}(s_{t+1}^{(1)}), & \text{for k} = \text{M}+1, \end{cases}$$

Summary and Further Study



Multiobjective Control Problem

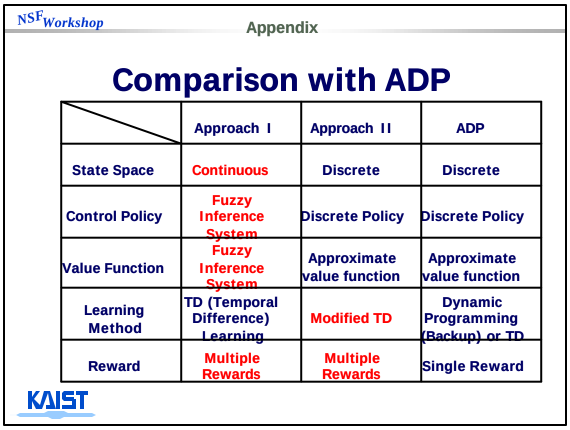


NSF Workshop Appendix

Similarity with ADP

- Approximate Value Function
- Markov Decision Process
- Stochastic Greedy Action Selection
- Policy Improvement
- Temporal Difference Learning

KAIST



NSF Workshop Appendix

Comparison with ADP

	Approach I	Approach II	ADP
State Space	Continuous	Discrete	Discrete
Control Policy	Fuzzy Inference System	Discrete Policy	Discrete Policy
Value Function	Fuzzy Inference System	Approximate value function	Approximate value function
Learning Method	TD (Temporal Difference) Learning	Modified TD	Dynamic Programming (Backup) or TD
Reward	Multiple Rewards	Multiple Rewards	Single Reward

KAIST

Performance Potential-Based On-line Optimization Of Markov Processes

Xi-Ren Cao

Department of Electrical and Electronic Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Tel: (852) 2358-7048 Fax: (852) 2358-1485
Email: eecao@ee.ust.hk

Abstract

Perturbation analysis provides performance derivatives based on a sample path of a discrete event system. The early work started with queueing networks. Recent study indicates that similar concepts can be applied to Markov processes; this leads to the area of performance potential-based on-line optimization of Markov processes, which provides a new vision from the sensitivity point of view to Markov decision processes (MDPs) and related topics in reinforcement learning (RL). The potentials can be estimated on a sample path, hence the potential-based optimization can be implemented on-line for real systems. The potential-based optimization includes the following three topics:

1. Gradient-based on-line optimization. Potentials are estimated to calculate the performance gradients with respect to continuous parameters, and stochastic approximation methods are used to improve the speed of convergence.
2. Potential-based on-line policy iteration. The estimated potentials are used to implement policy iteration. We emphasize that this approach does not require to know or to estimate the entire transition matrix (only the system parameters related to control actions are needed), and in many cases it can be implemented without knowing the system parameters. This approach also has some computational advantages, e.g., only the potentials at controlled-transition-related states need to be estimated. In addition, the system does not need to visit all the state-action pairs.
3. Gradient-based on-line policy iteration: this may be applied to Markov processes where the actions are correlated at different states, e.g., queueing systems with phase-type distributions where the phases are not observable. Many features have to be explored and an example will be given to illustrate the ideas.

In addition, we will discuss the relation among PA, MDPs and RL: Policy iteration can be easily derived and viewed as a result of performance sensitivity to discrete parameters; RL methods can be applied to obtain performance gradients. Performance potentials provide a natural link among these different research areas. The main concepts can be explained with two simple equations.

From Perturbation Analysis to Markov Decision Processes and Reinforcement Learning: An Extended Summary

Xi-Ren Cao*
Department of Electrical and Electronic Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

April 2002

Perturbation analysis (PA) [3, 7, 11, 13] provides performance sensitivities for a discrete event dynamic system (DEDS) by analyzing the dynamical behavior of a single sample path of the DEDS. Performance optimization can be achieved by combining PA with stochastic approximation methods. Markov decision process (MDP) [1, 16, 17, 18] is a general model for performance optimization of DEDSs. Policy iteration, the basic approach in MDPs, can be implemented based on sample paths. The goal of reinforcement learning (RL) [1, 2, 19, 20] is to learn how to make decisions to improve a system's performance by observing its behavior. In this paper, we study the relations among these closely related fields. We show that MDP solutions can be derived naturally from performance sensitivity analysis provided by PA, and that reinforcement learning, TD(λ), neuro-dynamic programming, etc, are sample-path-based efficient ways of estimating the performance potentials and related quantities, which are crucial elements in PA, MDPs and other optimization approaches. This sensitivity-based view of MDPs leads to a new approach, the gradient-based policy iteration, to MDPs with correlated actions at different states.

We first briefly review the basic principles for PA and introduce the fundamental concept of *perturbation realization*. We show that the effect of a (small or large) perturbation in a

*Supported in part by a grant from Hong Kong UGC. Tel: (852) 2358-7048 Fax: (852) 2358-1485 Email: eecao@ee.ust.hk

DEDS on its long-run performance can be measured by a quantity called *realization factor*, and that the effect of a small change in a system parameter on the system performance equals the sum of the effects of all the perturbations induced by this small change in the parameter. From this basic principle of PA, the performance sensitivities can be obtained [3, 7].

In particular, we consider an M-server closed Jackson (Gordon-Newell) network in which service times are exponentially distributed with means $\bar{s}_i$, $i = 1, 2, ...M$, respectively, where M is the number of servers in the network. The system state is $\mathbf{n} = (n_1, ..., n_M)$, with n_l being the number of customers at server l. Let $\mathbf{N}(t)$ denote a sample path ($\mathbf{N}(t)$ is the system state at time t). The system performance is defined as the long-run average

$$\eta^{(f)} = \lim_{L\to\infty} \frac{1}{L} \int_0^{T_L} f(\mathbf{N}(t))dt = \lim_{L\to\infty} \frac{F_L}{L}, \qquad F_L = \int_0^{T_L} f(\mathbf{N}(t))dt,$$

where f is a performance mapping from the state space to $\mathcal{R}$, and T_L is the Lth transition time of the system.

We first study the average effect of a small perturbation on F_L. Suppose that at time $t = 0$ server i obtains a small perturbation Δ, which means that the service completion time of the customer being served in server i at time t will be delayed by a small amount Δ. This perturbation will affect the service completion time of other customers in the system according to some simple rules (called *perturbation propagation rules* in literature [13]). Thus, the sample path with this perturbation will differ from $\mathbf{N}(t)$, and we call it a perturbed sample path and denote it as $\mathbf{N}'(t)$. The realization factor of a perturbation Δ of server i in state $\mathbf{n}$ is defined as

$$c^{(f)}(\mathbf{n}, i) = \lim_{L\to\infty} E\left[\frac{1}{\Delta}\left(\int_0^{T'_L} f(\mathbf{N}'(t))dt - \int_0^{T_L} f(\mathbf{N}(t))dt\right)\right]. \tag{1}$$

where the prime " ′ " represents variables on the perturbed path. It can be shown that [3]

$$\frac{\bar{s}_i}{\eta^{(I)}} \frac{\partial \eta^{(f)}}{\partial \bar{s}_i} = \sum_{all\ \mathbf{n}} p(\mathbf{n}) c^{(f)}(\mathbf{n}, i), \tag{2}$$

where

$$\eta^{(I)} = \lim_{L\to\infty} \frac{T_L}{L} = \lim_{L\to\infty} \frac{1}{L} \int_0^{T_L} I(\mathbf{n})dt,$$

in which $I(\mathbf{n}) = 1$ for all $\mathbf{n}$, and $p(\mathbf{n})$ is the steady state probability. Thus, the *normalized* performance sensitivity (the left-hand side of (2)) equals the steady-state expectation of the realization factor. In fact, realization factor measures the effect of a small perturbation, and the weighted sum on the righthand of (2) reflects the sum of effects of all the perturbations induced by the change in a mean service time. Very efficient algorithms can be developed to estimate the normalized sensitivity directly based on a single sample path [14].

The above basic principle for PA can be applied to Markov processes to obtain performance sensitivity [7]. Two types of sensitivities are studied: sensitivities with respect to discrete and continuous parameters. In the discrete case, the difference of the performance of the Markov processes under two different policies is found. In the continuous case, the policies are randomized, with a parameter δ indicating the probability of applying one policy and $1-\delta$ the probability of the other, the derivative of the performance with respect to δ is found; this is the derivative along the direction from one policy to the other in the policy space. With the discrete sensitivity formula, the standard policy iteration optimization algorithm can be very easily derived. Compared with the continuous sensitivity formula, it is clear that at each step the policy iteration algorithm simply chooses the "steepest" direction (with the largest absolute value of the directional derivative) to go for its next policy [4].

Specifically, we consider an irreducible and aperiodic Markov chain $\mathbf{X} = \{X_n : n \geq 0\}$ on a finite state space $\mathcal{S} = \{1, 2, \cdots, M\}$ with transition probability matrix $P = [p(i,j)] \in [0,1]^{M\times M}$. Let $\pi = (\pi_1, \ldots, \pi_M)$ be the vector representing its steady-state probabilities, and $f = (f_1, f_2, \cdots, f_M)^T$ be the performance vector, where "T" represents transpose. We have $Pe = e$, where $e = (1, 1, \cdots, 1)^T$ is an M-dimensional vector whose all components equal 1, and $\pi = \pi P$. The performance measure is the long-run average defined as

$$\eta = E_\pi(f) = \sum_{i=1}^{M} \pi_i f_i = \pi f = \lim_{L\to\infty} \frac{1}{L} \sum_{l=0}^{L-1} f(X_l) = \lim_{L\to\infty} \frac{F_L}{L}, \tag{3}$$

where

$$F_L = \sum_{l=0}^{L-1} f(X_l).$$

Let P' be another irreducible transition probability matrix on the same state space. Suppose P changes to $P(\delta) = P + \delta Q = \delta P' + (1-\delta)P$, with $\delta > 0$, $Q = P' - P = [q(i,j)]$. We have $Qe = 0$. The performance measure will change to $\eta(\delta) = \eta + \Delta\eta$. Denote $\eta' = \eta(1)$

be the performance corresponding to P'. The derivative of η in the direction of Q is defined as $\frac{d\eta}{d\delta} = \lim_{\delta\to 0}\frac{\Delta\eta}{\delta}$. The discrete sensitivity is $\eta' - \eta$.

In this system, a perturbation means that the system is perturbed from one state i to another state j. For example, consider the case where $q(k,i) = -\frac{1}{2}$, $q(k,j) = \frac{1}{2}$, and $q(k,l) = 0$ for all $l \neq i,j$. Suppose that in the original sample path the system is in state k and jumps to state i, then in the perturbed path it may jump to state j instead. Thus, we study two independent Markov chains $\mathbf{X} = \{X_n; n \geq 0\}$ and $\mathbf{X}' = \{X'_n; n \geq 0\}$ with $X_0 = i$ and $X'_0 = j$; both of them have the same transition matrix P. The *realization factor* is defined as [7]:

$$d(i,j) = \lim_{L\to\infty} E\left[\sum_{l=0}^{L-1}(\, f(X'_l) - f(X_l)\,)\,\middle|\, X_0 = i,\ X'_0 = j\right], \quad i,j = 1,\ldots,M\,. \tag{4}$$

Thus, $d(i,j)$ represents the average effect of a jump from i to j on F_L in (3) (cf. (1)).

The matrix $D \in \mathcal{R}^{M\times M}$, with $d(i,j)$ as its (i,j)th element, is called a *realization matrix*. From (4), we can prove that D satisfies the Lyapunov equation [7]

$$D - PDP^T = F, \tag{5}$$

where $F = fe^T - ef^T$. Similar to (2), we can prove

$$\frac{d\eta}{d\delta} = \pi Q D^T \pi^T. \tag{6}$$

Next, from (4), we can define $d(i,j) = g(i) - g(j)$ and $D = ge^T - eg^T$. g is called a *performance potential vector* with $g(i)$ the potential at state i. From (5), it is easy to prove that g satisfies the Poisson equation

$$(I - P + e\pi)g = f.$$

From (5), it is easy to derive that

$$\frac{d\eta}{d\delta} = \pi Q g, \tag{7}$$

this is the continuous sensitivity formula; and

$$\eta' - \eta = \pi' Q g, \tag{8}$$

this is the discrete sensitivity formula. Policy iteration procedure is a natural consequence of this formula: if P is the current policy, then the actions with the largest absolute value of Qg (componentwisely) are chosen as the policy for the next iteration. The diference between the continuous sensitivity (7) and the discrete one (8) is only that π is replaced by π'. This clearly shows that at each step the policy iteration algorithm simply chooses the "steepest" direction to go for its next policy [4].

The concept of *performance potentials* plays a significant role in sensitivity analysis of Markov processes and MDPs. Just as the potential energy in physics, only the relative values (i.e., the differences) of the performance potentials at different states are meaningful, i.e., the potentials are determined only up to an additive constant. Realization factor is defined as the difference between the potentials at two states i and j, $g(i) - g(j)$, which measures the effort of a jump from state j to state i on the long term (average or discounted) performance. This offers an intuitive explanation for performance sensitivity and policy iteration in MDPs (in view of discrete sensitivity) and some other approaches such as the receding horizon approach [12]. The performance potential at state i can be approximated by the mean sum of the performance at the first N states on a sample path starting from state i, and hence it can be estimated online. A number of other single sample path-based estimation algorithms for potentials have been derived.

With the potentials estimated, performance derivatives (i.e., PA) can be obtained and policy iteration (i.e., MDPs) can be implemented based on a single sample path. The single sample path-based approach is practically important because it can be applied online to real engineering systems; in most cases, it does not require the system parameters to be completely known (see the examples in [5]). There are two ways to achieve the optimal performance with this approach: by perturbation analysis using performance derivatives, or by policy iteration, both can be implemented online. Stochastic approximation methods have to be used in these two cases to improve the convergence speeds and to reduce stochastic errors. For details, see [10, 15].)

Inspired by the potential-based sensitivity view of MDPs and the single sample path-based approach, a number of new research topics emerge. First, we observe that policy iteration can be implemented by using the performance derivatives. This leads to a new approach, the gradient-based policy iteration, to MDPs with correlated actions at different states [9]. A number of issues, such as the local minimal policy, the convergence of the

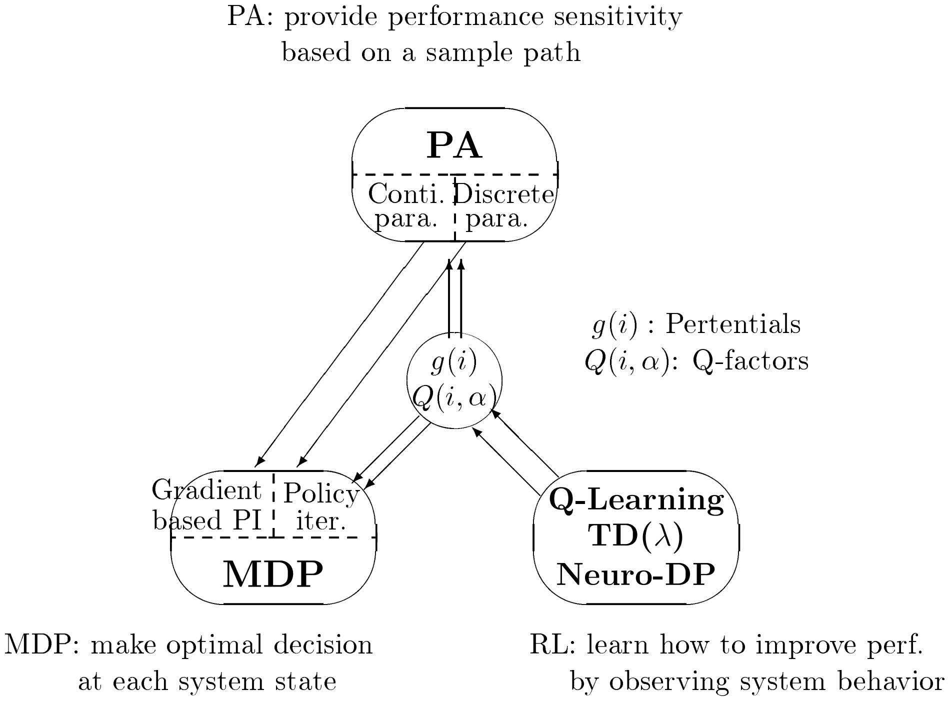


Figure 1: The Relations Among PA, MDP and RL

algorithms, are yet to be addressed. This gradient-based policy iteration can be applied to problems such as MDPs with hidden state components and the distributed control of MDPs, the details are to be worked out. The other topics include the time aggregation of MDPs [8], which was first used in [21] for perturbation analysis. When the number of controllable states is small, this approach may save computations, storage spaces, and/or the number of transitions in sample-path-based approaches.

Finally, potentials can be estimated on a single sample path. Efficient algorithms can be developed with stochastic approximation methods. Sometimes when system structure, i.e., the transition probabilities, $p(i,j)s$, are completely unknown, we cannot apply policy iteration directly by using the potentials. In this case, we can estimate the Q-factor defined as [2, 20]

$$Q(i,\alpha) = \{\sum_{j=1}^{M} p^{\alpha}(i,j)g(j)\} + f^{\alpha}(i) - \eta,$$

for every state-action pair (i,α). Smilar algorithms may be developed to estimate the performance gradients. These are the research topics in reinforcement learning.

Figure 1 illustrates the relation among PA, MDPs, and RL. All these areas are related to performance improvement and optimization. PA provides performance sensitivity with respect to both continuous and discrete parameters. Policy iteration in MDPs can be viewed as a direct consequence of discrete performance sensitivity provided by PA, This new view of PA and MDPs leads to the gradient-based policy iteration method that can be applied to some nonstandard problems such as those with correlated actions at different states. Performance potential plays an important role in both PA and MDPs; it also offers a clear intuitive interpretation for many results. Reinforcement learning, TD(λ), neuro-dynamic programming, etc, are efficient ways of estimating the performance potentials and related quantities based on sample paths. In particular, Q-learning can be used when the system structure is unknown. The potentials and Q-factors can be implemented on a single sample path; the sample path-based approach leads to on-line optimization schemes that are of practical importance to real world engineering problems such as the optimization of manufacturing and communication systems.

References

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control,* Vols. I, II, Athena Scientific, Belmont, Massachusetts, 1995.

[2] D. P. Bertsekas, and T. N. Tsitsiklis,*Neuro-Dynamic Programming,* Athena Scientific, Belmont, Massachusetts, 1996.

[3] X. R. Cao, *Realization Probabilities: The Dynamics of Queueing Systems,* Springer-Verlag, New York, 1994.

[4] X. R. Cao, “The Relation Among Potentials, Perturbation Analysis, Markov Decision Processes, and Other Topics,” *Journal of Discrete Event Dynamic Systems,* Vol. 8, 71-87, 1998.

[5] X. R. Cao, “Single Sample Path-Based Optimization of Markov Chains,” *Journal of Optimization: Theory and Application*, Vol. 100, 527-548, 1999.

[6] X. R. Cao, “A Unified Approach to Markov Decision Problems and Performance Sensitivity Analysis,” *Automatica*, Vol. 36, 771-774, 2000.

[7] X. R. Cao and H. F. Chen, “Potentials, Perturbation Realization, and Sensitivity Analysis of Markov Processes,” *IEEE Transactions on AC*, Vol. 42, 1382-1393, 1997.

[8] X. R. Cao, Z. Y. Ren, S. Bhatnagar, M. Fu, and S. Marcus, “A Time Aggregation Approach to Markov Decision Processes,” *Automatica*, to appear, 2001.

[9] X. R. Cao and H. T. Fang, “Gradient-Based Policy Iteration: An Example,” manuscript submitted to 2002 IEEE CDC.

[10] H. T. Fang and X. R. Cao, “Single Sample Path-Based Recursive Algorithms for Markov Decision Processes,” *IEEE Trans. on Automatic Control*, submitted.

[11] P. Glasserman, *Gradient Estimation Via Perturbation Analysis,* Kluwer Academic Publisher, Boston, 1991.

[12] O. Hernandez-Lerma and J. B. Lasserre, “Error bounds for rolling horizon policies in discrete-time Markov control processes,” *IEEE Transactions on AC*, Vol. 35, 1118-1124, 1990.

[13] Y. C. Ho and X. R. Cao, *Perturbation Analysis of Discrete-Event Dynamic Systems,* Kluwer Academic Publisher, Boston, 1991.

[14] Y. C. Ho and X. R. Cao, “Perturbation Analysis and Optimization of Queueing Networks,” *Journal of Optimization Theory and Applications,* Vol. 40, 4, 559-582, 1983.

[15] P. Marbach and T. N. Tsitsiklis, “Simulation-based optimization of Markov reward processes,” *IEEE Transactions on Automatic Control* Vol. 46, 191-209, 2001.

[16] S. P. Meyn, “The Policy Improvement Algorithm For Markov Decision Processes with General State Space,” *IEEE Transactions on Automatic Control*, Vol. 42, 1663-1680, 1997.

[17] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability,* Springer-Verlag, London, 1993.

[18] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley, New York, 1994.

[19] R. S. Sutton, “Learning to Predict by the Methods of Temporal Differences,” *Machine Learning*, Vol. 3, 835-846, 1988.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[21] B. Zhang and Y. C. Ho, “Performance Gradient Estimation for Very Large Finite Markov Chains,” *IEEE Transactions on AutomaticControl*, Vol. 36, 1218-1227, 1991.

Dynamic Programming Problems in Financial Engineering

Ronnie Sircar
(sircar@princeton.edu)

Abstract

I will discuss modern directions in dynamic programming approaches to stochastic control problems arising in financial engineering. Typical problems are optimal allocation of assets to maximize expected utility, minimizing expected shortfall in hedging derivative securities and options based portfolio insurance. Some of the difficulties we face are: obtaining a computational solution to a high dimensional problem in a reasonable time as observed prices change; uncertainty in model parameter estimates; lack of consensus on the most suitable utility function or risk measure. However, given these foundational problems, there is room for a feasible computational approach to define the "right" approach by its success and tractability. I shall also touch on some of the technical aspects of current methods: numerical solutions and asymptotic approximations for the related Bellman equations, particularly in models of stochastic volatility.

Optimal Investment Problems and Volatility Homogenization Approximations

Mattias Jonsson

Department of Mathematics
University of Michigan
Ann Arbor, MI 48109-1109.

Ronnie Sircar

Operations Research & Financial Engineering Department
Princeton University
Princeton, NJ 08544.

December 13, 2001

Contents

Abstract

We describe some stochastic control problems in financial engineering arising from the need to find investment strategies to optimize some goal. Typically, these problems are characterized by nonlinear Hamilton-Jacobi-Bellman partial differential equations, and often they can be reduced to linear PDEs with the Legendre transform of convex duality. One situation where this cannot be achieved is in a market with stochastic volatility. In this case, we discuss an approximation using asymptotic analysis in the limit of fast mean-reversion of the process driving volatility. Simulations illustrate that marginal improvement can be achieved with this approach even when volatility is not fluctuating that rapidly.

1 Introduction

The purpose of this article to introduce to students and researchers schooled in differential equations and scientific computing some problems involving optimal investment strategies that arise in financial engineering. These can often be characterized by nonlinear second-order Hamilton-Jacobi-Bellman (HJB) equations in three or more space dimensions, so that it is difficult to solve them numerically within the time-frame that the solution is required. We discuss an approximation method based on asymptotic analysis that is effective for models in which asset prices have randomly varying volatility. This method supposes that volatility is fluctuating rapidly with respect to the timescale of the optimization problem. The upshot of the singular perturbation analysis is to approximate the changing volatility by two averages when forecasting future expected performance. One homogenized volatility is the root-mean-square average that is common in financial calculations; the second is a harmonic average that arises from the particular structure of the HJB equations in this context. We show from simulation that even when the volatility is not fluctuating so rapidly, as the analysis assumes, the homogenization approximations serve to improve average performance, albeit marginally, over the competing constant volatility theory.

1.1 Background

In the financial services industry, stochastic modeling of prices has long been recognized as crucial to everyday management of risk, particularly that associated with derivative instruments. By derivatives, we refer to contracts such as options whose payoffs depend in some way on the behaviour of the price of some underlying asset (stock, commodity, exchange rate, for example) over some period of time. The canonical example, a call option on, say, a stock, pays nothing if the stock price X_T on a specified *expiration date* T in the future is less than the specified *strike price* K, and the difference $X_T - K$ otherwise. It is completely characterized by its payoff function

$$h(X_T) = (X_T - K)^+,$$

where $x^+ = \max\{x, 0\}$, the positive-part function.

Such securities with potentially unbounded payouts (for example if X_T becomes very large in the case of a call) can bring huge and rapid returns on relatively small investments if the stock moves the right way for the holder. But they are often associated with huge losses (for example for the writer of the call option). Large trading organizations have been using (to varying degrees) mathematical models for hedging their derivatives risk, at least since the Black-Scholes-Merton (BSM) methodology introduced in 1973 [2, 15].

In this context, let us remark that in some mathematical market models, it is possible to replicate the payoff of any derivative by continuous trading in the underlying. Such markets, which include the BSM model, are called *complete* and are much easier to analyze than the incomplete ones. In this article we will be largely interested in stochastic volatility, a feature that renders the market incomplete.

A recurrent problem in managing risks associated with derivatives positions is to maximize over suitably defined trading strategies

$$I\!E\{U(X_T, V_T)\},$$

the expectation of a *utility function* U of the future value of a controlled random *wealth* (or portfolio value) process $(V_t)_{0 \le t \le T}$ where the utility also depends on the (possibly vector-valued) state variable (X_t) at the future time T. In the situations of interest here, (X_t) is an asset price, possibly driven by other stochastic variables like volatility, U is a concave function of V_T, and the state-dependence of the utility function arises from risk associated with a derivative security $h(X_T)$.

The control variable, denoted (π_t), represents the amount the investor has in the stock. The rest $V_t - \pi_t$ is in the bank earning interest at rate r. The wealth process (V_t) therefore evolves according to

$$dV_t = \frac{\pi_t}{X_t}\, dX_t + r(V_t - \pi_t)\, dt, \tag{1}$$

which reads that the change in the portfolio value is the number of stocks held times the change in the stock price, plus interest earned on the rest in the bank. We assume the initial wealth v is given. Recent work on existence and uniqueness of optimal solutions under various models and constraints is summarized in [13].

We first give some practical examples that can be described in this framework, and then describe one approach to this problem using dynamic programming.

1.2 Examples

In a complete market, there exists an amount $v^\star$ of initial capital and a strategy $(\pi_t^\star)$ such that

$$V_T = h(X_T) \qquad \text{with probability 1.}$$

In this case, $(\pi_t^\star)$ is called the replicating strategy for the derivative. Alternatively, it is the *hedging* strategy for a short position in the derivative since it is often used to cancel *exactly* (in theory) the risk of a short derivative position. The quantity $v^\star$ is called the *Black-Scholes price* of the derivative.

The problems we describe here are concerned with the gap

$$h(X_T) - V_T$$

which may not be zero with probability 1 either because the initial capital $v < v^\star$ or because of market incompleteness.

1.2.1 Partial Hedging

Suppose the investor has written (sold) a derivative, but wants to insure or hedge against his future (uncertain) liability with a trading strategy in the stock. The problem is to find a strategy $(\pi_t)_{0\leq t\leq T}$ such that, starting with initial capital v, the terminal value of the *hedging portfolio* V_T comes as close as possible to $h(X_T)$. We want the performance of the strategy to be penalized for falling short, with the actual size of the shortfall taken into account, but when the strategy overshoots the target, the size of the overshoot has no bearing on the measure of *risk*. This problem has been studied under various assumptions recently by Föllmer-Leukert [7] and Cvitanić & Karatzas [3], where existence and uniqueness results are established. In Section 7, we discuss *computation* of optimal strategies when volatility is changing randomly.

If we are given enough initial capital v, we can make sure that the terminal wealth V_T exceeds $h(X_T)$ with probability 1. The smallest such v is called the *superhedging price* of the derivative $h(X_T)$. Clearly, the partial hedging problem is only interesting if v is strictly smaller than the superhedging price.

For the explicit computations and examples here, the penalty functon we use is

$$I\!\!E\{\frac{1}{p}((h(X_T) - V_T)^+)^p\},$$

the *one-sided* pth moment. The parameter $p > 1$ allows control of one's risk aversion. We will present simulations corresponding to the case $p = 1.1$ in Section 7. The limiting case $p = 1$ has a particular economic significance since the penalty function then corresponds to a coherent measure of risk (see [3]), but also leads to certain implementation difficulties. This is why we pick $p = 1.1$.

We also insist that $V_t \geq 0$ for all $0 \leq t \leq T$ almost surely for the strategy to be admissible. That is, the hedging portfolio is bounded below by zero, a portfolio constraint. The problem can be reformulated as a (state-dependent) utility maximization with

$$U(x,v) = \frac{1}{p}\left[h(x)^p - ((h(x)-v)^+)^p\right].$$

For fixed x, this is concave in v on $(0,\infty)$, strictly concave on $(0,h(x))$ and satisfies $U(x,0) = 0$, $U(x,v) = \frac{1}{p}h(x)^p$ for $v \geq h(x)$, see Figure 3.

We call this problem *partial hedging* of the derivative risk. It is the main example that we pick to illustrate the asymptotic analysis in Section 6; it is also treated in [12].

1.2.2 Mean-Square Pricing

The problem is to find the initial wealth level such that the minimum variance

$$\inf_{(\pi_t)} I\!E\{(h(X_T) - V_T)^2\}$$

is minimized as a function of v. This is then defined to be the option price. In our framework, this corresponds to the utility function

$$U(x,v) = h(x)^2 - (h(x)-v)^2.$$

Such a quadratic state-dependent utility is considered in [5, 1], for example. In complete markets the expectation above is zero for the optimal v because the option can be replicated exactly with initial capital $v^\star$, the Black-Scholes price. The technique of mean-square pricing is most interesting in incomplete markets, such as markets with stochastic volatility.

1.2.3 Options Based Portfolio Insurance

Fund managers are interested in selling trading opportunities in which the clients can take advantage of a rising market but are insured against losses in the sense that the value of the fund is bounded below by a number that is either constant or depends on a future asset price. More precisely, the client is guaranteed that the portfolio value V_T at time $T > 0$ is at least $h(X_T)$, i.e. the value of an option with payoff $h(X_T) \geq 0$. This is studied in complete markets in [6], for example, and corresponds to the utility function

$$U(x,v) = \begin{cases} \tilde{U}(v) & \text{if } v \geq h(x) \\ -\infty & \text{otherwise} \end{cases}$$

where $\tilde{U}$ is an increasing, concave (state-independent) utility function.

1.2.4 Utility-Indifference Pricing

In this mechanism for pricing, described in [4] for example, the (selling) price of a derivative with payoff $h(X_T)$ is defined to be the extra initial compensation the seller would have to receive so that he/she is *indifferent with respect to maximum expected utility* between having the liability of the short derivative position or just trading in the stock.

Let

$$H(v) = \sup_{\pi} \mathbb{E}\{\tilde{U}(V_T - h(X_T)) \mid \text{initial capital is } v\},$$

where $\tilde{U}$ is an increasing concave utility function and we show only the v dependence of H. Here the seller can trade in the stock, but at time T has liability $-h(X_T)$. Thus the relevant state-dependent utility function is

$$U(x, v) = \tilde{U}(v - h(x)),$$

where $\tilde{U}$ is an increasing, concave (state-independent) utility function. Let

$$M(v) = \sup_{\pi} \mathbb{E}\{\tilde{U}(V_T) \mid \text{initial capital is } v\},$$

the solution to the same problem without the derivative position. Then the (seller's) price P of the derivative security is defined by

$$H(v + P) = M(v).$$

In a complete market, this mechanism recovers the Black-Scholes price $v^\star$ which does not depend on the wealth level v of the seller. The same desirable wealth-independent property can be attained in an incomplete market by assuming an exponential utility $\tilde{U}(v) = -e^{-\gamma v}$, where γ is a risk-aversion coefficient.

2 The Merton Problem

To begin our exposition of the methods for dealing with utility maximization problems, we begin with the simplest and original continuous-time stochastic control problem in finance, the Merton optimal asset allocation problem. The original reference is [14] and this is reprinted in the book [16]. Here we concentrate on maximizing expected utility at a final time horizon T, mainly because we are later interested in problems associated with derivative contracts that have a natural terminal time; other versions of the asset allocation problem consider optimizing utility of consumption over all times, possibly with an infinite horizon.

An excellent mathematical introduction to this type of stochastic control problem, as well as stochastic modelling and connections to differential equations, is the book by Øksendal [17].

2.1 The BSM Model

The Black-Scholes-Merton (BSM) model takes the price of the underlying asset (for example a stock) to have a deterministic growth component (measured by an average rate of return μ) and uncertainty or risk generated (for mathematical convenience) by a Brownian motion and quantified by a volatility parameter σ. The price X_t at time t satisfies the stochastic differential equation

$$\frac{dX_t}{X_t} = \mu\, dt + \sigma\, dW_t, \tag{2}$$

where (W_t) is a standard Brownian motion. This model is simple and extremely tractable for a number of important problems (particularly the problem of *pricing* and hedging derivative securities), and as such, it has had enormous impact over the past thirty years. It is the primary example of a complete, continuous-time market.

An investor starts with capital v at time $t = 0$ and has the choice to invest in the stock or put his/her money in the bank earning interest at the (assumed constant) interest rate r. He continuously balances his portfolio, adjusting the weights between these two choices so as to maximize his expected utility of wealth at time T. That is, he has a utility function U that is increasing (he prefers more wealth to less) and concave (representing risk-aversion) through which he values one investment strategy over another.

If π_t is the dollar amount of stock that the investor holds at time t, we define his wealth process (V_t) by (1). The investor looks for a strategy $\pi_t^\star$ that maximizes

$$\mathbb{E}\{U(V_T)\}$$

starting with initial wealth v and under the constraint of nonbankruptcy: $V_t \geq 0$ for all $0 \leq t \leq T$ (almost surely).

Without loss of generality, we shall henceforth take the interest rate r to be zero. This can be justified simply by measuring all capital values *in units of the money market account*, i.e. replacing (X_t), (V_t) and (π_t) by $(X_t e^{-rt})$, $(V_t e^{-rt})$ and $(\pi_t e^{-rt})$, respectively. We recover the same equations as long as we relabel $\mu - r$ as μ. With this convention, the equations for (X_t, V_t) in terms of the control π_t are

$$\begin{aligned} \frac{dX_t}{X_t} &= \mu\, dt + \sigma\, dW_t, && (3) \\ dV_t &= \mu \pi_t\, dt + \sigma \pi_t\, dW_t, && (4) \end{aligned}$$

where μ is now the excess growth over the risk free rate. Notice that the process (V_t) is autonomous, i.e. it does not directly involve (X_t).

2.2 Dynamic Programming and the Bellman Equation

One approach to this problem is to observe that if π_t were chosen to be a (nice) function of V_t, the controlled process (V_t) defined by (4) would be a Markov process. Therefore, *if* the optimal strategy were of this so-called *feedback form*, we could take advantage of the structure and study the evolution of the optimal expected utility as a function of the starting wealth and starting time. To this end, we define the *value function*

$$H(t,v) = \sup_{(\pi_s)_{s\in[t,T)}} \mathbb{E}\{U(V_T) \mid V_t = v\},$$

where by hypothesis, H is a function of only the starting time t and wealth v (and not, for instance, details of the path). We shall also use the shorthand

$$\mathbb{E}_{t,v}\{\cdot\} = \mathbb{E}\{\cdot \mid V_t = v\}.$$

Notice that (X_t) has vanished from the problem because everything can be stated in terms of the process (V_t). This is a direct consequence of the geometric Brownian motion model (3) assumed here. If μ or σ were general functions of X_t, the reduction in dimension would not be possible. In the optimization problems involving derivative contracts, X_T appears in the utility function and, again, we cannot eliminate the x variable.

The following is a loose derivation of the Bellman equation for the value function H. The idea is to divide the control process (π_s) over the the time interval $[t,T)$ into the (constant) control π_t over $[t,t+dt)$ and the rest (π_s) over $s \in [t+dt,T)$. We then optimize over these two parts separately. Formally, conditioning on the wealth at time $t+dt$ and using the iterated expectations formula, this looks like

$$\begin{aligned}
H(t,v) &= \sup_{(\pi_s)_{s\in[t,T)}} I\!\!E_{t,v}\left\{I\!\!E\{U(V_T) \mid V_{t+dt}\}\right\} \\
&= \sup_{\pi_t} I\!\!E_{t,v}\left\{\sup_{(\pi_s)_{s>t}} I\!\!E\{U(V_T) \mid V_{t+dt}\}\right\} \\
&= \sup_{\pi_t} I\!\!E_{t,v}\left\{H(t+dt, V_{t+dt})\right\} \\
&= \sup_{\pi_t} I\!\!E_{t,v}\left\{H(t,v) + H_t(t,v)\,dt + \mathcal{L}_v H(t,v)\,dt + \sigma\pi_t H_v(t,v)\,dW_t\right\},
\end{aligned}$$

where $\mathcal{L}_v$ denotes the infinitesimal generator of (V_t), namely

$$\mathcal{L}_v = \frac{1}{2}\sigma^2\pi_t^2\frac{\partial^2}{\partial v^2} + \mu\pi_t\frac{\partial}{\partial v},$$

and we have used Itô's formula in the last step. Note that $\mathcal{L}_v$ depends on the control π, but we do not denote the dependence in this notation. The last term is zero since $I\!\!E\{dW_t\} = 0$. We therefore obtain

$$H_t + \sup_{\pi_t} \mathcal{L}_v H = 0. \tag{5}$$

Notice that the computation exploits the Markovian structure to reduce the optimization over the whole time period to successive optimizations over infinitesimal time intervals. This description of the optimal strategy for the long run, that is to do as well as one can over the short run, is called the Bellman principle.

The Bellman partial differential equation (5) applies in the domain $t < T$ and $v > 0$ with the terminal condition

$$H(T,v) = U(v)$$

and the boundary condition $H(t,0) = 0$ enforcing the bankruptcy constraint.

If we can find a smooth solution $H(t,v)$ to which Itô's formula can be applied, it follows from a verification theorem that H gives the maximum expected utility and the optimal strategy is given by a Markov control $\pi_t = \pi(V_t)$. For details, see [8].

In our case, the Bellman equation is

$$H_t + \sup_{\pi}\left(\frac{1}{2}\sigma^2\pi^2 H_{vv} + \mu\pi H_v\right) = 0.$$

The internal optimization is simply to find the extreme point of a quadratic. We shall assume (and it can be shown rigorously) that the value function inherits the convexity of the utility function. Moreover, under reasonable conditions, it is *strictly* convex for $t < T$ even if U is not. This follows from the diffusive part of the equation. Therefore $H_{vv} < 0$. The optimization over π is not constrained because we have not assumed any constraints on the trading strategies themselves, as long as the wealth stays positive, and therefore the quadratic is maximized by

$$\pi^\star = -\frac{\mu H_v}{\sigma^2 H_{vv}}. \tag{6}$$

Substituting with this π, we can rewrite the Bellman equation as

$$H_t - \frac{\mu^2}{2\sigma^2}\frac{H_v^2}{H_{vv}} = 0. \tag{7}$$

2.3 Solution by Legendre Transform

We further take advantage of the assumed convexity of the value function to define the Legendre transform

$$\hat{H}(t,z) = \sup_{v>0}\left\{H(t,v) - zv\right\},$$

where $z > 0$ denotes the dual variable to v. The value of v where this optimum is attained is denoted by $g(t,z)$, so that

$$g(t,z) = \inf\{v > 0 \mid H(t,v) \geq zv + \hat{H}(t,z)\}.$$

The two functions $g(t,z)$ and $\hat{H}(t,z)$ are closely related and we shall refer to either one of them as the (convex) dual of H. In this article, we will work mainly with the function g, as it is easier to compute numerically and suffices for the purposes of computing optimal trading strategies. The function $\hat{H}$ is related to g by $g = -\hat{H}_z$.

At the terminal time, we denote

$$\begin{aligned}\hat{U}(x,z) &= \sup\{U(v) - zv \mid 0 < v < \infty\}\\ G(x,z) &= \inf\{v > 0 \mid U(v) \geq zv + \hat{U}(z)\}.\end{aligned}$$

This is illustrated in Figure 1.

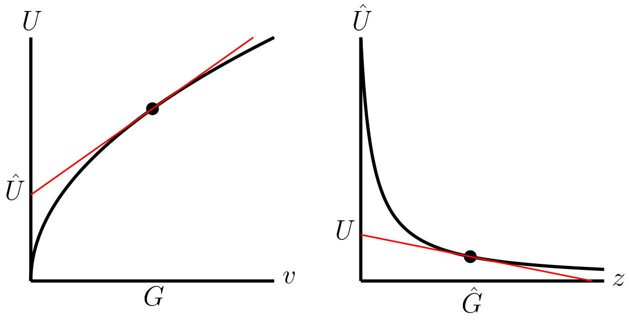


Figure 1: *The Legendre transform. The left curve is the graph of $U(v)$ the utility function and the tangent line has slope z for $v = G(z)$ and has vertical intercept $\hat{U}(z)$. The right curve is the graph of $\hat{U}(z)$ and the tangent line has slope $-v$ at the point $z = \hat{G}(v)$ and has vertical intercept $U(v)$.*

Assuming that H is strictly concave and smooth as a function of v, we have that

$$H_v(t, g(t,z)) = z,$$

or $g = H_v^{-1}$, whereby, in economics parlance, g is sometimes referred to as the inverse of marginal utility.

By differentiating this with respect to t and z we recover the following rules relating the derivatives of the value function to the dual function g:

$$H_{tv} = -\frac{g_t}{g_z} \qquad H_{vv} = \frac{1}{g_z} \qquad H_{vvv} = -\frac{g_{zz}}{g_z^3},$$

where the left is evaluated at $(t, g(t,z))$. Differentiating equation (7) with respect to v and substituting gives an autonomous equation for g:

$$\begin{aligned} g_t + \frac{\mu^2}{2\sigma^2} z^2 g_{zz} + \frac{\mu^2}{\sigma^2} z g_z &= 0, \\ g(T,z) &= G(z), \end{aligned}$$

in $t < T$ and $z > 0$. The key observation is that this is now a *linear* PDE.

2.4 Optimal Strategy

We are not usually interested in the value function, but rather in the optimal investment strategy. From (6), we can compute the optimal stock holding as a feedback formula in terms of derivatives of the value function. In terms of the dual function g, it is given by

$$\pi_t^\star = -\frac{\mu}{\sigma^2} z g_z(t,z).$$

Thus for a given Merton problem, we solve the linear PDE for g and recover the investment amount $\pi^\star$ from its first derivative. All that remains is to use the value of the dual variable z that corresponds to the current time and wealth level (t,v). This is obtained from the relation

$$g(t,z) = v.$$

2.5 Example: Power Utility

The success of the Merton approach is due primarily to an appealing explicit formula for certain simple utility functions. For an isoelastic (or power) utility function of the form

$$U(v) = v^\gamma/\gamma, \qquad 0 < \gamma < 1,$$

the Legendre duals G and $\hat{U}$ are given by

$$G(z) = z^{\frac{1}{\gamma-1}} \quad \text{and} \quad \hat{U}(z) = \frac{1-\gamma}{\gamma} z^{\frac{\gamma}{\gamma-1}}.$$

These are illustrated in Figure 2. In this case, the linear PDE for g admits a separable solution of the form

$$g(t,z) = z^{\frac{1}{\gamma-1}} u(t),$$

for some function $u(t)$ we can compute. It follows that for a given (t,v),

$$g(t,z) = v$$

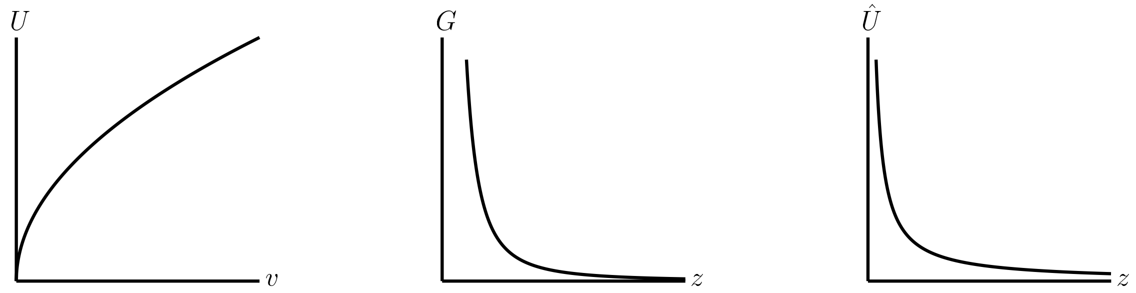


Figure 2: *Terminal conditions for the Merton problem with power utility.*

and the optimal strategy is given by

$$\pi_t^\star = -\frac{\mu}{\sigma^2} z g_z = -\frac{\mu}{\sigma^2}\frac{1}{\gamma - 1} g = \frac{\mu}{\sigma^2(1-\gamma)} v.$$

That is, the optimal strategy is to hold the fraction

$$M = \frac{\mu}{\sigma^2(1-\gamma)}$$

of current wealth in the risky asset (the stock) and to put the rest in the bank. As the stock price rises, this strategy says to sell some stock so that the fraction of the portfolio comprised of the risky asset remains the same.

The fraction M is known as the Merton ratio. More importantly, as we see next, this *fixed-mix* result generalizes to multiple securities as long as they are also assumed to be geometric Brownian motions.

2.6 The Multi-Dimensional Merton Problem

We have a market with $n \geq 1$ stocks where the ith stock price $(X_t^{(i)})$ (in units of the money market account) is modeled by

$$\frac{dX_t^{(i)}}{X_t^{(i)}} = \mu_i \, dt + \sum_{j=1}^{n} \sigma_{ij} dW_t^{(j)},$$

where the μ_i are the growth rates less the risk free rate and $(W_t^{(i)})$ are independent Brownian motions. The price processes are correlated through the diffusion terms. We denote by (X_t) the vector price process, μ is the vector of return rates, and σ is the $n \times n$ volatility matrix.

An investor chooses amounts $\pi_t = (\pi_t^{(1)}, \cdots, \pi_t^{(n)})$ to invest in each stock, and puts the rest in the bank. Again (π_t) denotes the vector control process. His wealth process is

$$dV_t = \pi_t^T \mu \, dt + \pi_t^T \sigma \, dW_t,$$

where ^{T} denotes transpose. Recall that in the one-dimensional Merton's problem, the stock price process disappeared from the problem. This happens in the multidimensional case, too, and the Bellman equation for the value function $H(t,v)$ becomes

$$H_t + \sup_\pi \left(\frac{1}{2} \pi^T \sigma \sigma^T \pi H_{vv} + \pi^T \mu H_v \right) = 0.$$

This can be transformed to a linear PDE using the Legendre transform. In the case of power utility, $U(v) = v^\gamma/\gamma$, the optimal strategy is given by

$$\pi_t^\star = \frac{1}{1-\gamma}(\sigma\sigma^T)^{-1}\mu v.$$

Thus, the optimal strategy is again to hold fixed fractions of current wealth in the risky assets (the stocks) and to put the rest in the bank. The fractions are given by the vector of Merton ratios $\frac{1}{1-\gamma}(\sigma\sigma^T)^{-1}\mu$.

3 State-Dependent Utility Maximization: Constant Volatility

In this section, we describe briefly the generalization of the utility maximization problem to the case when the process modeling the price of the stock cannot be eliminated. We assume still that it is a geometric Brownian motion, but that the utility depends on its final value, as in the examples outlined in Section 1.2.

The value function is

$$H(t,x,v) = \sup_\pi I\!\!E\{U(X_T, V_T) \mid X_t = x, V_t = v\}$$

and it is conjectured to satisfy the Bellman equation

$$\begin{aligned} 0 &= H_t + \mu x H_x + \frac{1}{2}\sigma^2 x^2 H_{xx} + \sup_\pi\left(\pi\mu H_v + \pi\sigma^2 x H_{xv} + \frac{1}{2}\pi^2\sigma^2 H_{vv}\right) \\ &= H_t + \mu x H_x + \frac{1}{2}\sigma^2 x^2 H_{xx} - \frac{(\mu H_v + \sigma^2 H_{xv})^2}{2\sigma^2 H_{vv}}. \end{aligned} \tag{8}$$

3.1 Convex Duals

Proceeding as in the Merton problem, we introduce the duals

$$\begin{aligned} \hat{H}(t,x,z) &= \sup\{H(t,x,v) - zv \mid 0 < v < \infty\} \\ g(t,x,z) &= \inf\{v > 0 \mid H(t,x,v) \geq zv + \hat{H}(t,x,z)\}. \end{aligned}$$

Notice that the we take the convex duals with respect to the variable v only. What is of interest is how the extra state variable x affects the transformation of the Bellman PDE. The transformation rules for the derivatives are given by

$$H_v = z \qquad \begin{aligned} H_t &= \hat{H}_t \\ H_x &= \hat{H}_x \end{aligned} \qquad \begin{aligned} H_{vv} &= -\frac{1}{\hat{H}_{zz}} \\ H_{xv} &= -\frac{\hat{H}_{xz}}{\hat{H}_{zz}} \end{aligned} \qquad H_{xx} = \hat{H}_{xx} - \frac{\hat{H}_{xz}^2}{\hat{H}_{zz}}.$$

This implies that

$$\hat{H}_{xx} = H_{xx} - \frac{H_{xv}^2}{H_{vv}}. \tag{9}$$

If we rewrite (8) as

$$H_t + \mu x H_x + \frac{1}{2}\sigma^2 x^2 \left(H_{xx} - \frac{H_{xv}^2}{H_{vv}} \right) - \mu x \frac{H_v H_{xv}}{H_{vv}} - \frac{\mu^2}{2\sigma^2} \frac{H_v^2}{H_{vv}} = 0,$$

we see that the first nonlinear term is transformed into a linear term because of (9), and the third is as in the Merton problem described in Section 2.3 where it became linear in the dual variables. Moreover the same happens to the second nonlinear term and we get

$$\hat{H}_t + \frac{1}{2}\sigma^2 x^2 \hat{H}_{xx} + \frac{1}{2}\frac{\mu^2}{\sigma^2} z^2 \hat{H}_{zz} + \mu x H_x - \mu x z \hat{H}_{xz} = 0. \tag{10}$$

Note that the first nonlinear term would not have been transformed to a linear one if the (V_t) process was not perfectly correlated with the (X_t) process. This is the situation in the stochastic volatility models we discuss later because volatility is not a traded asset.

We can recover the PDE for the other dual function $g(t, x, z)$ using

$$g = -\hat{H}_z$$

and differentiating the PDE for $\hat{H}$ with respect to z to give

$$g_t + \frac{1}{2}\sigma^2 x^2 g_{xx} + \frac{\mu^2}{\sigma^2} z g_z + \frac{1}{2}\frac{\mu^2}{\sigma^2} z^2 g_{zz} - \mu x z g_{xz} = 0. \tag{11}$$

3.2 Optimal Strategy

Having solved the *linear* PDE for either g or $\hat{H}$, we recover the optimal strategy from

$$\begin{aligned} \pi^* &= -\frac{(\mu H_v + \sigma^2 x H_{xv})}{\sigma^2 H_{vv}} \\ &= \frac{\mu}{\sigma^2} z \hat{H}_{zz} - x \hat{H}_{xz} \\ &= x g_x - \frac{\mu}{\sigma^2} z g_z. \end{aligned}$$

The value of z to be used when the stock and wealth are (x, v) at time t are found from

$$g(t, x, z) = v.$$

3.3 Example: Partial Hedging

In this problem, we are partially hedging a derivative with payoff $h(X_T) \geq 0$ at time T, as explained in Section 1.2.1. We assume that v is strictly less than the superhedging price of the derivative. In the present case of constant volatility, the market is complete and the superhedging price is the Black-Scholes price $v^\star$. It is the (uniquely determined) initial capital for a replicating strategy for the derivative.

Recall that the state-dependent utility function is

$$U(x, v) = \frac{1}{p}\left(h^p - \left((h - v)^+\right)^p \right),$$

where $h = h(x)$ and $p > 1$. This has convex duals

$$G(x,z) = \left(h - z^{\frac{1}{p-1}}\right)^+$$
$$\hat{U}(x,z) = \left(\frac{1}{p}h^p + \frac{p-1}{p} z^{\frac{p}{p-1}} - zh\right) \mathcal{H}\left(h - z^{\frac{1}{p-1}}\right),$$

where $\mathcal{H}$ is the Heaviside (step) function. See Figure 3.

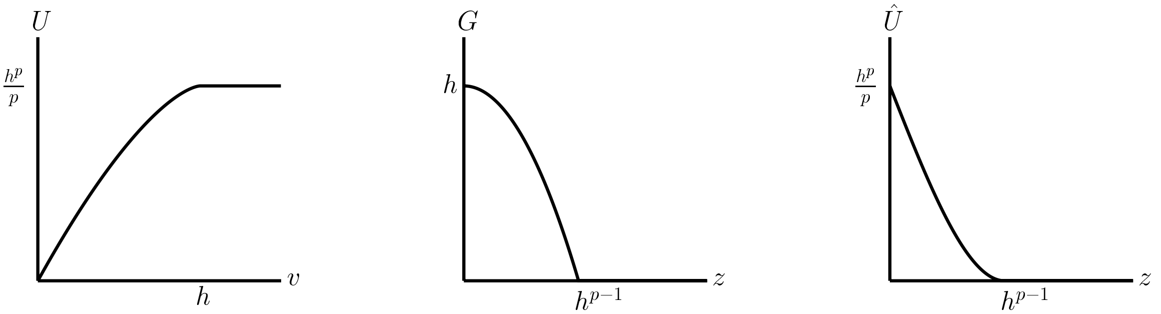


Figure 3: *Terminal conditions for partial hedging.*

The linear PDEs (10) and (11) have a probabilistic interpretation as the expectation of a function of a (two-dimensional) Markov process, but this can be reduced to a function of just the terminal stock price (because of the degeneracy of (11)). It can be shown that

$$g(t,x,z) = \mathbb{E}^\star_{t,x}\{\left(h(X_T) - \tilde{c}X_T^{-\kappa}\right)^+\}, \tag{12}$$

where $\kappa = \frac{\mu}{\sigma^2(p-1)}$, and $\mathbb{E}^\star$ denotes expectation with respect to the (so-called risk-neutral) probability measure $\mathbb{P}^\star$ under which

$$dX_t = \sigma X_t dW_t^\star,$$

with $(W_t^\star)$ a $\mathbb{P}^\star$-Brownian motion, and

$$\tilde{c} = \tilde{c}(t,x,z) = zx^{-\mu/\sigma^2} \exp\left(\frac{1}{2}(\frac{\mu^2}{\sigma^2} - \mu)(T-t)\right).$$

This representation is useful in finance where linear diffusion PDEs are associated with pricing equations for derivative securities. We do not stress this interpretation here, but merely comment that the optimal strategy is to trade the stock in such a way so as to replicate the *target wealth*, which here is a European derivative contract with the modified payoff function

$$\left(h(X_T) - \tilde{c}X_T^{-\kappa}\right)^+.$$

This is illustrated in Figure 4. The number z is determined by

$$g(t,x,z) = \text{current wealth}.$$

For computational purposes, the only requirement is to solve the PDE for g or $\hat{H}$. It turns out that g is more amenable to explicit computation.

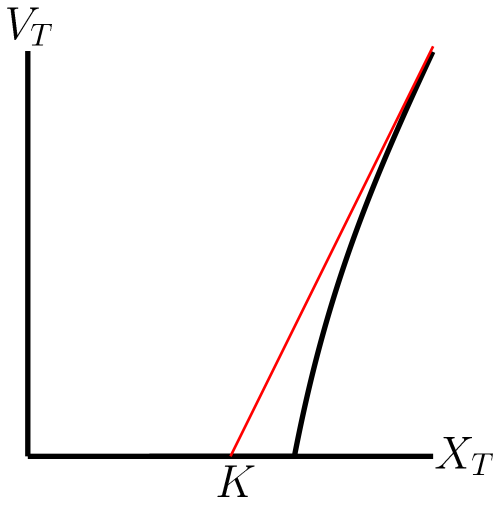


Figure 4: *Target wealth for partial hedging of a European call option. The thin line is the payoff function of the original call. When partial hedging, they payoff is replaced by the thick line.*

3.4 Explicit Computation for a Call Option

Sometimes we can get closed formulas for the function $g(t,x,z)$ and its derivatives, and hence the optimal strategy in the utility maximization. This greatly increases computational speed and is the case, for instance, when we are partially hedging a call option $h(x)=(x-K)^+$. Indeed, let us define $\tilde{d}_2=\tilde{d}_2(t,x,z)$ as the (unique) solution to

$$xe^{-\frac{1}{2}\sigma^2\tau-\sigma\sqrt{\tau}\tilde{d}_2}-K-z^{\frac{1}{p-1}}e^{\frac{\mu^2\tau}{2(p-1)\sigma^2}+\frac{\mu\sqrt{\tau}\tilde{d}_2}{(p-1)\sigma}}=0,$$

Given K and values of t,x,z, this can be solved numerically, since the left hand side is a strictly decreasing function of $\tilde{d}_2$. We then get from (12):

$$\begin{aligned} g(t,x,z) &= \int_{-\infty}^{\tilde{d}_2}\left(xe^{-\frac{1}{2}\sigma^2\tau-\sigma\sqrt{\tau}\xi}-K-z^{\frac{1}{p-1}}e^{\frac{\mu^2\tau}{2(p-1)\sigma^2}+\frac{\mu\sqrt{\tau}\xi}{(p-1)\sigma}}\right)\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}\xi^2}\,d\xi \\ &= xN(\tilde{d}_1)-KN(\tilde{d}_2)-z^{\frac{1}{p-1}}e^{\frac{p\mu^2\tau}{2(p-1)^2\sigma^2}}N(\tilde{d}_3), \qquad (13) \end{aligned}$$

where

$$\tilde{d}_1=\tilde{d}_2+\sigma\sqrt{\tau} \qquad \tilde{d}_3=\tilde{d}_2-\frac{\mu\sqrt{\tau}}{(p-1)\sigma}.$$

This is similar to the celebrated Black-Scholes formula for the price of a call option, and in fact reduces to this as $z\to 0$.

In special cases such as $p=2$ one can also get a closed formula for $\hat{H}(x,z)$, but we only need g in order to find the optimal strategy:

$$\begin{aligned} \frac{\pi^*(t,x,z)}{x} &= g_x-\frac{\mu z}{\sigma^2 x}g_z \\ &= N(\tilde{d}_1)+\frac{\mu z^{\frac{1}{p-1}}}{(p-1)\sigma^2 x}\exp\left(\frac{p\mu^2\tau}{2(p-1)^2\sigma^2}\right)N(\tilde{d}_3). \end{aligned}$$

This strategy was derived in [7] and we shall refer to it as the Föllmer-Leukert strategy. It is implemented in the simulations of Section 7.

4 Stochastic Volatility Models

The BSM model supposes that asset price volatility is constant. This is contradicted by empirical evidence of fluctuating historical volatility. More crucially, volatility as a measure of risk is seen by traders as the most important variable (after the price of the underlying asset itself) in driving probabilities of profit or loss.

Stochastic volatility models which replace σ in (2) by a random process (σ_t) arise not just because of empirical evidence of historical volatility's "random characteristics", but also from considerations of market hysteria, uncertainty in estimation, or they could be used to simulate non-Gaussian (heavy-tailed) returns distributions. They describe a much more complex market than the Black-Scholes model, and this is reflected in the increase in difficulty of the derivative hedging problems we shall describe. They were introduced in the academic literature in the late 1980's [11] and are popular in the industry today.

A model for stock prices in which volatility (σ_t) is a random process starts with the stochastic differential equation

$$\frac{dX_t}{X_t} = \mu\,dt + \sigma_t\,dW_t, \tag{14}$$

the analogue of (2).

A key aim in the modeling is to say as little about volatility as possible so that we are not tied to a specific model. Since volatility is not observed directly, there is a paucity of consistent econometric information about its behaviour. One feature that most empirical studies point out, and which squares with common experience, is that volatility is *mean-reverting*: it is not wandering into far-flung excursions, but seems to be pulled upwards when it is low and downwards when it is high. It is convenient to model volatility as a function of a simple mean-reverting (ergodic) Markov diffusion process (Y_t), for example an Ornstein-Uhlenbeck (OU) process. We discuss the class of models

$$\sigma_t = f(Y_t); \qquad dY_t = \alpha(m - Y_t)\,dt + \beta\,d\hat{Z}_t, \tag{15}$$

where f is a positive bounded function through which the generality of possible volatility models is obtained. In fact the asymptotic results for derivative pricing in [10] are insensitive to all but a few general features of f, and the way the method there is calibrated means that this function never has to be chosen.

In (15), $(\hat{Z}_t)$ is a Brownian motion modeling the fine-scale volatility fluctuations that is correlated with the other Brownian motion (W_t). Itô diffusions provide a simple way to model the much observed "leverage effect" that volatility and stock price shocks are negatively correlated: when volatility goes up, stock prices tend to fall. The instantaneous correlation coefficient ρ, where

$$I\!E\{dW_t\,d\hat{Z}_t\} = \rho\,dt$$

measures this asymmetry in the probability distribution of future stock prices: $\rho < 0$ generates a fatter left tail.

4.1 Fast Mean-Reverting Stochastic Volatility

The effects of *fast mean-reversion* in volatility were studied in [10]. Mean-reversion is mathematically described by ergodicity and refers to the characteristic *time* it takes for an ergodic process to get back to the mean-level of its long-run distribution. The separation of scales that we shall exploit is that while stock prices change on a tick-by-tick time-scale (minutes), volatility changes more slowly, on the scale of a few days, but still fluctuates rapidly over the time-scale (months) of a derivative contract. This phenomenon of bursty or clustering volatility is characterized as *fast mean-reversion* in the models we look at. That is, the volatility process is fast mean-reverting with respect to the long time-scale (months) of reference. (It is slow mean-reverting with respect to the tick time-scale, by which it is sometimes described).

The important parameter in (15) is α, the rate of mean-reversion of the volatility-driving process (Y_t). Fast mean-reversion describes the limit α tending to infinity with $\beta^2/2\alpha$, the variance of the long-run distribution of the OU process, fixed. A detailed study of high-frequency $S\&P500$ data, where the large rate of volatility mean-reversion was established by a variety of methods, appears in [9]. As an illustration, Figure 5 shows simulated volatility paths for one of the models above in which $\alpha = 1$ in the top graph and $\alpha = 200$ in the bottom graph. Of course volatility is not directly observable as a time-series, which complicates estimation issues and makes it desirable to have a theory that is robust to volatility modeling.

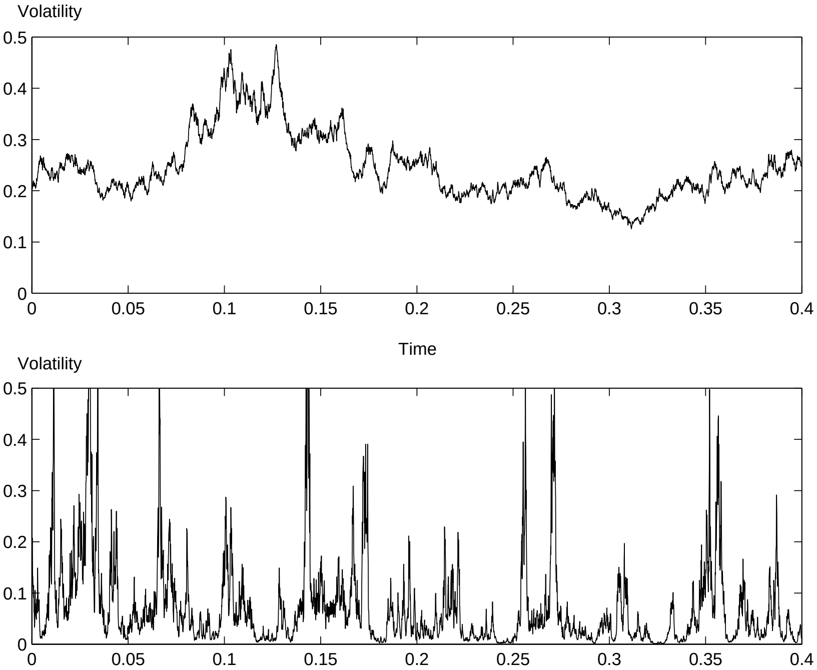


Figure 5: *Simulated volatility for small ($\alpha = 1$, top) and large ($\alpha = 200$, bottom) rates of mean-reversion for the OU model, with the choice $f(y) = e^y$. Note how high volatility appears in short bursts in the latter case.*

In Section 6, we exploit this separation of scales to construct an approximation for the state-dependent utility optimization problem under stochastic volatility, which is discussed in the next section.

5 Utility Maximization under Stochastic Volatility

In this section, we look at the state-dependent utility maximization problem and how it is modified for a market with stochastic volatility. Under the class of mean-reverting stochastic volatility models (14)-(15), we study as before the Bellman equation of dynamic programming: the value function

$$H(t,x,y,v) = \sup_{\pi} I\!E\{U(X_T,V_T) \mid X_t = x, V_t = v, Y_t = y\}$$

is conjectured to satisfy:

$$H_t + \mathcal{L}_{x,y}H - \frac{(\mu H_v + f(y)^2 x H_{xv} + \rho\beta f(y) H_{yv})^2}{2f(y)^2 H_{vv}} = 0, \tag{16}$$

where

$$\mathcal{L}_{x,y}H = \mu x H_x + \alpha(m-y)H_y + \frac{1}{2}f(y)^2 x^2 H_{xx} + \rho\beta f(y) x H_{xy} + \frac{1}{2}\beta^2 H_{yy},$$

the infinitesimal generator of the process (X_t, Y_t). The domain is $x > 0$, $-\infty < y < \infty$, $v > 0$ and $t < T$ and the terminal condition is

$$H(T,x,y,v) = U(x,v).$$

If a smooth solution (to which Itô's formula can be applied) can be found, a verification theorem shows that the optimal strategy is given in feedback form by

$$\pi_t^\star = -\frac{\mu H_v + f(y)^2 x H_{xv} + \rho\beta f(y) H_{yv}}{f(y)^2 H_{vv}}. \tag{17}$$

We proceed to study the dual optimization problem for the Legendre transform (with respect to v) of the value function. Defining

$$\begin{aligned}\hat{H}(t,x,y,z) &= \sup\{H(t,x,y,v) - zv \mid 0 < v < \infty\}\\ g(t,x,y,z) &= \inf\left\{v > 0 \mid H(t,x,y,v) \geq zv + \hat{H}(t,x,y,z)\right\},\end{aligned}$$

the dual functions g and $\hat{H}$ satisfy simpler-looking equation. In the case of g it reads

$$\begin{aligned} g_t + \mathcal{L}_{x,y}g + \frac{1}{2}\frac{\mu^2}{f(y)^2}z^2 g_{zz} - \mu x z g_{xz} - \frac{\rho\beta\mu}{f(y)} z g_{yz} + \frac{\mu^2}{f(y)^2} z g_z &\\ -\mu x g_x - \frac{\rho\beta\mu}{f(y)} g_y - \frac{1}{2}\beta^2(1-\rho^2)\left[\frac{2g_y g_{yz}}{g_z} - \frac{g_y^2}{g_z^2}g_{zz}\right] &= 0, \end{aligned} \tag{18}$$

which is still nonlinear, but only because of the last bracketed term. In the case of a complete market, meaning nonrandom ($\beta = 0$) or perfectly correlated ($|\rho| = 1$) volatility, notice that the nonlinear term disappears and the work done by the Legendre transform is to reveal that the optimization problem is simply a linear pricing problem in disguise. However this complete reduction is not possible with stochastic volatility. Nonetheless, the transform has done some work in isolating the nonlinearity due to the fact that volatility is not traded.

6 Asymptotics for Utility Maximization

In this section, we study the effect of uncertain volatility on the optimal strategies for state-dependent utility maximization. We take advantage of *fast mean-reversion* and use a singular perturbation analysis to find a relatively simple trading strategy that approximates the optimal one. The analysis in Section 5 still has not yielded a way to compute the optimal strategy short of solving one of the nonlinear PDEs (16) or (18) which have three spatial dimensions. One of the benefits of the approach described here is easing of this dimensional burden. Another one is robustness—as we will see we do not need to know all the parameters in the model for the purposes of the approximate strategy.

In the zero-order approximation derived here, two kinds of average (or homogenized) volatilities emerge: $\bar{\sigma} := \langle f^2 \rangle^{1/2}$ and $\sigma_\star := \langle f^{-2} \rangle^{-1/2}$, where $\langle \cdot \rangle$ denotes a particular averaging procedure described below.

6.1 Singular Perturbation Analysis

We introduce the scaling

$$\begin{aligned} \alpha &= 1/\varepsilon \\ \beta &= \sqrt{2}\,\nu/\sqrt{\varepsilon} \end{aligned}$$

where $0 < \varepsilon << 1$ and ν is fixed, to model fast mean-reversion (clustering) in market volatility. Recall that α measures the characteristic speed of mean-reversion of (Y_t) and ν^2 is the variance of the long-run distribution, measuring the typical size of the fluctuations of the volatility-driving process.

Then $g = g^\varepsilon$ satisfies the PDE (18), which we re-write with the new notation as:

$$\left(\frac{1}{\varepsilon}\mathcal{L}_0 + \frac{1}{\sqrt{\varepsilon}}\mathcal{L}_1 + \mathcal{L}_2 \right) g^\varepsilon + \frac{\nu^2}{\varepsilon}(1-\rho^2)\mathrm{NL}^\varepsilon = 0, \tag{19}$$

where we define

$$\begin{aligned} \mathcal{L}_0 &= \nu^2 \frac{\partial^2}{\partial y^2} + (m-y)\frac{\partial}{\partial y} && (20) \\ \mathcal{L}_1 &= \sqrt{2}\,\rho\nu \left(f(y)x\frac{\partial^2}{\partial x \partial y} - \frac{\mu}{f(y)} z \frac{\partial^2}{\partial y \partial z} - \frac{\mu}{f(y)}\frac{\partial}{\partial y} \right) && (21) \\ \mathcal{L}_2 &= \frac{\partial}{\partial t} + \frac{1}{2} f(y)^2 x^2 \frac{\partial^2}{\partial x^2} + \frac{\mu^2}{f(y)^2}\left(\frac{1}{2} z^2 \frac{\partial^2}{\partial z^2} + z\frac{\partial}{\partial z} \right) - \mu x z \frac{\partial^2}{\partial x \partial z}, && (22) \end{aligned}$$

and the nonlinear part is

$$\mathrm{NL}^\varepsilon = -\frac{\partial}{\partial z}\left(\frac{(g_y^\varepsilon)^2}{g_z^\varepsilon} \right) = -\left[\frac{2 g_y^\varepsilon g_{yz}^\varepsilon}{g_z^\varepsilon} - \left(\frac{g_y^\varepsilon}{g_z^\varepsilon} \right)^2 g_{zz}^\varepsilon \right].$$

Notice that $\mathcal{L}_0$ is the usual (scaled) OU generator and $\mathcal{L}_1$ takes derivatives in y and kills functions that do not depend on y.

The approach is now to think of the actual market (and our optimization problem) as embedded in a family of similar problems parametrized by (small) values of ε. For $\varepsilon = 0$, volatility is mean-reverting "infinitely fast" and can be replaced by some average as far as expectations are concerned. However two different averages are needed for different facets of the optimal strategy. This principal approximation may be sufficient for many purposes. It can be improved by perturbation or expansion around $\varepsilon = 0$, but the cost is greater reliance on model specification. We refer to [12] for details.

6.1.1 Expansion

We look for an expansion

$$g^{\varepsilon}(t,x,y,z) = g^{(0)}(t,x,y,z) + \sqrt{\varepsilon} g^{(1)}(t,x,y,z) + \varepsilon g^{(2)}(t,x,y,z) + \cdots$$

for small ε. There is a similar expansion for the other dual function $\hat{H}$ and the two are related by $\hat{H}_z = -g$.

6.1.2 Term of Order $1/\varepsilon$

Inserting the expansion for g and comparing terms of order $1/\varepsilon$ gives

$$\nu^2 g^{(0)}_{yy} + (m-y) g^{(0)}_y + \nu^2 (1-\rho^2) \frac{\partial}{\partial z} \left(\frac{(g^{(0)}_y)^2}{g^{(0)}_z} \right) = 0.$$

An obvious solution to this is any smooth function $g^{(0)}$ that does not depend on y. It is an interesting and crucial fact that the solution $g^{(0)}$ that we are looking for is of this kind, i.e. does not depend on y. To see this, it is easier to work with the PDE for the zero-order term $\hat{H}^{(0)}$ of $\hat{H}$ (recall that $g = -\hat{H}_z$ so $g^{(0)} = -\hat{H}^{(0)}_z$). This reads

$$\nu^2 \hat{H}^{(0)}_{yy} + (m-y) \hat{H}^{(0)}_y - \nu^2 (1-\rho^2) \frac{(\hat{H}^{(0)}_{yz})^2}{\hat{H}^{(0)}_{zz}} = 0. \tag{23}$$

Denoting $u = \hat{H}_y$ and using the concavity property $\hat{H}_{zz} > 0$ we may write (23) as an ordinary differential inequality:

$$\nu^2 u_y + (m-y) u \geq 0. \tag{24}$$

Let us define

$$\Phi(y) = \frac{1}{\sqrt{2\pi\nu^2}} e^{-(y-m)^2/2\nu^2}, \tag{25}$$

the density of the $\mathcal{N}(m,\nu^2)$ distribution. Using this, we can re-write (24) as

$$\frac{1}{\Phi(y)} \frac{\partial}{\partial y} \left(\Phi(y) u(y,z) \right) \geq 0.$$

By integrating we get

$$\begin{aligned} u(y,z) &\geq u(m,z) e^{\frac{(y-m)^2}{2\nu^2}} \quad \text{for } y \geq m \\ u(y,z) &\leq u(m,z) e^{\frac{(y-m)^2}{2\nu^2}} \quad \text{for } y \leq m. \end{aligned}$$

We conclude that $u(m,z)=0$, because otherwise $u(m,y)$ would grow too fast as $y\to\pm\infty$. This implies that any solution to (23) is independent of y. This is because if we specify the value q of any solution at the point $y=m$,

$$\hat{H}^{(0)}(m,z)=q,$$

then the constant function

$$\tilde{H}(y,z)\equiv q \qquad \text{for all } y$$

is also a solution to (23) with $\tilde{H}_y(m,z)=0$. From uniqueness of solutions it follows that $\hat{H}^{(0)}\equiv\tilde{H}$ and so $\hat{H}^{(0)}$ and $g^{(0)}=-\hat{H}^{(0)}_z$ do not depend on y, so

$$g^{(0)}=g^{(0)}(t,x,z).$$

6.1.3 Term of Order $1/\sqrt{\varepsilon}$

At the order $1/\sqrt{\varepsilon}$,

$$\mathcal{L}_1 g^{(0)}+\mathcal{L}_0 g^{(1)}=0,$$

which implies $g^{(1)}$ also does not depend on y because $\mathcal{L}_1 g^{(0)}=0$ and $\mathcal{L}_0$ has null space spanned by constants. This is a general property of generators of "nice" ergodic processes like the OU.

Since both $g^{(0)}$ and $g^{(1)}$ do not depend on y, the nonlinear term is effectively

$$\mathrm{NL}^{\varepsilon}=\mathcal{O}(\varepsilon^2)$$

and only contributes to the asymptotics when we compare order ε and higher. This fact is crucial to the further analysis. We will go as far as order one (ε^0) here, so we are dealing essentially with linear asymptotics (except for the very first equation).

6.1.4 Zero-order Term

At order 1, we have

$$\mathcal{L}_0 g^{(2)}+\mathcal{L}_1 g^{(1)}+\mathcal{L}_2 g^{(0)}=0.$$

The middle term is zero because $g^{(1)}$ does not depend on y. We have a Poisson equation (in y) for $g^{(2)}$. The solvability condition is that $\mathcal{L}_2 g^{(0)}$ must be centered with respect to the invariant distribution of the OU process (Y_t) (equivalently, orthogonal to the null space of the adjoint of $\mathcal{L}_0$, the Fredholm alternative). Therefore

$$\langle \mathcal{L}_2 g^{(0)}\rangle=\langle\mathcal{L}_2\rangle g^{(0)}=0, \tag{26}$$

where $\langle\cdot\rangle$ denotes the averaging

$$\langle\Psi\rangle=\int\Psi\Phi=\frac{1}{\sqrt{2\pi\nu^2}}\int_{-\infty}^{\infty}\Psi(y)\,e^{-\frac{(m-y)^2}{2\nu^2}}\,dy,$$

that is, the average with respect to the $\mathcal{N}(m,\nu^2)$ distribution, the invariant or long-run distribution of the OU process (Y_t).

The averaged operator is

$$\langle \mathcal{L}_2 \rangle = \frac{\partial}{\partial t} + \frac{1}{2}\bar{\sigma}^2 x^2 \frac{\partial^2}{\partial x^2} + \frac{\mu^2}{\sigma_\star^2}\left(\frac{1}{2}z^2\frac{\partial^2}{\partial z^2} + z\frac{\partial}{\partial z}\right) - \mu x z \frac{\partial^2}{\partial x \partial z},$$

where we define

$$\bar{\sigma}^2 = \langle f^2 \rangle \tag{27}$$

$$\frac{1}{\sigma_\star^2} = \langle \frac{1}{f^2} \rangle. \tag{28}$$

The terminal condition is

$$g^{(0)}(T, x, z) = G(x, z).$$

The problem for $g^{(0)}(t, x, z)$ is similar to the constant volatility problem (11), with two important differences:

1. The zero-order approximation $g^{(0)}$ depends not just on the usual long-run average historical volatility $\bar{\sigma}$, but also on the *harmonically-averaged volatility* $\sigma_\star$ defined by (28). Thus the asymptotic approximation of the optimal strategy will depend on estimating this unusual volatility too. By Jensen's inequality, $\sigma_\star \leq \bar{\sigma}$ and equality holds if and only if volatility is constant a.s.

2. The "homogenized" operator $\langle \mathcal{L}_2 \rangle$ is nondegenerate even though $\mathcal{L}_2$ is degenerate. As a result, $g^{(0)}(t, x, z)$ is the expectation of a functional of a *two-dimensional* Brownian motion, unlike the expectation in (12). In other words, the zero-order asymptotic approximation is not simply the complete market problem with constant averaged volatility.

The consequences of this are discussed in Section 6.2.

6.1.5 Zero-order Strategy

The optimal zero-order strategy is given by

$$\pi^\star = \left(x\frac{\partial}{\partial x} - \frac{\mu}{f(y)^2} z \frac{\partial}{\partial z}\right) g^{(0)}. \tag{29}$$

Notice that this *does* depend on tracking volatility $f(y)$ even though the corrected minimum expected loss does not (to zero-order). However, it does not depend on the difficult to estimate parameters of a volatility model, namely $\alpha,\ \beta,\ \rho,\ m$, nor on a specific choice of model f.

6.1.6 Interpretation and Estimation of $\sigma_\star$

One possible way to estimate $\sigma_\star$ is to use the Taylor expansion

$$\begin{aligned}\frac{1}{\sigma_t^2} &= \frac{1}{\bar{\sigma}^2 + (\sigma_t^2 - \bar{\sigma}^2)} \\ &\approx \frac{1}{\bar{\sigma}^2}\left[1 - \left(\frac{\sigma_t^2 - \bar{\sigma}^2}{\bar{\sigma}^2}\right) + \left(\frac{\sigma_t^2 - \bar{\sigma}^2}{\bar{\sigma}^2}\right)^2\right],\end{aligned}$$

so that

$$\langle \frac{1}{\sigma_t^2} \rangle \approx \frac{\langle \sigma_t^4 \rangle}{\bar{\sigma}^6}.$$

The long-run volatility $\bar{\sigma}$ and the fourth-moment $\langle \sigma_t^4 \rangle$ can be estimated stably from the second and fourth-moments of high-frequency historical returns. There is no need to specify a volatility model $f(Y_t)$.

This rough estimator also shows that

$$\frac{\bar{\sigma}^2}{\sigma_\star^2} \approx \frac{\langle \sigma_t^4 \rangle}{\bar{\sigma}^4}, \tag{30}$$

and so $\bar{\sigma}/\sigma_\star$ is a measure of excess kurtosis.

As an indication of the type of volatility observed in the market, we plot in Figure 6 the time-series of the VIX index produced by the CBOE, which is a measure of the daily volatility of US equity indices. It is produced from option implied volatilities and is only used here to gauge the order of volatility fluctuations. Taking this as a realization of volatility, we estimate the two average volatilities:

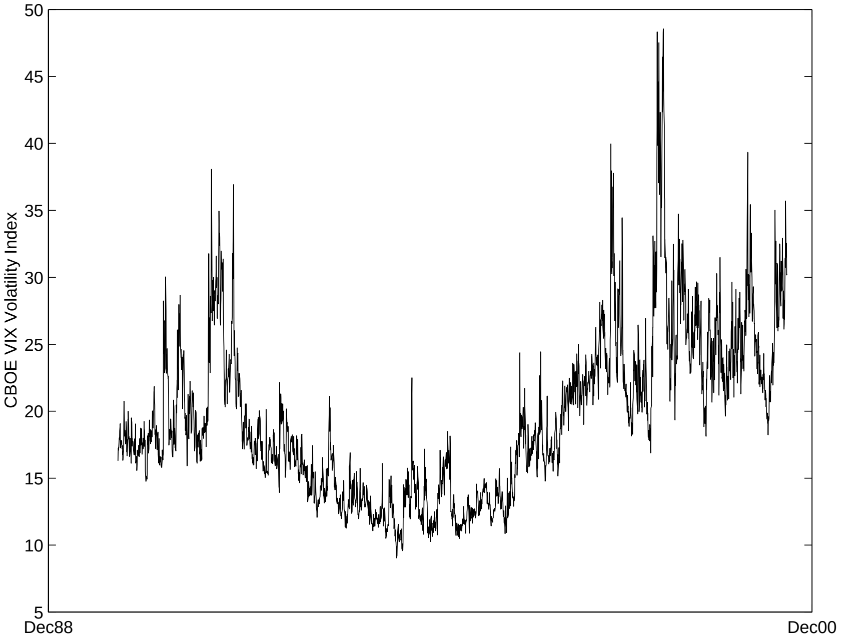


Figure 6: *CBOE VIX volatility index Dec 1988-Dec 2000 (in percentage units).*

$$\bar{\sigma} \approx 0.220 \qquad \sigma_\star \approx 0.177,$$

which gap describes wide fluctuations in this measure of volatility, as is seen from the picture.

6.2 Explicit Computations

In order to implement the zero order strategy we need to know the function $g^{(0)}(t,x,z)$ and its derivatives with respect to x and z. While these three quantities could be computed numerically from a discretization of the linear PDE (26), it is quicker to use the Feynman-Kac representation (see e.g. [17]) of the solution. We refer to [12] for further details on the following argument (in the case of partial hedging). It is possible to write

$$g^{(0)}(t,x,z) = I\!E\left\{g\left(t,x,za;\bar{\sigma}\right)\right\}, \tag{31}$$

where $g(t,x,z;\bar{\sigma})$ is the solution to the constant volatility ($=\bar{\sigma}$) problem (13) and the expectation is over η drawn from a standard $\mathcal{N}(0,1)$ distribution. Here

$$a = a(\eta) = e^{\frac{\mu}{\sigma_\star}\sqrt{1-\hat{\rho}^2}\sqrt{\tau}\eta+\frac{1}{2}\frac{\mu^2}{\sigma_\star^2}(1-\hat{\rho}^2)\tau},$$

where the parameter $\hat{\rho}$ is given by

$$\hat{\rho} := \frac{\sigma_\star}{\bar{\sigma}}, \tag{32}$$

and satisfies $0 < \hat{\rho} \leq 1$ by Jensen's inequality. This parameter can be thought of as a correlation coefficient and is a measure of how much volatility is fluctuating. In particular, $\hat{\rho} = 1$ if and only if volatility is constant a.s. In the case $f(y) = e^y$ in (15), known as the expOU model, for example, the approximation (30) turns out to be exact and $\hat{\rho} = e^{-2\nu^2}$, where $\nu^2 = \beta^2/2\alpha$ is a measure of the *size* of volatility fluctuations.

We can also use (31) to compute derivatives of $g^{(0)}$. For instance, we get

$$\begin{aligned} g_x^{(0)}(t,x,z) &= I\!E\left\{g_x\left(t,x,za;\bar{\sigma}\right)\right\} \\ g_z^{(0)}(t,x,z) &= I\!E\left\{a g_z\left(t,x,za;\bar{\sigma}\right)\right\}, \end{aligned}$$

which further yields a formula for the optimal zero-order strategy using (29). The main point of the formulas above is that if we have an efficient way of computing the constant-volatility solution (such as in the case of partial hedging of a call option), then we can fairly easily compute the zero-order approximation through (31). Specifically, we may compute $g^{(0)}(t,x,z)$ (and its derivatives) on a grid in z as weighted averages of the constant volatility solutions $g(t,x,z;\bar{\sigma})$ on a corresponding grid.

7 Simulations

In this section we present numerical results that illustrate the approximation described above. We focus on partial hedging of a call option. In particular, we demonstrate the performance of the zero-order strategy suggested by the asymptotics and the Föllmer-Leukert strategy derived in [7] for a constant volatility market (see Section 3.4). However, in the latter at every rebalancing of the portfolio, we update the level of the volatility as would be done in market practice. We assume throughout the level of volatility is perfectly observable. A reasonable approximation of current volatility can usually be obtained each day (for example from near-to-expiration implied volatilities or by averaging intraday returns), but it is harder to assemble a reasonable time-series including intraday volatility levels on which to perform detailed model calibration.

7.1 Simulation Details

For the simulation, we discretized the explicit model

$$\begin{aligned}\frac{dX_t}{X_t} &= \mu\, dt + e^{Y_t}\, dW_t \\ dY_t &= \alpha(m - Y_t)\, dt + \nu\sqrt{2\alpha}\left(\rho\, dW_t + \sqrt{1-\rho^2}\, dB_t\right) \\ dV_t &= \pi_t \frac{dX_t}{X_t},\end{aligned}$$

where $r = 0.03, \mu = 0.15 - r$, $K = 100$, $\rho = -0.2$ and (W_t) and (B_t) are independent Brownian motions. In the (Y_t) process, we fixed $\bar{\sigma} = 0.1$ and $\nu = 0.25$ and ran hedging tests for various α. Changing α does not change $\bar{\sigma}$ or $\sigma_\star$ because the invariant density of the OU process does not change if we hold $\bar{\sigma}$ and ν fixed. The characteristic speed of the process varies with α, but not the characteristic size of volatility fluctuations. In this case, with f the exponential function,

$$\sigma_\star = \bar{\sigma}e^{-2\nu^2} = 0.0882.$$

We used initial stock price $X_0 = 100$ (at the money) and initial capital equal to 25% of the Black-Scholes price with constant volatility $\bar{\sigma}$.

To compute the two strategies along the path we did as follows.

- For the Föllmer-Leukert strategy, given t, X_t and V_t we solved numerically the equation $g(t, X_t, z; \bar{\sigma}) = V_t$ for z, where g satisfies (13). The hedging ratio $\Delta = \pi/X$ is given by

$$\Delta_t = g_x(t, X_t, z; \bar{\sigma}) - \frac{\mu}{f(Y_t)^2}\frac{z}{X_t}g_z(t, X_t, z; \bar{\sigma})$$

  and the value of the portfolio was updated using $dV_t = \Delta_t\, dX_t$. Notice that the level of volatility is updated in this hedge as would be done in practice.

- For the zero-order strategy we did as follows. Given t, X_t and V_t we solved the equation $g^{(0)}(t, X_t, z) = V_t$ for z, with $g^{(0)}$ from (31). The hedging ratio was then chosen as

$$\Delta_t = g_x^{(0)}(t, X_t, z) - \frac{\mu}{(\exp(Y_t)^2)}\frac{z}{X_t}g_z^{(0)}(t, X_t, z)$$

  and the value of the portfolio updated using $dV_t = \Delta_t\, dX_t$.

Throughout, we used $p = 1.1$; see Section 1.2.1 for a comment on this choice. The time horizon of the problem is $T = 0.5$ years, so we are hedging a six-month option. We used a time-discretization of 200 even intervals over this period, corresponding roughly to rehedging twice a day. This might be more frequent than in practice, but we wanted to compare the two strategies rather than deal with significant discretization issues.

7.2 Large α Simulation

We first compared the strategies when the volatility *is* fast mean-reverting as assumed by the asymptotics. We set $\alpha = 200$ in annualized units which corresponds to a typical mean-reversion time of about one day. We computed estimates for the expected losses

$$L = I\!E\{(\frac{1}{p}(X_T - K - V_T)^+)^p\}$$

for the two strategies.

With 60000 paths, we got $L_1 = 2.979$ (Föllmer-Leukert) and $L_2 = 2.903$ (zero-order), so the zero-order strategy outperformed the Föllmer-Leukert strategy by approximately 2.6%. This is illustrated in Figure 7. One natural question is whether the improvement in the expected loss comes at significant cost to the variance. From these simulations, the answer is no (for the Föllmer-Leukert strategy, the standard deviation of the loss was 4.46, and it was 4.21 for the zero-order strategy.)

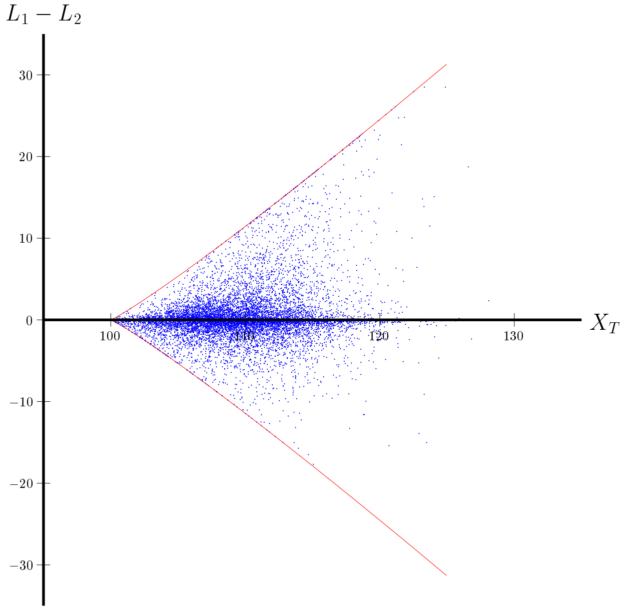


Figure 7: *Difference in losses between Föllmer-Leukert strategy and zero-order strategy as a function of the terminal stock price for* 60, 000 *paths with* $\alpha = 200$.

7.3 Small α Simulation

In some markets, the rate of mean-reversion may not be so rapid. Nonetheless, the zero-order strategy may still have some effect as a crude approximation to the full stochastic volatility strategy. We compared the strategies when the volatility is mean-reverting with $\alpha = 1$ in annualized units which corresponds to a typical mean-reversion time of about one year.

With 60000 paths, we got $L_1 = 3.156$ (Föllmer-Leukert) and $L_2 = 3.111$ (zero-order), so the zero-order strategy outperformed the Föllmer-Leukert strategy by approximately 1.4%. This is illustrated in Figure 8.

The standard deviations were 3.88 for the Föllmer-Leukert strategy and again it was lower (3.68) for the zero-order strategy.

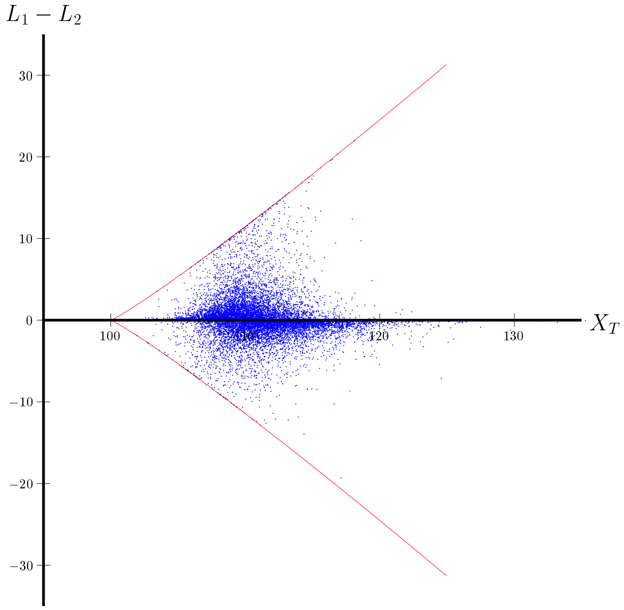


Figure 8: *Difference in losses between Föllmer-Leukert strategy and zero-order strategy as a function of the terminal stock price for* 60, 000 *paths with* $\alpha = 1$*.*

8 Conclusions

In this article we have reviewed some stochastic optimization problems that are the subject of much current research in financial mathematics. These present significant computational challenges, particularly within realistic market models such as those with stochastic volatility that we have focused on here.

Our approach is to use a homogenization approximation exploiting rapid mean-reversion in the stochastic process driving volatility. This leads to a strategy that, as demonstrated by simulation, performs better than the strategy based on assuming volatility is constant (the usual root-mean-square average $\bar{\sigma}$), but with minimal implementation overhead. The key ingredient is the harmonically averaged volatility $\sigma_\star$ which encapsulates the effect of mean-reverting stochastic volatility for these problems. Obviously, the degree of suboptimality of this approximation will be extremely model dependent, but since volatility models and parameters are not specified with much confidence in practice, this robust method is preferable to solving a high-dimensional Bellman equation whose parameters are anyway uncertain.

Additionally, as is often found with homogenization approximations, the zero-order strategy is an improvement *even outside the regime of fast mean-reversion.* This is demonstrated in the small α simulations above.

References

[1] D. Bertsimas, L. Kogan, and A. Lo. Pricing and hedging derivative securities in incomplete markets: an epsilon-arbitrage approach. Technical report, MIT Sloan School,

1999.

[2] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *J. Political Econ.*, 81:637–659, 1973.

[3] J. Cvitanić and I. Karatzas. On dynamic measures of risk. *Finance and Stochastics*, 3(4), 1999.

[4] M. Davis, V. Panas, and T. Zariphopoulou. European pricing with transaction costs. *SIAM J. Control and Optim.*, 31:470–93, 1993.

[5] D. Duffie and H. Richardson. Mean-variance hedging in continuous time. *Annals of Applied Probability*, 1:1–15, 1991.

[6] N. ElKaroui, M. Jeanblanc-Picque, and V. Lacoste. Optimal portfolio management with American capital guarantee. Technical report, CMAP, Ecole Polytechnique, November 2000. Preliminary version.

[7] H. Főllmer and P. Leukert. Efficient hedging: Cost versus shortfall risk. *Finance and Stochastics*, 4(2), 2000.

[8] W. H. Fleming and H. M. Soner. *Controlled Markov Processes and Viscosity Solutions.* Springer-Verlag, 1993.

[9] J.-P. Fouque, G. Papanicolaou, K. R. Sircar, and K. Solna. Mean-Reversion of S&P 500 Volatility. *Submitted*, 2000.

[10] J.-P. Fouque, G. Papanicolaou, and K.R. Sircar. *Derivatives in Financial Markets with Stochastic Volatility.* Cambridge University Press, 2000.

[11] J. Hull and A. White. The Pricing of Options on Assets with Stochastic Volatilities. *J. Finance*, XLII(2):281–300, June 1987.

[12] M. Jonsson and K. R. Sircar. Partial Hedging in a Stochastic Volatility Environment. *Submitted*, 2000.

[13] I. Karatzas and S. Shreve. *Methods of Mathematical Finance.* Springer-Verlag, 1998.

[14] R. C. Merton. Lifetime portfolio selection under uncertainty: the continous-time case. *Rev. Econom. Statist.*, 51:247–257, 1969.

[15] R. C. Merton. Theory of rational option pricing. *Bell Journal of Economics*, 4(1):141–183, Spring 1973.

[16] R. C. Merton. *Continuous-Time Finance.* Blackwell, 1992.

[17] B. Øksendal. *Stochastic Differential Equations, 5th ed.* Springer-Verlag, 1998.

MaxQ Hierarchical RL, value functions, action smoothing

Tom Dietterich
(tgd@cs.orst.edu)

Abstract

1. MAXQ Hierarchical Reinforcement Learning

I could describe the MAXQ formalism, the theoretical results, and our experiments on toy problems, along with some of the extensions that other people have been pursuing.

Reference:

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. Journal of Artificial Intelligence Research, 13, 227--303.

2. Value function approximation using support vector machines and regression trees.

The main goal here is to get a practical and scalable value function approximation methodology that can be applied to nearly-deterministic, episodic MDPs defined with discrete action spaces and continuous state spaces. I can describe the methods we have developed and experiments we have done, but perhaps more importantly the problems we have encountered.

References:

Dietterich, T. G. and Wang, X. (2002). Batch Value Function Approximation via Support Vectors. Accepted for publication in Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.) Advances in Neural Information Processing Systems 14. Cambridge, MA: MIT Press.

Wang, X. and Dietterich, T. G. (2002). Stabilizing Value Function Approximation with the BFBP algorithm. Accepted for publication Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.) Advances in Neural Information Processing Systems 14. Cambridge, MA: MIT Press.

3. Applying RL to resolve resource tradeoffs in mobile robotics / probability smoothing for action refinement.

This work concerns a mobile robot navigating in an unknown environment via landmarks. The robot camera is a valuable resource that must be shared between

navigation (identifying and detecting landmarks) and obstacle avoidance. We have applied model-based RL (with a hand-designed discrete state space and discrete action set) to learn to resolve this tradeoff. On a high-fidelity simulation, a hand-coded policy reached the goal only 25% of the time, whereas a learned policy was more than 80% successful.

An interesting issue arose here: When we were designing the action space, we came up with many, similar actions that could be made available to the robot. We have developed a method that we call probability smoothing for smoothly pooling the training examples of several similar actions to estimate the transition model for a single action. This behaves as a form of action refinement in which, initially, all of the similar actions are treated as equivalent, but as more experience is obtained, the models of the different actions are refined.

References:

Busquets, Lopez de Mantaras, Sierra, and Dietterich (2001). Reinforcement learning for landmark-based robot navigation. Technical Report. Artificial Intelligence Research Institute (IIIA), Barcelona, Spain.

Busquets, Lopez de Mantaras, Sierra, and Dietterich (2002). Reinforcement learning for landmark-based robot navigation (abstract). To appear, Autonomous Agents and Multi-Agent Systems, 2002.

Dietterich, Busquest, Sierra, Lopez de Mantaras (2002). Action refinement in reinforcement learning by probability smoothing. Submitted: International Conference on Machine Learning (ICML-2002).

Journal of Artificial Intelligence Research 13 (2000) 227-303 Submitted 11/99; published 11/00

Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition

Thomas G. Dietterich TGD@CS.ORST.EDU
Department of Computer Science, Oregon State University
Corvallis, OR 97331

Abstract

This paper presents a new approach to hierarchical reinforcement learning based on decomposing the target Markov decision process (MDP) into a hierarchy of smaller MDPs and decomposing the value function of the target MDP into an additive combination of the value functions of the smaller MDPs. The decomposition, known as the MAXQ decomposition, has both a procedural semantics—as a subroutine hierarchy—and a declarative semantics—as a representation of the value function of a hierarchical policy. MAXQ unifies and extends previous work on hierarchical reinforcement learning by Singh, Kaelbling, and Dayan and Hinton. It is based on the assumption that the programmer can identify useful subgoals and define subtasks that achieve these subgoals. By defining such subgoals, the programmer constrains the set of policies that need to be considered during reinforcement learning. The MAXQ value function decomposition can represent the value function of any policy that is consistent with the given hierarchy. The decomposition also creates opportunities to exploit state abstractions, so that individual MDPs within the hierarchy can ignore large parts of the state space. This is important for the practical application of the method. This paper defines the MAXQ hierarchy, proves formal results on its representational power, and establishes five conditions for the safe use of state abstractions. The paper presents an online model-free learning algorithm, MAXQ-Q, and proves that it converges with probability 1 to a kind of locally-optimal policy known as a recursively optimal policy, even in the presence of the five kinds of state abstraction. The paper evaluates the MAXQ representation and MAXQ-Q through a series of experiments in three domains and shows experimentally that MAXQ-Q (with state abstractions) converges to a recursively optimal policy much faster than flat Q learning. The fact that MAXQ learns a representation of the value function has an important benefit: it makes it possible to compute and execute an improved, non-hierarchical policy via a procedure similar to the policy improvement step of policy iteration. The paper demonstrates the effectiveness of this non-hierarchical execution experimentally. Finally, the paper concludes with a comparison to related work and a discussion of the design tradeoffs in hierarchical reinforcement learning.

1. Introduction

The area of Reinforcement Learning (Bertsekas & Tsitsiklis, 1996; Sutton & Barto, 1998) studies methods by which an agent can learn optimal or near-optimal plans by interacting directly with the external environment. The basic methods in reinforcement learning are based on the classical dynamic programming algorithms that were developed in the late 1950s (Bellman, 1957; Howard, 1960). However, reinforcement learning methods offer two important advantages over classical dynamic programming. First, the methods are online. This permits them to focus their attention on the parts of the state space that are important and to ignore the rest of the space. Second, the methods can employ function approximation algorithms (e.g., neural networks) to represent their knowledge. This allows them to generalize across the state space so that the learning time scales much better.

Despite recent advances in reinforcement learning, there are still many shortcomings. The biggest of these is the lack of a fully satisfactory method for incorporating hierarchies into reinforcement learning algorithms. Research in classical planning has shown that hierarchical methods such as hierarchical task networks (Currie & Tate, 1991), macro actions (Fikes, Hart, & Nilsson, 1972; Korf, 1985), and state abstraction methods (Sacerdoti, 1974; Knoblock, 1990) can provide exponential reductions in the computational cost of finding good plans. However, all of the basic algorithms for probabilistic planning and reinforcement learning are "flat" methods—they treat the state space as one huge flat search space. This means that the paths from the start state to the goal state are very long, and the length of these paths determines the cost of learning and planning, because information about future rewards must be propagated backward along these paths.

Many researchers (Singh, 1992; Lin, 1993; Kaelbling, 1993; Dayan & Hinton, 1993; Hauskrecht, et al., 1998; Parr & Russell, 1998; Sutton, Precup, & Singh, 1998) have experimented with different methods of hierarchical reinforcement learning and hierarchical probabilistic planning. This research has explored many different points in the design space of hierarchical methods, but several of these systems were designed for specific situations. We lack crisp definitions of the main approaches and a clear understanding of the relative merits of the different methods.

This paper formalizes and clarifies one approach and attempts to understand how it compares with the other techniques. The approach, called the MAXQ method, provides a hierarchical decomposition of the given reinforcement learning problem into a set of subproblems. It simultaneously provides a decomposition of the value function for the given problem into a set of value functions for the subproblems. Hence, it has both a declarative semantics (as a value function decomposition) and a procedural semantics (as a subroutine hierarchy).

The decomposition into subproblems has many advantages. First, policies learned in subproblems can be shared (reused) for multiple parent tasks. Second, the value functions learned in subproblems can be shared, so when the subproblem is reused in a new task, learning of the overall value function for the new task is accelerated. Third, if state abstractions can be applied, then the overall value function can be represented compactly as the sum of separate terms that each depends on only a subset of the state variables. This more compact representation of the value function will require less data to learn, and hence, learning will be faster.

Previous research shows that there are several important design decisions that must be made when constructing a hierarchical reinforcement learning system. To provide an overview of the results in this paper, let us review these issues and see how the MAXQ method approaches each of them.

The first issue is how to specify subtasks. Hierarchical reinforcement learning involves breaking the target Markov decision problem into a hierarchy of subproblems or subtasks. There are three general approaches to defining these subtasks. One approach is to define each subtask in terms of a fixed policy that is provided by the programmer (or that has been learned in some separate process). The "option" method of Sutton, Precup, and Singh (1998) takes this approach. The second approach is to define each subtask in terms of a non-deterministic finite-state controller. The Hierarchy of Abstract Machines (HAM) method of Parr and Russell (1998) takes this approach. This method permits the programmer to provide a "partial policy" that constrains the set of permitted actions at each point, but does not specify a complete policy for each subtask. The third approach is to define each subtask in terms of a termination predicate and a local reward function. These define what it means for the subtask to be completed and what the final reward should be for completing the subtask. The MAXQ method described in this paper follows this approach, building upon previous work by Singh (1992), Kaelbling (1993), Dayan and Hinton (1993), and Dean and Lin (1995).

An advantage of the "option" and partial policy approaches is that the subtask can be defined in terms of an amount of effort or a course of action rather than in terms of achieving a particular goal condition. However, the "option" approach (at least in the simple form described in this paper), requires the programmer to provide complete policies for the subtasks, which can be a difficult programming task in real-world problems. On the other hand, the termination predicate method requires the programmer to guess the relative desirability of the different states in which the subtask might terminate. This can also be difficult, although Dean and Lin show how these guesses can be revised automatically by the learning algorithm.

A potential drawback of all hierarchical methods is that the learned policy may be suboptimal. The hierarchy constrains the set of possible policies that can be considered. If these constraints are poorly chosen, the resulting policy will be suboptimal. Nonetheless, the learning algorithms that have been developed for the "option" and partial policy approaches guarantee that the learned policy will be the best possible policy consistent with these constraints.

The termination predicate method suffers from an additional source of suboptimality. The learning algorithm described in this paper converges to a form of local optimality that we call *recursive optimality*. This means that the policy of each subtask is locally optimal given the policies of its children. But there might exist better hierarchical policies where the policy for a subtask must be locally suboptimal so that the overall policy is optimal. For example, a subtask of buying milk might be performed suboptimally (at a more distant store) because the larger problem also involves buying film (at the same store). This problem can be avoided by careful definition of termination predicates and local reward functions, but this is an added burden on the programmer. (It is interesting to note that this problem of recursive optimality has not been noticed previously. This is because previous work focused on subtasks with a single terminal state, and in such cases, the problem does not arise.)

The second design issue is whether to employ state abstractions within subtasks. A subtask employs state abstraction if it ignores some aspects of the state of the environment. For example, in many robot navigation problems, choices about what route to take to reach a goal location are independent of what the robot is currently carrying. With few exceptions, state abstraction has not been explored previously. We will see that the MAXQ method creates many opportunities to exploit state abstraction, and that these abstractions can have a huge impact in accelerating learning. We will also see that there is an important design tradeoff: the successful use of state abstraction requires that subtasks be defined in terms of termination predicates rather than using the option or partial policy methods. This is why the MAXQ method must employ termination predicates, despite the problems that this can create.

The third design issue concerns the non-hierarchical "execution" of a learned hierarchical policy. Kaelbling (1993) was the first to point out that a value function learned from a hierarchical policy could be evaluated incrementally to yield a potentially much better non-hierarchical policy. Dietterich (1998) and Sutton, et al. (1999) generalized this to show how arbitrary subroutines could be executed non-hierarchically to yield improved policies. However, in order to support this non-hierarchical execution, extra learning is required. Ordinarily, in hierarchical reinforcement learning, the only states where learning is required at the higher levels of the hierarchy are states where one or more of the subroutines could terminate (plus all possible initial states). But to support non-hierarchical execution, learning is required in all states (and at all levels of the hierarchy). In general, this requires additional exploration as well as additional computation and memory. As a consequence of the hierarchical decomposition of the value function, the MAXQ method is able to support either form of execution, and we will see that there are many problems where the improvement from non-hierarchical execution is worth the added cost.

The fourth and final issue is what form of learning algorithm to employ. An important advantage of reinforcement learning algorithms is that they typically operate online. However, finding online algorithms that work for general hierarchical reinforcement learning has been difficult, particularly within the termination predicate family of methods. Singh's method relied on each subtask having a unique terminal state; Kaelbling employed a mix of online and batch algorithms to train her hierarchy; and work within the "options" framework usually assumes that the policies for the subproblems are given and do not need to be learned at all. The best previous online algorithms are the HAMQ Q learning algorithm of Parr and Russell (for the partial policy method) and the Feudal Q algorithm of Dayan and Hinton. Unfortunately, the HAMQ method requires "flattening" the hierarchy, and this has several undesirable consequences. The Feudal Q algorithm is tailored to a specific kind of problem, and it does not converge to any well-defined optimal policy.

In this paper, we present a general algorithm, called MAXQ-Q, for fully-online learning of a hierarchical value function. This algorithm enables all subtasks within the hierarchy to be learned simultaneously and online. We show experimentally and theoretically that the algorithm converges to a recursively optimal policy. We also show that it is substantially faster than "flat" (i.e., non-hierarchical) Q learning when state abstractions are employed.

The remainder of this paper is organized as follows. After introducing our notation in Section 2, we define the MAXQ value function decomposition in Section 3 and illustrate it with a simple example Markov decision problem. Section 4 presents an analytically tractable version of the MAXQ-Q learning algorithm called the MAXQ-0 algorithm and proves its convergence to a recursively optimal policy. It then shows how to extend MAXQ-0 to produce the MAXQ-Q algorithm, and shows how to extend the theorem similarly. Section 5 takes up the issue of state abstraction and formalizes a series of five conditions under which state abstractions can be safely incorporated into the MAXQ representation. State abstraction can give rise to a hierarchical credit assignment problem, and the paper briefly discusses one solution to this problem. Finally, Section 7 presents experiments with three example domains. These experiments give some idea of the generality of the MAXQ representation. They also provide results on the relative importance of temporal and state abstractions and on the importance of non-hierarchical execution. The paper concludes with further discussion of the design issues that were briefly described above, and in particular, it addresses the tradeoff between the method of defining subtasks (via termination predicates) and the ability to exploit state abstractions.

Some readers may be disappointed that MAXQ provides no way of learning the structure of the hierarchy. Our philosophy in developing MAXQ (which we share with other reinforcement learning researchers, notably Parr and Russell) has been to draw inspiration from the development of Belief Networks (Pearl, 1988). Belief networks were first introduced as a formalism in which the knowledge engineer would describe the structure of the networks and domain experts would provide the necessary probability estimates. Subsequently, methods were developed for learning the probability values directly from observational data. Most recently, several methods have been developed for learning the structure of the belief networks from data, so that the dependence on the knowledge engineer is reduced.

In this paper, we will likewise require that the programmer provide the structure of the hierarchy. The programmer will also need to make several important design decisions. We will see below that a MAXQ representation is very much like a computer program, and we will rely on the programmer to design each of the modules and indicate the permissible ways in which the modules can invoke each other. Our learning algorithms will fill in "implementations" of each module in such a way that the overall program will work well. We believe that this approach will provide a practical tool for solving large real-world MDPs. We also believe that it will help us understand the structure of hierarchical learning algorithms. It is our hope that subsequent research will be able to automate most of the work that we are currently requiring the programmer to do.

2. Formal Definitions

We begin by introducing definitions for Markov Decision Problems and Semi-Markov Decision Problems.

2.1 Markov Decision Problems

We employ the standard definition for Markov Decision Problems (also known as Markov decision processes). In this paper, we restrict our attention to situations in which an agent is interacting with a fully-observable stochastic environment. This situation can be modeled as a Markov Decision Problem (MDP) $\langle S, A, P, R, P_0 \rangle$ defined as follows:

- S: the finite set of states of the environment. At each point in time, the agent can observe the complete state of the environment.
- A: a finite set of actions. Technically, the set of available actions depends on the current state s, but we will suppress this dependence in our notation.
- P: When an action $a \in A$ is performed, the environment makes a probabilistic transition from its current state s to a resulting state s' according to the probability distribution $P(s'|s, a)$.
- R: Similarly, when action a is performed and the environment makes its transition from s to s', the agent receives a real-valued (possibly stochastic) reward r whose expected value is $R(s'|s, a)$. To simplify the notation, it is customary to treat this reward as being given at the time that action a is initiated, even though it may in general depend on s' as well as on s and a.
- P_0: The starting state distribution. When the MDP is initialized, it is in state s with probability $P_0(s)$.

A *policy*, π, is a mapping from states to actions that tells what action $a = \pi(s)$ to perform when the environment is in state s.

We will consider two settings: episodic and infinite-horizon.

In the episodic setting, all rewards are finite and there is at least one zero-cost absorbing terminal state. An absorbing terminal state is a state in which all actions lead back to the same state with probability 1 and zero reward. For technical reasons, we will only consider problems where all deterministic policies are "proper"—that is, all deterministic policies have a non-zero probability of reaching a terminal state when started in an arbitrary state. (We believe this condition can be relaxed, but we have not verified this formally.) In the episodic setting, the goal of the agent is to find a policy that maximizes the expected cumulative reward. In the special case where all rewards are non-positive, these problems are referred to as stochastic shortest path problems, because the rewards can be viewed as costs (i.e., lengths), and the policy attempts to move the agent along the path of minimum expected cost.

In the infinite horizon setting, all rewards are also finite. In addition, there is a discount factor γ, and the agent's goal is to find a policy that minimizes the infinite discounted sum of future rewards.

The value function V^π for policy π is a function that tells, for each state s, what the expected cumulative reward will be of executing policy π starting in state s. Let r_t be a random variable that tells the reward that the agent receives at time step t while following policy π. We can define the value function in the episodic setting as

$$V^\pi(s) = E\left\{r_t + r_{t+1} + r_{t+2} + \cdots | s_t = s, \pi\right\}.$$

In the discounted setting, the value function is

$$V^\pi(s) = E\left\{ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \middle| s_t = s, \pi \right\}.$$

We can see that this equation reduces to the previous one when $\gamma = 1$. However, in infinite-horizon MDPs this sum may not converge when $\gamma = 1$.

The value function satisfies the Bellman equation for a fixed policy:

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s)) \left[R(s'|s, \pi(s)) + \gamma V^{\pi}(s') \right].$$

The quantity on the right-hand side is called the *backed-up value* of performing action a in state s. For each possible successor state s', it computes the reward that would be received and the value of the resulting state and then weights those according to the probability of ending up in s'.

The optimal value function V^* is the value function that simultaneously maximizes the expected cumulative reward in all states $s \in S$. Bellman (1957) proved that it is the unique solution to what is now known as the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s, a) \left[R(s'|s, a) + \gamma V^*(s') \right]. \tag{1}$$

There may be many optimal policies that achieve this value. Any policy that chooses a in s to achieve the maximum on the right-hand side of this equation is an optimal policy. We will denote an optimal policy by π^*. Note that all optimal policies are "greedy" with respect to the backed-up value of the available actions.

Closely related to the value function is the so-called *action-value function*, or Q function (Watkins, 1989). This function, $Q^{\pi}(s, a)$, gives the expected cumulative reward of performing action a in state s and then following policy π thereafter. The Q function also satisfies a Bellman equation:

$$Q^{\pi}(s, a) = \sum_{s'} P(s'|s, a) \left[R(s'|s, a) + \gamma Q^{\pi}(s', \pi(s')) \right].$$

The optimal action-value function is written $Q^*(s, a)$, and it satisfies the equation

$$Q^*(s, a) = \sum_{s'} P(s'|s, a) \left[R(s'|s, a) + \gamma \max_{a'} Q^*(s', a') \right]. \tag{2}$$

Note that any policy that is greedy with respect to Q^* is an optimal policy. There may be many such optimal policies—they differ only in how they break ties between actions with identical Q^* values.

An *action order*, denoted ω, is a total order over the actions within an MDP. That is, ω is an anti-symmetric, transitive relation such that $\omega(a_1, a_2)$ is true iff a_1 is strictly preferred to a_2. An *ordered greedy policy*, π_ω is a greedy policy that breaks ties using ω. For example, suppose that the two best actions at state s are a_1 and a_2, that $Q(s, a_1) = Q(s, a_2)$, and that $\omega(a_1, a_2)$. Then the ordered greedy policy π_ω will choose a_1: $\pi_\omega(s) = a_1$. Note that although there may be many optimal policies for a given MDP, the ordered greedy policy, π^*_ω, is unique.

2.2 Semi-Markov Decision Processes

In order to introduce and prove some of the properties of the MAXQ decomposition, we need to consider a simple generalization of MDPs—the semi-Markov decision process.

A discrete-time *semi-Markov Decision Process* (SMDP) is a generalization of the Markov Decision Process in which the actions can take a variable amount of time to complete. In particular, let the random variable N denote the number of time steps that action a takes when it is executed in state s. We can extend the state transition probability function to be the joint distribution of the result states s' and the number of time steps N when action a is performed in state s: $P(s', N|s, a)$. Similarly, the expected reward can be changed to be $R(s', N|s, a)$.[1]

It is straightforward to modify the Bellman equation to define the value function for a fixed policy π as

$$V^\pi(s) = \sum_{s',N} P(s', N|s, \pi(s)) \left[R(s', N|s, \pi(s)) + \gamma^N V^\pi(s') \right].$$

The only change is that the expected value on the right-hand side is taken with respect to both s' and N, and γ is raised to the power N to reflect the variable amount of time that may elapse while executing action a.

Note that because expectation is a linear operator, we can write each of these Bellman equations as the sum of the expected reward for performing action a and the expected value of the resulting state s'. For example, we can rewrite the equation above as

$$V^\pi(s) = \overline{R}(s, \pi(s)) + \sum_{s',N} P(s', N|s, \pi(s)) \gamma^N V^\pi(s'). \tag{3}$$

where $\overline{R}(s, \pi(s))$ is the expected reward of performing action $\pi(s)$ in state s, and the expectation is taken with respect to s' and N.

All of the results given in this paper can be generalized to apply to discrete-time semi-Markov Decision Processes. A consequence of this is that whenever this paper talks of executing a primitive action, it could just as easily talk of executing a hand-coded open-loop "subroutine". These subroutines would not be learned, and nor could their execution be interrupted as discussed below in Section 6. But in many applications (e.g., robot control with limited sensors), open-loop controllers can be very useful (e.g., to hide partial-observability). For an example, see Kalmár, Szepesvári, and A. Lörincz (1998).

Note that for the episodic case, there is no difference between a MDP and a Semi-Markov Decision Process, because the discount factor γ is 1, and therefore neither the optimal policy nor the optimal value function depend on the amount of time each action takes.

2.3 Reinforcement Learning Algorithms

A reinforcement learning algorithm is an algorithm that tries to construct an optimal policy for an unknown MDP. The algorithm is given access to the unknown MDP via the following

[1] This formalization is slightly different from the standard formulation of SMDPs, which separates $P(s'|s, a)$ and $F(t|s, a)$, where F is the cumulative distribution function for the probability that a will terminate in t time units, and t is real-valued rather than integer-valued. In our case, it is important to consider the joint distribution of s' and N, but we do not need to consider actions with arbitrary real-valued durations.

reinforcement learning protocol. At each time step t, the algorithm is told the current state s of the MDP and the set of actions $A(s) \subseteq A$ that are executable in that state. The algorithm chooses an action $a \in A(s)$, and the MDP executes this action (which causes it to move to state s') and returns a real-valued reward r. If s is an absorbing terminal state, the set of actions $A(s)$ contains only the special action reset, which causes the MDP to move to one of its initial states, drawn according to P_0.

In this paper, we will make use of two well-known learning algorithms: Q learning (Watkins, 1989; Watkins & Dayan, 1992) and SARSA(0) (Rummery & Niranjan, 1994). We will apply these algorithms to the case where the action value function $Q(s,a)$ is represented as a table with one entry for each pair of state and action. Every entry of the table is initialized arbitrarily.

In Q learning, after the algorithm has observed s, chosen a, received r, and observed s', it performs the following update:

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma \max_{a'} Q_{t-1}(s',a')],$$

where α_t is a learning rate parameter.

Jaakkola, Jordan and Singh (1994) and Bertsekas and Tsitsiklis (1996) prove that if the agent follows an "exploration policy" that tries every action in every state infinitely often and if

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t = \infty \quad \text{and} \quad \lim_{\mathrm{T}\to\infty} \sum_{\mathrm{t}=1}^{\mathrm{T}} \alpha_{\mathrm{t}}^2 < \infty \tag{4}$$

then Q_t converges to the optimal action-value function Q^* with probability 1. Their proof holds in both settings discussed in this paper (episodic and infinite-horizon).

The SARSA(0) algorithm is very similar. After observing s, choosing a, observing r, observing s', and choosing a', the algorithm performs the following update:

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma Q_{t-1}(s',a')],$$

where α_t is a learning rate parameter. The key difference is that the Q value of the chosen action a', $Q(s',a')$, appears on the right-hand side in the place where Q learning uses the Q value of the best action. Singh, et al. (1998) provide two important convergence results: First, if a fixed policy π is employed to choose actions, SARSA(0) will converge to the value function of that policy provided α_t decreases according to Equations (4). Second, if a so-called GLIE policy is employed to choose actions, SARSA(0) will converge to the value function of the optimal policy, provided again that α_t decreases according to Equations (4). A GLIE policy is defined as follows:

Definition 1 *A GLIE (Greedy in the Limit with Infinite Exploration) policy is any policy satisfying*

1. *Each action is executed infinitely often in every state that is visited infinitely often.*

2. *In the limit, the policy is greedy with respect to the Q-value function with probability 1.*

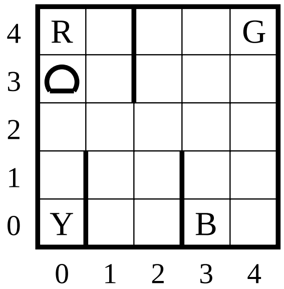


Figure 1: The Taxi Domain.

3. The MAXQ Value Function Decomposition

At the center of the MAXQ method for hierarchical reinforcement learning is the MAXQ value function decomposition. MAXQ describes how to decompose the overall value function for a policy into a collection of value functions for individual subtasks (and subsubtasks, recursively).

3.1 A Motivating Example

To make the discussion concrete, let us consider the following simple example. Figure 1 shows a 5-by-5 grid world inhabited by a taxi agent. There are four specially-designated locations in this world, marked as R(ed), B(lue), G(reen), and Y(ellow). The taxi problem is episodic. In each episode, the taxi starts in a randomly-chosen square. There is a passenger at one of the four locations (chosen randomly), and that passenger wishes to be transported to one of the four locations (also chosen randomly). The taxi must go to the passenger's location (the "source"), pick up the passenger, go to the destination location (the "destination"), and put down the passenger there. (To keep things uniform, the taxi must pick up and drop off the passenger even if he/she is already located at the destination!) The episode ends when the passenger is deposited at the destination location.

There are six primitive actions in this domain: (a) four navigation actions that move the taxi one square North, South, East, or West, (b) a Pickup action, and (c) a Putdown action. There is a reward of -1 for each action and an additional reward of $+20$ for successfully delivering the passenger. There is a reward of -10 if the taxi attempts to execute the Putdown or Pickup actions illegally. If a navigation action would cause the taxi to hit a wall, the action is a no-op, and there is only the usual reward of -1.

To simplify the examples throughout this section, we will make the six primitive actions deterministic. Later, we will make the actions stochastic in order to create a greater challenge for our learning algorithms.

We seek a policy that maximizes the total reward per episode. There are 500 possible states: 25 squares, 5 locations for the passenger (counting the four starting locations and the taxi), and 4 destinations.

This task has a simple hierarchical structure in which there are two main sub-tasks: Get the passenger and Deliver the passenger. Each of these subtasks in turn involves the subtask of navigating to one of the four locations and then performing a Pickup or Putdown action.

This task illustrates the need to support temporal abstraction, state abstraction, and subtask sharing. The temporal abstraction is obvious—for example, the process of navigating to the passenger's location and picking up the passenger is a temporally extended action that can take different numbers of steps to complete depending on the distance to the target. The top level policy (get passenger; deliver passenger) can be expressed very simply if these temporal abstractions can be employed.

The need for state abstraction is perhaps less obvious. Consider the subtask of getting the passenger. While this subtask is being solved, the destination of the passenger is completely irrelevant—it cannot affect any of the nagivation or pickup decisions. Perhaps more importantly, when navigating to a target location (either the source or destination location of the passenger), only the target location is important. The fact that in some cases the taxi is carrying the passenger and in other cases it is not is irrelevant.

Finally, support for subtask sharing is critical. If the system could learn how to solve the navigation subtask once, then the solution could be shared by both the "Get the passenger" and "Deliver the passenger" subtasks. We will show below that the MAXQ method provides a value function representation and learning algorithm that supports temporal abstraction, state abstraction, and subtask sharing.

To construct a MAXQ decomposition for the taxi problem, we must identify a set of individual subtasks that we believe will be important for solving the overall task. In this case, let us define the following four tasks:

- Navigate(t). In this subtask, the goal is to move the taxi from its current location to one of the four target locations, which will be indicated by the formal parameter t.

- Get. In this subtask, the goal is to move the taxi from its current location to the passenger's current location and pick up the passenger.

- Put. The goal of this subtask is to move the taxi from the current location to the passenger's destination location and drop off the passenger.

- Root. This is the whole taxi task.

Each of these subtasks is defined by a subgoal, and each subtask terminates when the subgoal is achieved.

After defining these subtasks, we must indicate for each subtask which other subtasks or primitive actions it should employ to reach its goal. For example, the Navigate(t) subtask should use the four primitive actions North, South, East, and West. The Get subtask should use the Navigate subtask and the Pickup primitive action, and so on.

All of this information can be summarized by a directed acyclic graph called the *task graph*, which is shown in Figure 2. In this graph, each node corresponds to a subtask or a primitive action, and each edge corresponds to a potential way in which one subtask can "call" one of its child tasks. The notation $formal/actual$ (e.g., $t/source$) tells how a formal parameter is to be bound to an actual parameter.

Now suppose that for each of these subtasks, we write a policy (e.g., as a computer program) to achieve the subtask. We will refer to the policy for a subtask as a "subroutine", and we can view the parent subroutine as invoking the child subroutine via ordinary

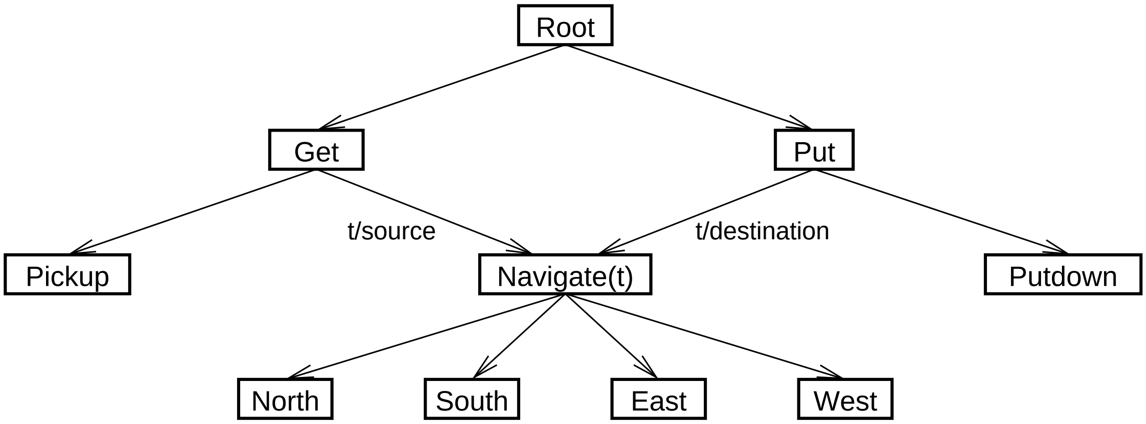


Figure 2: A task graph for the Taxi problem.

subroutine-call-and-return semantics. If we have a policy for each subtask, then this gives us an overall policy for the Taxi MDP. The Root subtask executes its policy by calling subroutines that are policies for the Get and Put subtasks. The Get policy calls subroutines for the Navigate(t) subtask and the Pickup primitive action. And so on. We will call this collection of policies a *hierarchical policy*. In a hierarchical policy, each subroutine executes until it enters a terminal state for its subtask.

3.2 Definitions

Let us formalize the discussion so far.

The MAXQ decomposition takes a given MDP M and decomposes it into a finite set of subtasks $\{M_0, M_1, \ldots, M_n\}$ with the convention that M_0 is the root subtask (i.e., solving M_0 solves the entire original MDP M).

Definition 2 *An* unparameterized subtask *is a three-tuple, $\langle T_i, A_i, \tilde{R}_i \rangle$, defined as follows:*

1. *T_i is a* termination predicate *that partitions S into a set of active states, S_i, and a set of terminal states, T_i. The policy for subtask M_i can only be executed if the current state s is in S_i. If, at any time that subtask M_i is being executed, the MDP enters a state in T_i, then M_i terminates immediately (even if it is still executing a subtask, see below).*

2. *A_i is a* set of actions *that can be performed to achieve subtask M_i. These actions can either be primitive actions from A, the set of primitive actions for the MDP, or they can be other subtasks, which we will denote by their indexes i. We will refer to these actions as the "children" of subtask i. The sets A_i define a directed graph over the subtasks $M_0, \ldots, M_n$, and this graph may not contain any cycles. Stated another way, no subtask can invoke itself recursively either directly or indirectly.*

   If a child subtask M_j has formal parameters, then this is interpreted as if the subtask occurred multiple times in A_i, with one occurrence for each possible tuple of actual *values that could be bound to the formal parameters. The set of actions A_i may differ from one state to another and from one set of actual parameter values to another, so technically, A_i is a function of s and the actual parameters. However, we will suppress this dependence in our notation.*

3. *$\tilde{R}_i(s')$ is the* pseudo-reward function*, which specifies a (deterministic)* pseudo-reward *for each transition to a terminal state $s' \in T_i$. This pseudo-reward tells how desirable each of the terminal states is for this subtask. It is typically employed to give goal terminal states a pseudo-reward of 0 and any non-goal terminal states a negative reward. By definition, the pseudo-reward $\tilde{R}_i(s)$ is also zero for all* non-terminal *states s. The pseudo-reward is only used during learning, so it will not be mentioned further until Section 4.*

Each primitive action a from M is a primitive subtask in the MAXQ decomposition such that a is always executable, it always terminates immediately after execution, and its pseudo-reward function is uniformly zero.

If a subtask has formal parameters, then each possible binding of actual values to the formal parameters specifies a distinct subtask. We can think of the values of the formal parameters as being part of the "name" of the subtask. In practice, of course, we implement a parameterized subtask by parameterizing the various components of the task. If b specifies the actual parameter values for task M_i, then we can define a parameterized termination predicate $T_i(s, b)$ and a parameterized pseudo-reward function $\tilde{R}_i(s', b)$. To simplify notation in the rest of the paper, we will usually omit these parameter bindings. However, it should be noted that if a parameter of a subtask takes on a large number of possible values, this is equivalent to creating a large number of different subtasks, each of which will need to be learned. It will also create a large number of candidate actions for the parent task, which will make the learning problem more difficult for the parent task as well.

Definition 3 *A* hierarchical policy*, π, is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0, \ldots, \pi_n\}$.*

Each subtask policy π_i takes a state and returns the name of a primitive action to execute or the name of a subroutine (and bindings for its formal parameters) to invoke. In the terminology of Sutton, Precup, and Singh (1998), a subtask policy is a deterministic "option", and its probability of terminating in state s (which they denote by $\beta(s)$) is 0 if $s \in S_i$, and 1 if $s \in T_i$.

In a parameterized task, the policy must be parameterized as well so that π takes a state and the bindings of formal parameters and returns a chosen action and the bindings (if any) of its formal parameters.

Table 1 gives a pseudo-code description of the procedure for executing a hierarchical policy. The hierarchical policy is executed using a stack discipline, similar to ordinary programming languages. Let K_t denote the contents of the pushdown stack at time t. When a subroutine is invoked, its name and actual parameters are pushed onto the stack. When a subroutine terminates, its name and actual parameters are popped off the stack. Notice (line 16) that if *any* subroutine on the stack terminates, then all subroutines below

Table 1: Pseudo-Code for Execution of a Hierarchical Policy.

```
Procedure ExecuteHierarchicalPolicy($\pi$)
$s_t$ is the state of the world at time $t$
$K_t$ is the state of the execution stack at time $t$

Let $t = 0$; $K_t$ = empty stack; observe $s_t$
push $(0, nil)$ onto stack $K_t$ (invoke the root task with no parameters)

repeat
    while $top(K_t)$ is not a primitive action
        Let $(i, f_i) := top(K_t)$, where
            $i$ is the name of the "current" subroutine, and
            $f_i$ gives the parameter bindings for $i$
        Let $(a, f_a) := \pi_i(s, f_i)$, where
            $a$ is the action and $f_a$ gives the parameter bindings chosen by policy $\pi_i$
        push $(a, f_a)$ onto the stack $K_t$
    end // while

    Let $(a, nil) := pop(K_t)$ be the primitive action on the top of the stack.
    Execute primitive action $a$, observe $s_{t+1}$, and receive reward $R(s_{t+1}|s_t, a)$

    If any subtask on $K_t$ is terminated in $s_{t+1}$ then
        Let $M'$ be the terminated subtask that is highest (closest to the root) on the stack.
        while $top(K_t) \neq M'$ do $pop(K_t)$
        $pop(K_t)$

    $K_{t+1} := K_t$ is the resulting execution stack.
until $K_{t+1}$ is empty

end ExecuteHierarchicalPolicy
```

it are immediately aborted, and control returns to the subroutine that had invoked the terminated subroutine.

It is sometimes useful to think of the contents of the stack as being an additional part of the state space for the problem. Hence, a hierarchical policy implicitly defines a mapping from the current state s_t and current stack contents K_t to a primitive action a. This action is executed, and this yields a resulting state s_{t+1} and a resulting stack contents K_{t+1}. Because of the added state information in the stack, the hierarchical policy is non-Markovian with respect to the original MDP.

Because a hierarchical policy maps from states s and stack contents K to actions, the value function for a hierarchical policy must assign values to combinations of states s and stack contents K.

Definition 4 *A* hierarchical value function*, denoted $V^\pi(\langle s, K\rangle)$, gives the expected cumulative reward of following the hierarchical policy π starting in state s with stack contents K.*

This hierarchical value function is exactly what is learned by Ron Parr's (1998b) HAMQ algorithm, which we will discuss below. However, in this paper, we will focus on learning only the *projected value functions* of each of the subtasks $M_0, \ldots, M_n$ in the hierarchy.

Definition 5 *The* projected value function *of hierarchical policy π on subtask M_i, denoted $V^\pi(i,s)$, is the expected cumulative reward of executing π_i (and the policies of all descendents of M_i) starting in state s until M_i terminates.*

The purpose of the MAXQ value function decomposition is to decompose $V(0,s)$ (the projected value function of the root task) in terms of the projected value functions $V(i,s)$ of all of the subtasks in the MAXQ decomposition.

3.3 Decomposition of the Projected Value Function

Now that we have defined a hierarchical policy and its projected value function, we can show how that value function can be decomposed hierarchically. The decomposition is based on the following theorem:

Theorem 1 *Given a task graph over tasks $M_0, \ldots, M_n$ and a hierarchical policy π, each subtask M_i defines a semi-Markov decision process with states S_i, actions A_i, probability transition function $P_i^\pi(s', N|s,a)$, and* expected *reward function $\overline{R}(s,a) = V^\pi(a,s)$, where $V^\pi(a,s)$ is the projected value function for child task M_a in state s. If a is a primitive action, $V^\pi(a,s)$ is defined as the expected immediate reward of executing a in s: $V^\pi(a,s) = \sum_{s'} P(s'|s,a)R(s'|s,a)$.*

Proof: Consider all of the subroutines that are descendents of task M_i in the task graph. Because all of these subroutines are executing fixed policies (specified by hierarchical policy π), the probability transition function $P_i^\pi(s', N|s,a)$ is a well defined, stationary distribution for each child subroutine a. The set of states S_i and the set of actions A_i are obvious. The interesting part of this theorem is the fact that the expected reward function $\overline{R}(s,a)$ of the SMDP is the projected value function of the child task M_a.

To see this, let us write out the value of $V^\pi(i,s)$:

$$V^\pi(i,s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, \pi\} \tag{5}$$

This sum continues until the subroutine for task M_i enters a state in T_i.

Now let us suppose that the first action chosen by π_i is a subroutine a. This subroutine is invoked, and it executes for a number of steps N and terminates in state s' according to $P_i^\pi(s', N|s,a)$. We can rewrite Equation (5) as

$$V^\pi(i,s) = E\left\{ \sum_{u=0}^{N-1} \gamma^u r_{t+u} \; + \; \sum_{u=N}^{\infty} \gamma^u r_{t+u} \right| \left. s_t = s, \pi \right\} \tag{6}$$

The first summation on the right-hand side of Equation (6) is the discounted sum of rewards for executing subroutine a starting in state s until it terminates, in other words, it is $V^\pi(a,s)$, the projected value function for the child task M_a. The second term on the right-hand side of the equation is the value of s' for the current task i, $V^\pi(i,s')$, discounted by γ^N, where s' is the current state when subroutine a terminates. We can write this in the form of a Bellman equation:

$$V^\pi(i,s) = V^\pi(\pi_i(s), s) + \sum_{s',N} P_i^\pi(s', N|s, \pi_i(s)) \gamma^N V^\pi(i,s') \tag{7}$$

This has the same form as Equation (3), which is the Bellman equation for an SMDP, where the first term is the expected reward $\overline{R}(s, \pi(s))$. **Q.E.D.**

To obtain a hierarchical decomposition of the projected value function, let us switch to the action-value (or Q) representation. First, we need to extend the Q notation to handle the task hierarchy. Let $Q^\pi(i, s, a)$ be the expected cumulative reward for subtask M_i of performing action a in state s and then following hierarchical policy π until subtask M_i terminates. Action a may be either a primitive action or a child subtask. With this notation, we can re-state Equation (7) as follows:

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s', N} P_i^\pi(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')), \tag{8}$$

The right-most term in this equation is the expected discounted reward of *completing* task M_i after executing action a in state s. This term only depends on i, s, and a, because the summation marginalizes away the dependence on s' and N. Let us define $C^\pi(i, s, a)$ to be equal to this term:

Definition 6 *The* completion function, $C^\pi(i, s, a)$, *is the expected discounted cumulative reward of* completing *subtask M_i after invoking the subroutine for subtask M_a in state s. The reward is discounted back to the point in time where a begins execution.*

$$C^\pi(i, s, a) = \sum_{s', N} P_i^\pi(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')) \tag{9}$$

With this definition, we can express the Q function recursively as

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a). \tag{10}$$

Finally, we can re-express the definition for $V^\pi(i, s)$ as

$$V^\pi(i, s) = \begin{cases} Q^\pi(i, s, \pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s, i)R(s'|s, i) & \text{if } i \text{ is primitive} \end{cases} \tag{11}$$

We will refer to equations (9), (10), and (11) as the *decomposition equations* for the MAXQ hierarchy under a fixed hierarchical policy π. These equations recursively decompose the projected value function for the root, $V^\pi(0, s)$ into the projected value functions for the individual subtasks, $M_1, \ldots, M_n$ and the individual completion functions $C^\pi(j, s, a)$ for $j = 1, \ldots, n$. The fundamental quantities that must be stored to represent the value function decomposition are just the C values for all non-primitive subtasks and the V values for all primitive actions.

To make it easier for programmers to design and debug MAXQ decompositions, we have developed a graphical representation that we call the *MAXQ graph*. A MAXQ graph for the Taxi domain is shown in Figure 3. The graph contains two kinds of nodes, Max nodes and Q nodes. The Max nodes correspond to the subtasks in the task decomposition—there is one Max node for each primitive action and one Max node for each subtask (including the Root) task. Each primitive Max node i stores the value of $V^\pi(i, s)$. The Q nodes correspond to the actions that are available for each subtask. Each Q node for parent task i, state s

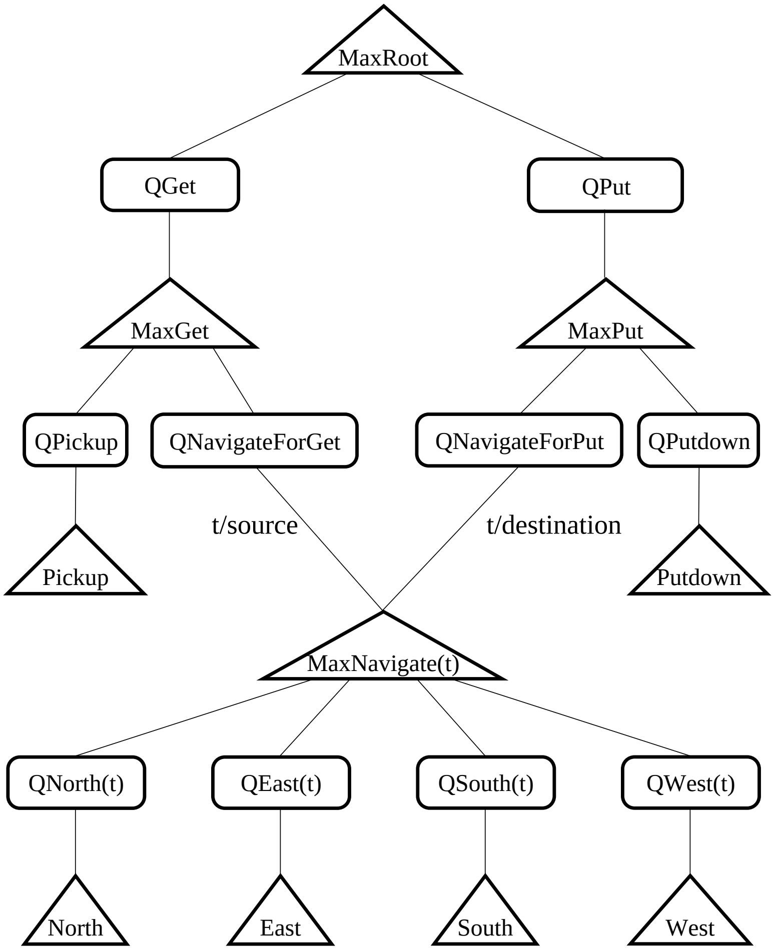


Figure 3: A MAXQ graph for the Taxi Domain.

and subtask a stores the value of $C^{\pi}(i,s,a)$. The children of any node are *unordered*—that is, the order in which they are drawn in Figure 3 does not imply anything about the order in which they will be executed. Indeed, a child action may be executed multiple times before its parent subtask is completed.

In addition to storing information, the Max nodes and Q nodes can be viewed as performing parts of the computation described by the decomposition equations. Specifically, each Max node i can be viewed as computing the projected value function $V^{\pi}(i,s)$ for its subtask. For primitive Max nodes, this information is stored in the node. For composite Max nodes, this information is obtained by "asking" the Q node corresponding to $\pi_i(s)$. Each Q node with parent task i and child task a can be viewed as computing the value of $Q^{\pi}(i,s,a)$. It does this by "asking" its child task a for its projected value function $V^{\pi}(a,s)$ and then adding its completion function $C^{\pi}(i,s,a)$.

As an example, consider the situation shown in Figure 1, which we will denote by s_1. Suppose that the passenger is at R and wishes to go to B. Let the hierarchical policy we are evaluating be an optimal policy denoted by π (we will omit the superscript * to reduce the clutter of the notation). The value of this state under π is 10, because it will cost 1 unit to move the taxi to R, 1 unit to pickup the passenger, 7 units to move the taxi to B, and 1 unit to putdown the passenger, for a total of 10 units (a reward of -10). When the passenger is delivered, the agent gets a reward of +20, so the net value is +10.

Figure 4 shows how the MAXQ hierarchy computes this value. To compute the value $V^\pi(\mathsf{Root}, s_1)$, MaxRoot consults its policy and finds that $\pi_{\mathsf{Root}}(s_1)$ is Get. Hence, it "asks" the Q node, QGet to compute $Q^\pi(\mathsf{Root}, s_1, \mathsf{Get})$. The completion cost for the Root task after performing a Get, $C^\pi(\mathsf{Root}, s_1, \mathsf{Get})$, is 12, because it will cost 8 units to deliver the customer (for a net reward of $20 - 8 = 12$) after completing the Get subtask. However, this is just the reward *after* completing the Get, so it must ask MaxGet to estimate the expected reward of performing the Get itself.

The policy for MaxGet dictates that in s_1, the Navigate subroutine should be invoked with t bound to R, so MaxGet consults the Q node, QNavigateForGet to compute the expected reward. QNavigateForGet knows that after completing the Navigate(R) task, one more action (the Pickup) will be required to complete the Get, so $C^\pi(\mathsf{MaxGet}, s_1, \mathsf{Navigate}(R)) = -1$. It then asks MaxNavigate(R) to compute the expected reward of performing a Navigate to location R.

The policy for MaxNavigate chooses the North action, so MaxNavigate asks QNorth to compute the value. QNorth looks up its completion cost, and finds that $C^\pi(\mathsf{Navigate}, s_1, \mathsf{North})$ is 0 (i.e., the Navigate task will be completed after performing the North action). It consults MaxNorth to determine the expected cost of performing the North action itself. Because MaxNorth is a primitive action, it looks up its expected reward, which is -1.

Now this series of recursive computations can conclude as follows:

- $Q^\pi(\mathsf{Navigate}(R), s_1, \mathsf{North}) = -1 + 0$
- $V^\pi(\mathsf{Navigate}(R), s_1) = -1$
- $Q^\pi(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) = -1 + -1$
  (-1 to perform the Navigate plus -1 to complete the Get.
- $V^\pi(\mathsf{Get}, s_1) = -2$
- $Q^\pi(\mathsf{Root}, s_1, \mathsf{Get}) = -2 + 12$
  (-2 to perform the Get plus 12 to complete the Root task and collect the final reward).

The end result of all of this is that the value of $V^\pi(\mathsf{Root}, s_1)$ is decomposed into a sum of C terms plus the expected reward of the chosen primitive action:

$$\begin{aligned} V^\pi(\mathsf{Root}, s_1) &= V^\pi(\mathsf{North}, s_1) + C^\pi(\mathsf{Navigate}(R), s_1, \mathsf{North}) + \\ &\quad C^\pi(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) + C^\pi(\mathsf{Root}, s_1, \mathsf{Get}) \\ &= -1 + 0 + -1 + 12 \\ &= 10 \end{aligned}$$

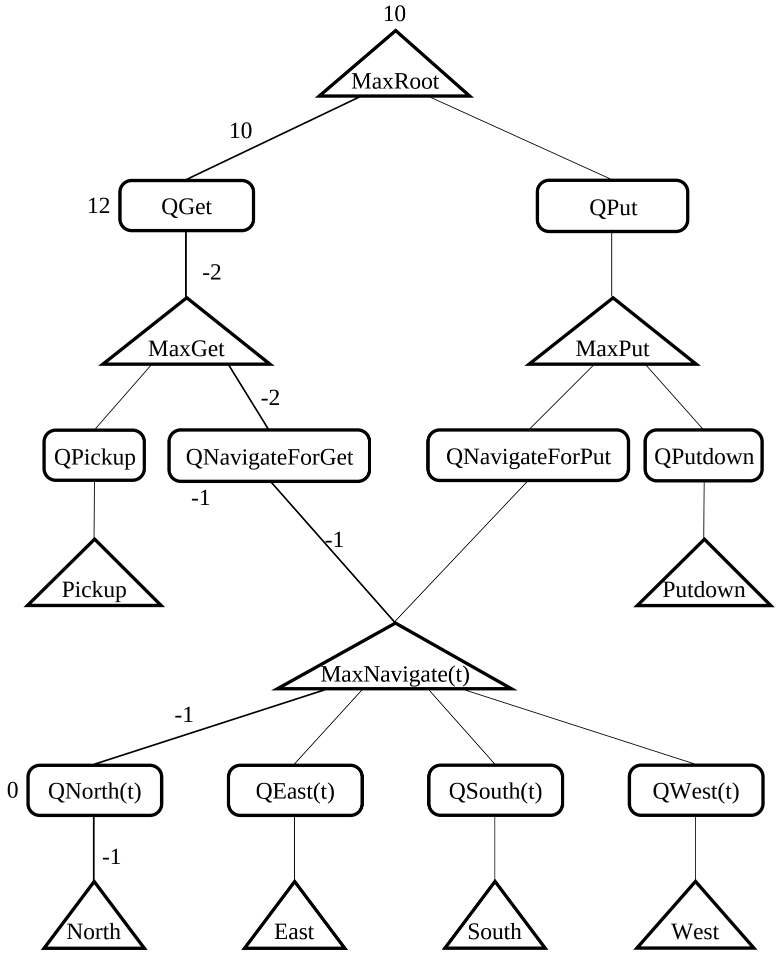


Figure 4: Computing the value of a state using the MAXQ hierarchy. The C value of each Q node is shown to the left of the node. All other numbers show the values being returned up the graph.

In general, the MAXQ value function decomposition has the form

$$V^\pi(0,s) \quad = \quad V^\pi(a_m,s) + C^\pi(a_{m-1},s,a_m) + \ldots + C^\pi(a_1,s,a_2) + C^\pi(0,s,a_1), \qquad (12)$$

where $a_0, a_1, \ldots, a_m$ is the "path" of Max nodes chosen by the hierarchical policy going from the Root down to a primitive leaf node. This is summarized graphically in Figure 5.

We can summarize the presentation of this section by the following theorem:

Theorem 2 *Let $\pi = \{\pi_i; i = 0, \ldots, n\}$ be a hierarchical policy defined for a given MAXQ graph with subtasks $M_0, \ldots, M_n$, and let $i = 0$ be the root node of the graph. Then there exist values for $C^\pi(i,s,a)$ (for internal Max nodes) and $V^\pi(i,s)$ (for primitive, leaf Max*

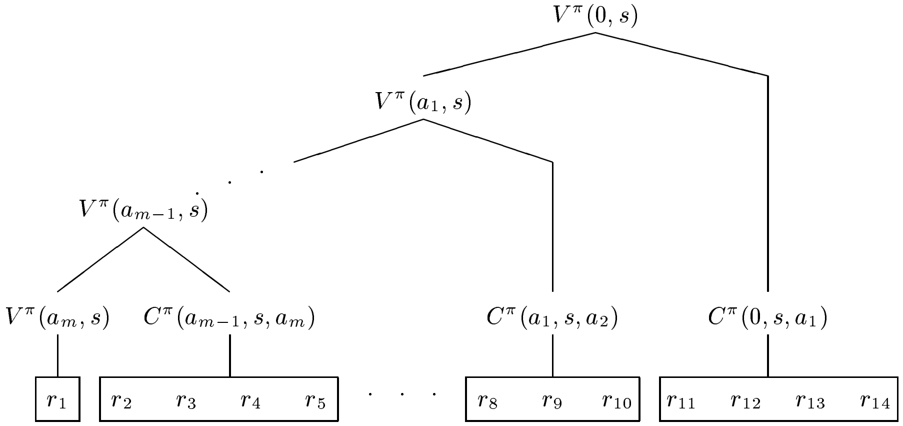


Figure 5: The MAXQ decomposition; $r_1, \ldots, r_{14}$ denote the sequence of rewards received from primitive actions at times $1, \ldots, 14$.

nodes) such that $V^\pi(0, s)$ (as computed by the decomposition equations (9), (10), and (11)) is the expected discounted cumulative reward of following policy π starting in state s.

Proof: The proof is by induction on the number of levels in the task graph. At each level i, we compute values for $C^\pi(i, s, \pi(s))$ (or $V^\pi(i, s)$, if i is primitive) according to the decomposition equations. We can apply the decomposition equations again to compute $Q^\pi(i, s, \pi(s))$ and apply Equation (8) and Theorem 1 to conclude that $Q^\pi(i, s, \pi(s))$ gives the value function for level i. When $i = 0$, we obtain the value function for the entire hierarchical policy. **Q. E. D.**

It is important to note that this representation theorem does not mention the pseudo-reward function, because the pseudo-reward is used only during learning. This theorem captures the *representational power* of the MAXQ decomposition, but it does not address the question of whether there is a learning algorithm that can find a given policy. That is the subject of the next section.

4. A Learning Algorithm for the MAXQ Decomposition

This section presents the central contributions of the paper. First, we discuss what optimality criteria should be employed in hierarchical reinforcement learning. Then we introduce the MAXQ-0 learning algorithm, which can learn value functions (and policies) for MAXQ hierarchies in which there are no pseudo-rewards (i.e., the pseudo-rewards are zero). The central theoretical result of the paper is that MAXQ-0 converges to a recursively optimal policy for the given MAXQ hierarchy. This is followed by a brief discussion of ways of accelerating MAXQ-0 learning. The section concludes with a description of the MAXQ-Q learning algorithm, which handles non-zero pseudo-reward functions.

4.1 Two Kinds of Optimality

In order to develop a learning algorithm for the MAXQ decomposition, we must consider exactly what we are hoping to achieve. Of course, for any MDP M, we would like to find an optimal policy π^*. However, in the MAXQ method (and in hierarchical reinforcement learning in general), the programmer imposes a hierarchy on the problem. This hierarchy constrains the space of possible policies so that it may not be possible to represent the optimal policy or its value function.

In the MAXQ method, the constraints take two forms. First, within a subtask, only some of the possible primitive actions may be permitted. For example, in the taxi task, during a Navigate(t), only the North, South, East, and West actions are available—the Pickup and Putdown actions are not allowed. Second, consider a Max node M_j with child nodes $\{M_{j_1}, \ldots, M_{j_k}\}$. The policy learned for M_j must involve executing the learned policies of these child nodes. When the policy for child node M_{j_i} is executed, it will run until it enters a state in T_{j_i}. Hence, any policy learned for M_j must pass through some subset of these terminal state sets $\{T_{j_1}, \ldots, T_{j_k}\}$.

The HAM method shares these same two constraints and in addition, it imposes a partial policy on each node, so that the policy for any subtask M_i must be a deterministic refinement of the given non-deterministic initial policy for node i.

In the "option" approach, the policy is even further constrained. In this approach, there are only two non-primitive levels in the hierarchy, and the subtasks at the lower level (i.e., whose children are all primitive actions) are given complete policies by the programmer. Hence, any learned policy at the upper level must be constructed by "concatenating" the given lower level policies in some order.

The purpose of imposing these constraints on the policy is to incorporate prior knowledge and thereby reduce the size of the space that must be searched to find a good policy. However, these constraints may make it impossible to learn the optimal policy.

If we can't learn the optimal policy, the next best target would be to learn the best policy that is consistent with (i.e., can be represented by) the given hierarchy.

Definition 7 *A* hierarchically optimal policy *for MDP M is a policy that achieves the highest cumulative reward among all policies consistent with the given hierarchy.*

Parr (1998b) proves that his HAMQ learning algorithm converges with probability 1 to a hierarchically optimal policy. Similarly, given a fixed set of options, Sutton, Precup, and Singh (1998) prove that their SMDP learning algorithm converges to a hierarchically optimal value function. Incidentally, they also show that if the primitive actions are also made available as "trivial" options, then their SMDP method converges to the optimal policy. However, in this case, it is hard to say anything formal about how the options speed the learning process. They may in fact hinder it (Hauskrecht et al., 1998).

Because the MAXQ decomposition can represent the value function of any hierarchical policy, we could easily construct a modified version of the HAMQ algorithm and apply it to learn hierarchically optimal policies for the MAXQ hierarchy. However, we decided to pursue an even weaker form of optimality, for reasons that will become clear as we proceed. This form of optimality is called recursive optimality.

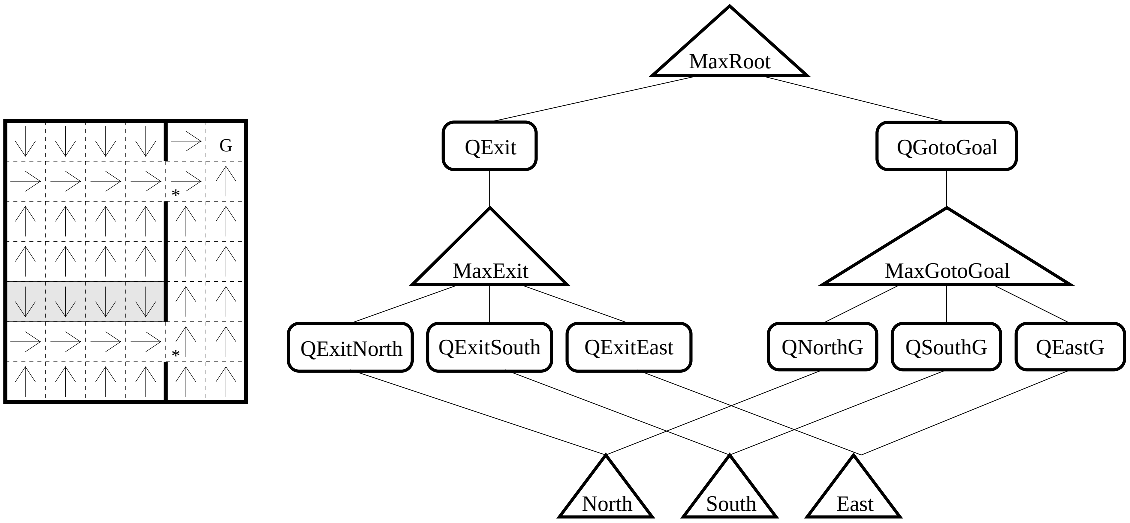


Figure 6: A simple MDP (left) and its associated MAXQ graph (right). The policy shown in the left diagram is recursively optimal but not hierarchically optimal. The shaded cells indicate points where the locally-optimal policy is not globally optimal.

Definition 8 *A* recursively optimal policy *for Markov decision process M with MAXQ decomposition $\{M_0, \ldots, M_k\}$ is a hierarchical policy $\pi = \{\pi_0, \ldots, \pi_k\}$ such that for each subtask M_i, the corresponding policy π_i is optimal for the SMDP defined by the set of states S_i, the set of actions A_i, the state transition probability function $P^\pi(s', N|s, a)$, and the reward function given by the* sum *of the original reward function $R(s'|s, a)$ and the pseudo-reward function $\tilde{R}_i(s')$.*

Note that the state transition probability distribution, $P^\pi(s', N|s, a)$ for subtask M_i is defined by the locally optimal policies $\{\pi_j\}$ of all subtasks that are descendents of M_i in the MAXQ graph. Hence, recursive optimality is a kind of local optimality in which the policy at each node is optimal given the policies of its children.

The reason to seek recursive optimality rather than hierarchical optimality is that recursive optimality makes it possible to solve each subtask without reference to the context in which it is executed. This context-free property makes it easier to share and re-use subtasks. It will also turn out to be essential for the successful use of state abstraction.

Before we proceed to describe our learning algorithm for recursive optimality, let us see how recursive optimality differs from hierarchical optimality.

It is easy to construct examples of policies that are recursively optimal but not hierarchically optimal (and vice versa). Consider the simple maze problem and its associated MAXQ graph shown in Figures 6. Suppose a robot starts somewhere in the left room, and it must reach the goal G in the right room. The robot has three actions, North, South, and East, and these actions are deterministic. The robot receives a reward of -1 for each move. Let us define two subtasks:

- Exit. This task terminates when the robot exits the left room. We can set the pseudo-reward function $\tilde{R}$ to be 0 for the two terminal states (i.e., the two states indicated by *'s).

- GotoGoal. This task terminates when the robot reaches the goal G.

The arrows in Figure 6 show the locally optimal policy within each room. The arrows on the left seek to exit the left room by the shortest path, because this is what we specified when we set the pseudo-reward function to 0. The arrows on the right follow the shortest path to the goal, which is fine. However, the resulting policy is neither hierarchically optimal nor optimal.

There exists a hierarchical policy that would always exit the left room by the upper door. The MAXQ value function decomposition can represent the value function of this policy, but such a policy would not be locally optimal (because, for example, the states in the "shaded" region would not follow the shortest path to a doorway). Hence, this example illustrates both a recursively optimal policy that is not hierarchically optimal and a hierarchically optimal policy that is not recursively optimal.

If we consider for a moment, we can see a way to fix this problem. The value of the upper starred state under the optimal hierarchical policy is -2 and the value of the lower starred state is -6. Hence, if we changed $\tilde{R}$ to have these values (instead of being zero), then the recursively-optimal policy would be hierarchically optimal (and globally optimal). In other words, if the programmer can guess the right values for the terminal states of a subtask, then the recursively optimal policy will be hierarchically optimal.

This basic idea was first pointed out by Dean and Lin (1995). They describe an algorithm that makes initial guesses for the values of these starred states and then updates those guesses based on the computed values of the starred states under the resulting recursively-optimal policy. They proved that this will converge to a hierarchically optimal policy. The drawback of their method is that it requires repeated solution of the resulting hierarchical learning problem, and this does not always yield a speedup over just solving the original, flat problem.

Parr (1998a) proposed an interesting approach that constructs a *set* of different $\tilde{R}$ functions and computes the recursively optimal policy under each of them for each subtask. His method chooses the $\tilde{R}$ functions in such a way that the hierarchically optimal policy can be approximated to any desired degree. Unfortunately, the method is quite expensive, because it relies on solving a series of linear programming problems each of which requires time polynomial in several parameters, including the number of states $|S_i|$ within the subtask.

This discussion suggests that while, in principle, it is possible to learn good values for the pseudo-reward function, in practice, we must rely on the programmer to specify a single pseudo-reward function, $\tilde{R}$, for each subtask. If the programmer wishes to consider a small number of alternative pseudo-reward functions, they can be handled by defining a small number of subtasks that are identical except for their $\tilde{R}$ functions, and permitting the learning algorithm to choose the one that gives the best recursively-optimal policy.

In our experiments, we have employed the following simplified approach to defining $\tilde{R}$. For each subtask M_i, we define two predicates: the termination predicate, T_i, and a goal predicate, G_i. The goal predicate defines a subset of the terminal states that are "goal states", and these have a pseudo-reward of 0. All other terminal states have a fixed constant pseudo-reward (e.g., -100) that is set so that it is always better to terminate in a goal state than in a non-goal state. For the problems on which we have tested the MAXQ method, this worked very well.

In our experiments with MAXQ, we have found that it is easy to make mistakes in defining T_i and G_i. If the goal is not defined carefully, it is easy to create a set of subtasks that lead to infinite looping. For example, consider again the problem in Figure 6. Suppose we permit a fourth action, West, in the MDP and let us define the termination and goal predicates for the right hand room to be satisfied iff either the robot reaches the goal or it exits the room. This is a very natural definition, since it is quite similar to the definition for the left-hand room. However, the resulting locally-optimal policy for this room will attempt to move to the nearest of these three locations: the goal, the upper door, or the lower door. We can easily see that for all but a few states near the goal, the only policies that can be constructed by MaxRoot will loop forever, first trying to leave the left room by entering the right room, and then trying to leave the right room by entering the left room. This problem is easily fixed by defining the goal predicate G_i for the right room to be true if and only if the robot reaches the goal G. But avoiding such "undesired termination" bugs can be hard in more complex domains.

In the worst case, it is possible for the programmer to specify pseudo-rewards such that the recursively optimal policy can be made arbitrarily worse than the hierarchically optimal policy. For example, suppose that we change the original MDP in Figure 6 so that the state immediately to the left of the upper doorway gives a large negative reward $-L$ whenever the robot visits that square. Because rewards everywhere else are -1, the hierarchically-optimal policy exits the room by the lower door. But suppose the programmer has chosen instead to force the robot to exit by the upper door (e.g., by assigning a pseudo-reward of $-10L$ for leaving via the lower door). In this case, the recursively-optimal policy will leave by the upper door and suffer the large $-L$ penalty. By making L arbitrarily large, we can make the difference between the hierarchically-optimal policy and the recursively-optimal policy arbitrarily large.

4.2 The MAXQ-0 Learning Algorithm

Now that we have an understanding of recursively optimal policies, we present two learning algorithms. The first one, called MAXQ-0, applies only in the case when the pseudo-reward function $\tilde{R}$ is always zero. We will first prove its convergence properties and then show how it can be extended to give the second algorithm, MAXQ-Q, which works with general pseudo-reward functions.

Table 2 gives pseudo-code for MAXQ-0. MAXQ-0 is a recursive function that executes the current exploration policy starting at Max node i in state s. It performs actions until it reaches a terminal state, at which point it returns a count of the total number of primitive actions that have been executed. To execute an action, MAXQ-0 calls itself recursively (line 9). When the recursive call returns, it updates the value of the completion function for node i. It uses the count of the number of primitive actions to appropriately discount the value of the resulting state s'. At leaf nodes, MAXQ-0 updates the estimated one-step expected reward, $V(i, s)$. The value $\alpha_t(i)$ is a "learning rate" parameter that should be gradually decreased to zero in the limit.

Table 2: The MAXQ-0 learning algorithm.

```
     function MAXQ-0(MaxNode $i$, State $s$)
1    if $i$ is a primitive MaxNode
2        execute $i$, receive $r$, and observe result state $s'$
3        $V_{t+1}(i,s) := (1-\alpha_t(i)) \cdot V_t(i,s) + \alpha_t(i) \cdot r_t$
4        return 1
5    else
6        let $count = 0$
7        while $T_i(s)$ is false do
8            choose an action $a$ according to the current exploration policy $\pi_x(i,s)$
9            let $N$ = MAXQ-0$(a,s)$ (recursive call)
10           observe result state $s'$
11           $C_{t+1}(i,s,a) := (1-\alpha_t(i)) \cdot C_t(i,s,a) + \alpha_t(i) \cdot \gamma^N V_t(i,s')$
12           $count := count + N$
13           $s := s'$
14           end
15       return $count$
     end MAXQ-0

16   // Main program
17   initialize $V(i,s)$ and $C(i,s,j)$ arbitrarily
18   MAXQ-0(root node 0, starting state $s_0$)
```

There are three things that must be specified in order to make this algorithm description complete.

First, to keep the pseudo-code readable, Table 2 does not show how "ancestor termination" is handled. Recall that after each action, the termination predicates of all of the subroutines on the calling stack are checked. If the termination predicate of any one of these is satisfied, then the calling stack is unwound up to the highest terminated subroutine. In such cases, no C values are updated in any of the subroutines that were interrupted except as follows. If subroutine i had invoked subroutine j, and j's termination condition is satisfied, then subroutine i can update the value of $C(i,s,j)$.

Second, we must specify how to compute $V_t(i,s')$ in line 11, since it is not stored in the Max node. It is computed by the following modified versions of the decomposition equations:

$$V_t(i,s) = \begin{cases} \max_a Q_t(i,s,a) & \text{if } i \text{ is composite} \\ V_t(i,s) & \text{if } i \text{ is primitive} \end{cases} \tag{13}$$

$$Q_t(i,s,a) = V_t(a,s) + C_t(i,s,a). \tag{14}$$

These equations reflect two important changes compared with Equations (10) and (11). First, in Equation (13), $V_t(i,s)$ is defined in terms of the Q value of the *best* action a, rather than of the action chosen by a fixed hierarchical policy. Second, there are no π superscripts, because the current value function, $V_t(i,s)$, is not based on a fixed hierarchical policy π.

To compute $V_t(i,s)$ using these equations, we must perform a complete search of all paths through the MAXQ graph starting at node i and ending at the leaf nodes. Table 3

Table 3: Pseudo-code for Greedy Execution of the MAXQ Graph.

```
function EvaluateMaxNode($i, s$)
    if $i$ is a primitive Max node
        return $\langle V_t(i,s), i \rangle$
    else
        for each $j \in A_i$,
            let $\langle V_t(j,s), a_j \rangle$ = EvaluateMaxNode($j, s$)
        let $j^{hg} = \text{argmax}_j \; V_t(j,s) + C_t(i,s,j)$
        return $\langle V_t(j^{hg}, s), a_{j^{hg}} \rangle$
end // EvaluateMaxNode
```

gives pseudo-code for a recursive function, EvaluateMaxNode, that implements a depth-first search. In addition to returning $V_t(i,s)$, EvaluateMaxNode also returns the action at the leaf node that achieves this value. This information is not needed for MAXQ-0, but it will be useful later when we consider non-hierarchical execution of the learned recursively-optimal policy.

This search can be computationally expensive, and a problem for future research is to develop more efficient methods for computing the best path through the graph. One approach is to perform a best-first search and use bounds on the values within subtrees to prune useless paths through the MAXQ graph. A better approach would be to make the computation incremental, so that when the state of the environment changes, only those nodes whose values have changed as a result of the state change are re-considered. It should be possible to develop an efficient bottom-up method similar to the RETE algorithm (and its successors) that is used in the SOAR architecture (Forgy, 1982; Tambe & Rosenbloom, 1994).

The third thing that must be specified to complete our definition of MAXQ-0 is the exploration policy, π_x. We require that π_x be an ordered GLIE policy.

Definition 9 *An* ordered GLIE policy *is a GLIE policy (Greedy in the Limit with Infinite Exploration) that converges in the limit to an* ordered greedy policy*, which is a greedy policy that imposes an arbitrary fixed order ω on the available actions and breaks ties in favor of the action a that appears earliest in that order.*

We need this property in order to ensure that MAXQ-0 converges to a uniquely-defined recursively optimal policy. A fundamental problem with recursive optimality is that in general, each Max node i will have a choice of many different locally optimal policies given the policies adopted by its descendent nodes. These different locally optimal policies will all achieve the same locally optimal value function, but they can give rise to different probability transition functions $P(s', N|s, i)$. The result will be that the Semi-Markov Decision Problems defined at the next level above node i in the MAXQ graph will differ depending on which of these various locally optimal policies is chosen by node i. These differences may lead to better or worse policies at higher levels of the MAXQ graph, even though they make no difference inside subtask i. In practice, the designer of the MAXQ graph will need to design the pseudo-reward function for subtask i to ensure that all locally optimal policies are equally valuable for the parent subroutine. But to carry out our formal analysis, we will just rely on an arbitrary tie-breaking mechanism.[2] If we establish a fixed ordering over the Max nodes in the MAXQ graph (e.g., a left-to-right depth-first numbering), and break ties in favor of the lowest-numbered action, then this defines a unique policy at each Max node. And consequently, by induction, it defines a unique policy for the entire MAXQ graph. Let us call this policy π_r^*. We will use the r subscript to denote recursively optimal quantities under an ordered greedy policy. Hence, the corresponding value function is V_r^*, and C_r^* and Q_r^* denote the corresponding completion function and action-value function. We now prove that the MAXQ-0 algorithm converges to π_r^*.

Theorem 3 *Let $M = \langle S, A, P, R, P_0 \rangle$ be either an episodic MDP for which all deterministic policies are proper or a discounted infinite horizon MDP with discount factor γ. Let H be a MAXQ graph defined over subtasks $\{M_0, \ldots, M_k\}$ such that the pseudo-reward function $\tilde{R}_i(s')$ is zero for all i and s'. Let $\alpha_t(i) > 0$ be a sequence of constants for each Max node i such that*

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t(i) = \infty \quad \textit{and} \quad \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2(i) < \infty \tag{15}$$

Let $\pi_x(i,s)$ be an ordered GLIE policy at each node i and state s and assume that the immediate rewards are bounded. Then with probability 1, algorithm MAXQ-0 converges to π_r^*, the unique recursively optimal policy for M consistent with H and π_x.

Proof: The proof follows an argument similar to those introduced to prove the convergence of Q learning and $SARSA(0)$ (Bertsekas & Tsitsiklis, 1996; Jaakkola et al., 1994). We will employ the following result from stochastic approximation theory, which we state without proof:

Lemma 1 *(Proposition 4.5 from Bertsekas and Tsitsiklis, 1996) Consider the iteration*

$$r_{t+1}(i) := (1-\alpha_t(i))r_t(i) + \alpha_t(i)((Ur_t)(i) + w_t(i) + u_t(i)).$$

Let $\mathcal{F}_t = \{r_0(i), \ldots, r_t(i), w_0(i), \ldots, w_{t-1}(i), \alpha_0(i), \ldots, \alpha_t(i), \forall i\}$ be the entire history of the iteration.

If

(a) The $\alpha_t(i) \geq 0$ satisfy conditions (15)

(b) For every i and t, the noise terms $w_t(i)$ satisfy $E[w_t(i)|\mathcal{F}_t] = 0$

(c) Given any norm $||\cdot||$ on $\mathcal{R}^n$, there exist constants A and B such that $E[w_t^2(i)|\mathcal{F}_t] \leq A + B||r_t||^2$.

(d) There exists a vector r^*, a positive vector ξ, and a scalar $\beta \in [0,1)$, such that for all t,

$$||Ur_t - r^*||_\xi \leq \beta||r_t - r^*||_\xi$$

[2] Alternatively, we could break ties by using a stochastic policy that chose randomly among the tied actions.

(e) There exists a nonnegative random sequence θ_t that converges to zero with probability 1 and is such that for all t

$$|u_t(i)| \leq \theta_t(||r_t||_\xi + 1)$$

then r_t converges to r^* with probability 1. The notation $||\cdot||_\xi$ denotes a weighted maximum norm

$$||A||_\xi = \max_i \frac{|A(i)|}{\xi(i)}.$$

The structure of the proof of Theorem 3 will be inductive, starting at the leaves of the MAXQ graph and working toward the root. We will employ a different time clock at each node i to count the number of update steps performed by MAXQ-0 at that node. The variable t will always refer to the time clock of the current node i.

To prove the base case for any primitive Max node, we note that line 3 of MAXQ-0 is just the standard stochastic approximation algorithm for computing the expected reward for performing action a in state s, and therefore it converges under the conditions given above.

To prove the recursive case, consider any composite Max node i with child node j. Let $P_t(s', N|s, j)$ be the transition probability distribution for performing child action j in state s at time t (i.e., while following the exploration policy in all descendent nodes of node j). By the inductive assumption, MAXQ-0 applied to j will converge to the (unique) recursively optimal value function $V_r^*(j, s)$ with probability 1. Furthermore, because MAXQ-0 is following an ordered GLIE policy for j and its descendents, they will all converge to executing a greedy policy with respect to their value functions, so $P_t(s', N|s, j)$ will converge to $P_r^*(s', N|s, j)$, the unique transition probability function for executing child j under the locally optimal policy π_r^*. What remains to be shown is that the update assignment for C (line 11 of the MAXQ-0 algorithm) converges to the optimal C_r^* function with probability 1.

To prove this, we will apply Lemma 1. We will identify the x in the lemma with a state-action pair (s, a). The vector r_t will be the completion-cost table $C_t(i, s, a)$ for all s, a and fixed i after t update steps. The vector r^* will be the optimal completion-cost $C_r^*(i, s, a)$ (again, for fixed i). Define the mapping U to be

$$(UC)(i, s, a) = \sum_{s'} P_r^*(s', N|s, a)\gamma^N \left(\max_{a'}[C(i, s', a') + V_r^*(a', s')]\right)$$

This is a C update under the MDP M_i assuming that all descendent value functions, $V_r^*(a, s)$, and transition probabilities, $P_r^*(s', N|s, a)$, have converged.

To apply the lemma, we must first express the C update formula in the form of the update rule in the lemma. Let $\overline{s}$ be the state that results from performing a in state s. Line 11 can be written

$$\begin{aligned} C_{t+1}(i, s, a) &:= (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot \gamma^{\overline{N}} \left(\max_{a'}[C_t(i, \overline{s}, a') + V_t(a', \overline{s})]\right) \\ &:= (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot [(UC_t)(i, s, a) + w_t(i, s, a) + u_t(i, s, a)] \end{aligned}$$

where

$$\begin{aligned} w_t(i,s,a) &= \gamma^{\overline{N}} \left(\max_{a'} [C_t(i,\overline{s},a') + V_t(a',\overline{s})] \right) - \\ & \sum_{s',N} P_t(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_t(a',s')] \right) \\ u_t(i,s,a) &= \sum_{s',N} P_t(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_t(a',s')] \right) - \\ & \sum_{s',N} P_r^*(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_r^*(a',s')] \right) \end{aligned}$$

Here $w_t(i,s,a)$ is the difference between doing an update at node i using the single *sample point* $\overline{s}$ drawn according to $P_t(s',N|s,a)$ and doing an update using the full distribution $P_t(s',N|s,a)$. The value of $u_t(i,s,a)$ captures the difference between doing an update using the current probability transitions $P_t(s',N|s,a)$ and current value functions of the children $V_t(a',s')$ and doing an update using the optimal probability transitions $P_r^*(s',N|s,a)$ and the optimal values of the children $V_r^*(a',s')$.

We now verify the conditions of Lemma 1.

Condition (a) is assumed in the conditions of the theorem with $\alpha_t(s,a) = \alpha_t(i)$.

Condition (b) is satisfied because $\overline{s}$ is sampled from $P_t(s',N|s,a)$, so the expected value of the difference is zero.

Condition (c) follows directly from the fact that $|C_t(i,s,a)|$ and $|V_t(i,s)|$ are bounded. We can show that these are bounded for both the episodic case and the discounted case as follows. In the episodic case, we have assumed all policies are proper. Hence, all trajectories terminate in finite time with a finite total reward. In the discounted case, the infinite sum of future rewards is bounded if the one-step rewards are bounded. The values of C and V are computed as temporal averages of the cumulative rewards received over a finite number of (bounded) updates, and hence, their means, variances, and maximum values are all bounded.

Condition (d) is the condition that U is a weighted max norm pseudo-contraction. We can derive this by starting with the weighted max norm for Q learning. It is well known that Q is a weighted max norm pseudo-contraction (Bertsekas & Tsitsiklis, 1996) in both the episodic case where all deterministic policies are proper (and the discount factor $\gamma = 1$) and in the infinite horizon discounted case (with $\gamma < 1$). That is, there exists a positive vector ξ and a scalar $\beta \in [0,1)$, such that for all t,

$$||TQ_t - Q^*||_\xi \leq \beta ||Q_t - Q^*||_\xi, \tag{16}$$

where T is the operator

$$(TQ)(s,a) = \sum_{s',N} P(s',N|s,a) \gamma^N [R(s'|s,a) + \max_{a'} Q(s',a')].$$

Now we will show how to derive the pseudo-contraction for the C update operator U. Our plan is to show first how to express the U operator for learning C in terms of the T operator for updating Q values. Then we will replace TQ in the pseudo-contraction equation for Q

learning with UC, and show that U is a weighted max-norm pseudo-contraction under the same weights ξ and the same β.

Recall from Eqn. (10) that $Q(i,s,a) = C(i,s,a) + V(a,s)$. Furthermore, the U operator performs its updates using the optimal value functions of the child nodes, so we can write this as $Q_t(i,s,a) = C_t(i,s,a) + V^*(a,s)$. Now once the children of node i have converged, the Q-function version of the Bellman equation for MDP M_i can be written as

$$Q(i,s,a) = \sum_{s',N} P_r^*(s',N|s,a)\gamma^N [V_r^*(a,s) + \max_{a'} Q(i,s',a')].$$

As we have noted before, $V_r^*(a,s)$ plays the role of the immediate reward function for M_i. Therefore, for node i, the T operator can be rewritten as

$$(TQ)(i,s,a) = \sum_{s',N} P_r^*(s'|s,a)\gamma^N [V_r^*(a,s) + \max_{a'} Q(i,s',a')].$$

Now we replace $Q(i,s,a)$ by $C(i,s,a) + V_r^*(a,s)$, and obtain

$$(TQ)(i,s,a) = \sum_{s',N} P_r^*(s',N|s,a)\gamma^N (V_r^*(a,s) + \max_{a'}[C(i,s',a') + V_r^*(a',s')]).$$

Note that $V_r^*(a,s)$ does not depend on s' or N, so we can move it outside the expectation and obtain

$$\begin{aligned}(TQ)(i,s,a) &= V_r^*(a,s) + \sum_{s',N} P_r^*(s',N|s,a)\gamma^N (\max_{a'}[C(i,s',a') + V_r^*(a',s')]) \\ &= V_r^*(a,s) + (UC)(i,s,a)\end{aligned}$$

Abusing notation slightly, we will express this in vector form as $TQ(i) = V_r^* + UC(i)$. Similarly, we can write $Q_t(i,s,a) = C_t(i,s,a) + V_r^*(a,s)$ in vector form as $Q_t(i) = C_t(i) + V_r^*$.

Now we can substitute these two formulas into the max norm pseudo-contraction formula for T, Eqn. (16) to obtain

$$||V_r^* + UC_t(i) - (C_r^*(i) + V_r^*)||_\xi \le \beta ||V_r^* + C_t(i) - (C_r^*(i) + V_r^*)||_\xi.$$

Thus, U is a weighted max-norm pseudo-contraction,

$$||UC_t(i) - C_r^*(i)||_\xi \le \beta ||C_t(i) - C_r^*(i)||_\xi,$$

and condition (d) is satisfied.

Finally, it is easy to verify (e), the most important condition. By assumption, the ordered GLIE policies in the child nodes converge with probability 1 to locally optimal policies for the children. Therefore $P_t(s',N|s,a)$ converges to $P_r^*(s',N|s,a)$ for all s', N, s, and a with probability 1 and $V_t(a,s)$ converges with probability 1 to $V_r^*(a,s)$ for all child actions a. Therefore, $|u_t|$ converges to zero with probability 1. We can trivially construct a sequence $\theta_t = |u_t|$ that bounds this convergence, so

$$|u_t(s,a)| \le \theta_t \le \theta_t(||C_t(s,a)||_\xi + 1).$$

We have verified all of the conditions of Lemma 1, so we can conclude that $C_t(i)$ converges to $C^*_r(i)$ with probability 1. By induction, we can conclude that this holds for all nodes in the MAXQ including the root node, so the value function represented by the MAXQ graph converges to the unique value function of the recursively optimal policy π^*_r. **Q.E.D.**

The most important aspect of this theorem is that it proves that Q learning can take place at all levels of the MAXQ hierarchy simultaneously—the higher levels do not need to wait until the lower levels have converged before they begin learning. All that is necessary is that the lower levels eventually converge to their (locally) optimal policies.

4.3 Techniques for Speeding Up MAXQ-0

Algorithm MAXQ-0 can be extended to accelerate learning in the higher nodes of the graph by a technique that we call "all-states updating". When an action a is chosen for Max node i in state s, the execution of a will move the environment through a sequence of states $s = s_1, \ldots, s_N, s_{N+1} = s'$. Because all of our subroutines are Markovian, the same resulting state s' would have been reached if we had started executing action a in state s_2, or s_3, or any state up to and including s_N. Hence, we can execute a version of line 11 in MAXQ-0 for each of these intermediate states as shown in this replacement pseudo-code:

```
11a    for $j$ from 1 to $N$ do
11b        $C_{t+1}(i, s_j, a) := (1 - \alpha_t(i)) \cdot C_t(i, s_j, a) + \alpha_t(i) \cdot \gamma^{(N+1-j)} max_{a'}\, Q_t(i, s', a')$
11c        end // for
```

In our implementation, as each composite action is executed by MAXQ-0, it constructs a linked list of the sequence of primitive states that were visited. This list is returned when the composite action terminates. The parent Max node can then process each state in this list as shown above. The parent Max node concatenates the state lists that it receives from its children and passes them to its parent when it terminates. All experiments in this paper employ all-states updating.

Kaelbling (1993) introduced a related, but more powerful, method for accelerating hierarchical reinforcement learning that she calls "all-goals updating." To understand this method, suppose that for each primitive action, there are several composite tasks that could have invoked that primitive action. In all-goals updating, whenever a primitive action is executed, the equivalent of line 11 of MAXQ-0 is applied in every composite task that could have invoked that primitive action. Sutton, Precup, and Singh (1998) prove that each of the composite tasks will converge to the optimal Q values under all-goals updating. Furthermore, they point out that the exploration policy employed for choosing the primitive actions can be different from the policies of *any* of the subtasks being learned.

It is straightforward to implement a simple form of all-goals updating within the MAXQ hierarchy for the case where composite tasks invoke primitive actions. Whenever one of the primitive actions a is executed in state s, we can update the $C(i, s, a)$ value for all parent tasks i that can invoke a.

However, additional care is required to implement all-goals updating for non-primitive actions. Suppose that by executing the exploration policy, the following sequence of world states and actions has been obtained: $s_0, a_0, s_1, \ldots, a_{k-1}, s_{k-1}, a_k, s_{k+1}$. Let j be a composite task that is terminated in state s_{k+1}, and let $s_{k-n}, a_{k-n}, \ldots, a_{k-1}, a_k$ be a sequence of actions that *could have been executed* by subtask j and its children. In other words, suppose it is possible to "parse" this state-action sequence in terms of a series of subroutine calls and returns for one invocation of subtask j. Then for each possible parent task i that invokes j, we can update the value of $C(i, s_{k-n}, j)$. Of course, in order for these updates to be useful, the exploration policy must be an ordered GLIE policy that will converge to the recursively optimal policy for subtask j and its descendents. We cannot follow an arbitrary exploration policy, because this would not produce accurate samples of result states drawn according to $P^*(s', N|s, j)$. Hence, unlike the simple case described by Sutton, Precup, and Singh, the exploration policy cannot be different from the policies of the subtasks being learned.

Although this considerably reduces the usefulness of all-goals updating, it does not completely eliminate it. A simple way of implementing non-primitive all-goals updating would be to perform MAXQ-Q learning as usual, but whenever a subtask j was invoked in state s and returned, we could update the value of $C(i, s, j)$ for all potential calling subtasks i. We have not implemented this, however, because of the complexity involved in identifying the possible actual parameters of the potential calling subroutines.

4.4 The MAXQ-Q Learning Algorithm

Now that we have shown the convergence of MAXQ-0, let us design a learning algorithm that can work with arbitrary pseudo-reward functions, $\tilde{R}_i(s')$. We could just add the pseudo-reward into MAXQ-0, but this would have the effect of changing the MDP M to have a different reward function. The pseudo-rewards "contaminate" the values of all of the completion functions computed in the hierarchy. The resulting learned policy will not be recursively optimal for the original MDP.

This problem can be solved by learning one completion function for use "inside" each Max node and a separate completion function for use "outside" the Max node. The quantities used "inside" a node will be written with a tilde: $\tilde{R}$, $\tilde{C}$, and $\tilde{Q}$. The quantities used "outside" a node will be written without the tilde.

The "outside" completion function, $C(i, s, a)$ is the completion function that we have been discussing so far in this paper. It computes the expected reward for completing task M_i after performing action a in state s and then following the learned policy for M_i. It is computed without any reference to $\tilde{R}_i$. This completion function will be used by parent tasks to compute $V(i, s)$, the expected reward for performing action i starting in state s.

The second completion function $\tilde{C}(i, s, a)$ is a completion function that we will use only "inside" node i in order to discover the locally optimal policy for task M_i. This function will incorporate rewards both from the "real" reward function, $R(s'|s, a)$, and from the pseudo-reward function, $\tilde{R}_i(s')$. It will also be used by EvaluateMaxNode in line 6 to choose the best action j^{hg} to execute. Note, however, that EvaluateMaxNode will still return the "external" value $V_t(j^{hg}, s)$ of this chosen action.

We will employ two different update rules to learn these two completion functions. The $\tilde{C}$ function will be learned using an update rule similar to the Q learning rule in line 11 of MAXQ-0. But the C function will be learned using an update rule similar to SARSA(0)—its purpose is to learn the value function for the policy that is discovered by optimizing $\tilde{C}$. Pseudo-code for the resulting algorithm, MAXQ-Q is shown in Table 4.

The key step is at lines 15 and 16. In line 15, MAXQ-Q first updates $\tilde{C}$ using the value of the greedy action, a^*, in the resulting state. This update includes the pseudo-reward $\tilde{R}_i$.

Table 4: The MAXQ-Q learning algorithm.

```
        function MAXQ-Q(MaxNode $i$, State $s$)

1       let $seq = ()$ be the sequence of states visited while executing $i$
2       if $i$ is a primitive MaxNode
3           execute $i$, receive $r$, and observe result state $s'$
4           $V_{t+1}(i,s) := (1 - \alpha_t(i)) \cdot V_t(i,s) + \alpha_t(i) \cdot r_t$
5           push $s$ onto the beginning of $seq$
6       else
7           let $count = 0$
8           while $T_i(s)$ is false do
9               choose an action $a$ according to the current exploration policy $\pi_x(i,s)$
10              let $childSeq$ = MAXQ-Q($a, s$), where $childSeq$ is the sequence of states visited
                    while executing action $a$. (in reverse order)
11              observe result state $s'$
12              let $a^* = \text{argmax}_{a'} \; [\tilde{C}_t(i,s',a') + V_t(a',s')]$
13              let $N = 1$
14              for each $s$ in $childSeq$ do
15                  $\tilde{C}_{t+1}(i,s,a) := (1 - \alpha_t(i)) \cdot \tilde{C}_t(i,s,a) + \alpha_t(i) \cdot \gamma^N [\tilde{R}_i(s') + \tilde{C}_t(i,s',a^*) + V_t(a^*,s)]$
16                  $C_{t+1}(i,s,a) := (1 - \alpha_t(i)) \cdot C_t(i,s,a) + \alpha_t(i) \cdot \gamma^N [C_t(i,s',a^*) + V_t(a^*,s')]$
17                  $N := N + 1$
18                  end // for
19              append $childSeq$ onto the front of $seq$
20              $s := s'$
21              end // while
22          end // else
23      return $seq$
        end MAXQ-Q
```

Then in line 16, MAXQ-Q updates C using this *same greedy action* a^*, even if this would not be the greedy action according to the "uncontaminated" value function. This update, of course, does not include the pseudo-reward function.

It is important to note that whereever $V_t(a,s)$ appears in this pseudo-code, it refers to the "uncontaminated" value function of state s when executing the Max node a. This is computed recursively in exactly the same way as in MAXQ-0.

Finally, note that the pseudo-code also incorporates all-states updating, so each call to MAXQ-Q returns a list of all of the states that were visited during its execution, and the updates of lines 15 and 16 are performed for each of those states. The list of states is ordered most-recent-first, so the states are updated starting with the last state visited and working backward to the starting state, which helps speed up the algorithm.

When MAXQ-Q has converged, the resulting recursively optimal policy is computed at each node by choosing the action a that maximizes $\tilde{Q}(i,s,a) = \tilde{C}(i,s,a) + V(a,s)$ (breaking ties according to the fixed ordering established by the ordered GLIE policy). It is for this reason that we gave the name "Max nodes" to the nodes that represent subtasks (and learned policies) within the MAXQ graph. Each Q node j with parent node i stores both $\tilde{C}(i,s,j)$ and $C(i,s,j)$, and it computes both $\tilde{Q}(i,s,j)$ and $Q(i,s,j)$ by invoking its child Max node j. Each Max node i takes the maximum of these Q values and computes either $V(i,s)$ or computes the best action, a^* using $\tilde{Q}$.

Corollary 1 *Under the same conditions as Theorem 3, MAXQ-Q converges to the unique recursively optimal policy for MDP M defined by MAXQ graph H, pseudo-reward functions $\tilde{R}$, and ordered GLIE exploration policy π_x.*

Proof: The argument is identical to, but more tedious than, the proof of Theorem 3. The proof of convergence of the $\tilde{C}$ values is identical to the original proof for the C values, but it relies on proving convergence of the "new" C values as well, which follows from the same weighted max norm pseudo-contraction argument. **Q.E.D.**

5. State Abstraction

There are many reasons to introduce hierarchical reinforcement learning, but perhaps the most important reason is to create opportunities for state abstraction. When we introduced the simple taxi problem in Figure 1, we pointed out that within each subtask, we can ignore certain aspects of the state space. For example, while performing a MaxNavigate(t), the taxi should make the same navigation decisions regardless of whether the passenger is in the taxi. The purpose of this section is to formalize the conditions under which it is safe to introduce such state abstractions and to show how the convergence proofs for MAXQ-Q can be extended to prove convergence in the presence of state abstraction. Specifically, we will identify five conditions that permit the "safe" introduction of state abstractions.

Throughout this section, we will use the taxi problem as a running example, and we will see how each of the five conditions will permit us to reduce the number of distinct values that must be stored in order to represent the MAXQ value function decomposition. To establish a starting point, let us compute the number of values that must be stored for the taxi problem *without* any state abstraction.

The MAXQ representation must have tables for each of the C functions at the internal nodes and the V functions at the leaves. First, at the six leaf nodes, to store $V(i, s)$, we must store 500 values at each node, because there are 500 states; 25 locations, 4 possible destinations for the passenger, and 5 possible current locations for the passenger (the four special locations and inside the taxi itself). Second, at the root node, there are two children, which requires $2 \times 500 = 1000$ values. Third, at the MaxGet and MaxPut nodes, we have 2 actions each, so each one requires 1000 values, for a total of 2000. Finally, at MaxNavigate(t), we have four actions, but now we must also consider the target parameter t, which can take four possible values. Hence, there are effectively 2000 combinations of states and t values for each action, or 8000 total values that must be represented. In total, therefore, the MAXQ representation requires 14,000 separate quantities to represent the value function.

To place this number in perspective, consider that a flat Q learning representation must store a separate value for each of the six primitive actions in each of the 500 possible states, for a total of 3,000 values. Hence, we can see that without state abstraction, the MAXQ representation requires more than four times the memory of a flat Q table!

5.1 Five Conditions that Permit State Abstraction

We now introduce five conditions that permit the introduction of state abstractions. For each condition, we give a definition and then prove a lemma which states that if the condition is satisfied, then the value function for some corresponding class of policies can be represented abstractly (i.e., by abstract versions of the V and C functions). For each condition, we then provide some rules for identifying when that condition can be satisfied and give examples from the taxi domain.

We begin by introducing some definitions and notation.

Definition 10 *Let M be a MDP and H be a MAXQ graph defined over M. Suppose that each state s can be written as a vector of values of a set of state variables. At each Max node i, suppose the state variables are partitioned into two sets X_i and Y_i, and let χ_i be a function that projects a state s onto only the values of the variables in X_i. Then H combined with χ_i is called a* state-abstracted MAXQ graph.

In cases where the state variables can be partitioned, we will often write $s = (x, y)$ to mean that a state s is represented by a vector of values for the state variables in X and a vector of values for the state variables in Y. Similarly, we will sometimes write $P(x', y', N|x, y, a)$, $V(a, x, y)$, and $\tilde{R}_a(x', y')$ in place of $P(s', N|s, a)$, $V(a, s)$, and $\tilde{R}_a(s')$, respectively.

Definition 11 (Abstract Policy) *An* abstract hierarchical policy *for MDP M with state-abstracted MAXQ graph H and associated abstraction functions χ_i, is a hierarchical policy in which each policy π_i (corresponding to subtask M_i) satisfies the condition that for any two states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$, $\pi_i(s_1) = \pi_i(s_2)$. (When π_i is a stochastic policy, such as an exploration policy, this is interpreted to mean that the probability distributions for choosing actions are the same in both states.)*

In order for MAXQ-Q to converge in the presence of state abstractions, we will require that at all times t its (instantaneous) exploration policy is an abstract hierarchical policy. One way to achieve this is to construct the exploration policy so that it only uses information from the relevant state variables in deciding what action to perform. Boltzmann exploration based on the (state-abstracted) Q values, ϵ-greedy exploration, and counter-based exploration based on abstracted states are all abstract exploration policies. Counter-based exploration based on the full state space is not an abstract exploration policy.

Now that we have introduced our notation, let us describe and analyze the five abstraction conditions. We have identified three different kinds of conditions under which abstractions can be introduced. The first kind involves eliminating irrelevant variables within a subtask of the MAXQ graph. Under this form of abstraction, nodes toward the leaves of the MAXQ graph tend to have very few relevant variables, and nodes higher in the graph have more relevant variables. Hence, this kind of abstraction is most useful at the lower levels of the MAXQ graph.

The second kind of abstraction arises from "funnel" actions. These are macro actions that move the environment from some large number of initial states to a small number of resulting states. The completion cost of such subtasks can be represented using a number of values proportional to the number of resulting states. Funnel actions tend to appear higher in the MAXQ graph, so this form of abstraction is most useful near the root of the graph.

The third kind of abstraction arises from the structure of the MAXQ graph itself. It exploits the fact that large parts of the state space for a subtask may not be reachable because of the termination conditions of its ancestors in the MAXQ graph.

We begin by describing two abstraction conditions of the first type. Then we will present two conditions of the second type. And finally, we describe one condition of the third type.

5.1.1 Condition 1: Max Node Irrelevance

The first condition arises when a set of state variables is irrelevant to a Max node.

Definition 12 (Max Node Irrelevance) *Let M_i be a Max node in a MAXQ graph H for MDP M. A set of state variables Y is* irrelevant for node *i if the state variables of M can be partitioned into two sets X and Y such that for* any *stationary abstract hierarchical policy π executed by the* descendents *of i, the following two properties hold:*

- *the state transition probability distribution $P^\pi(s', N|s, a)$ at node i can be factored into the product of two distributions:*

$$P^\pi(x', y', N|x, y, a) = P^\pi(y'|x, y, a) \cdot P^\pi(x', N|x, a), \tag{17}$$

  where y and y' give values for the variables in Y, and x and x' give values for the variables in X.

- *for any pair of states $s_1 = (x, y_1)$ and $s_2 = (x, y_2)$ such that $\chi(s_1) = \chi(s_2) = x$, and any child action a, $V^\pi(a, s_1) = V^\pi(a, s_2)$ and $\tilde{R}_i(s_1) = \tilde{R}_i(s_2)$.*

Note that the two conditions must hold for all stationary abstract policies π executed by all of the descendents of the subtask i. We will discuss below how these rather strong requirements can be satisfied in practice. First, however, we prove that these conditions are sufficient to permit the C and V tables to be represented using state abstractions.

Lemma 2 *Let M be an MDP with full-state MAXQ graph H, and suppose that state variables Y_i are irrelevant for Max node i. Let $\chi_i(s) = x$ be the associated abstraction function that projects s onto the remaining relevant variables X_i. Let π be any abstract hierarchical policy. Then the action-value function Q^π at node i can be represented compactly, with only one value of the completion function $C^\pi(i, s, j)$ for each equivalence class of states s that share the same values on the relevant variables.*

Specifically $Q^\pi(i, s, j)$ can be computed as follows:

$$Q^\pi(i, s, j) = V^\pi(j, \chi_i(s)) + C^\pi(i, \chi_i(s), j)$$

where

$$C^\pi(i, x, j) = \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [V^\pi(\pi(x'), x') + \tilde{R}_i(x') + C^\pi(i, x', \pi(x'))],$$

where $V^\pi(j', x') = V^\pi(j', x', y_0)$, $\tilde{R}_i(x') = \tilde{R}_i(x', y_0)$, and $\pi(x) = \pi(x, y_0)$ for some arbitrary value y_0 for the irrelevant state variables Y_i.

Proof: Define a new MDP $\chi_i(M_i)$ at node i as follows:

- States: $X = \{x \mid \chi_i(s) = x, \text{ for some } s \in S\}$.
- Actions: A.
- Transition probabilities: $P^\pi(x', N|x, a)$
- Reward function: $V^\pi(a, x) + \tilde{R}_i(x')$

Because π is an abstract policy, its decisions are the same for all states s such that $\chi_i(s) = x$ for some x. Therefore, it is also a well-defined policy over $\chi_i(M_i)$. The action-value function for π over $\chi_i(M_i)$ is the unique solution to the following Bellman equation:

$$Q^\pi(i, x, j) = V^\pi(j, x) + \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, x', \pi(x'))] \tag{18}$$

Compare this to the Bellman equation over M_i:

$$Q^\pi(i, s, j) = V^\pi(j, s) + \sum_{s', N} P^\pi(s', N|s, j) \cdot \gamma^N [\tilde{R}_i(s') + Q^\pi(i, s', \pi(s'))] \tag{19}$$

and note that $V^\pi(j, s) = V^\pi(j, \chi(s)) = V^\pi(j, x)$ and $\tilde{R}_i(s') = \tilde{R}_i(\chi(s')) = \tilde{R}_i(x')$. Furthermore, we know that the distribution P^π can be factored into separate distributions for Y_i and X_i. Hence, we can rewrite (19) as

$$Q^\pi(i, s, j) = V^\pi(j, x) + \sum_{y'} P(y'|x, y, j) \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, s', \pi(s'))]$$

The right-most sum does not depend on y or y', so the sum over y' evaluates to 1, and can be eliminated to give

$$Q^\pi(i, s, j) = V^\pi(j, x) + \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, s', \pi(s'))]. \tag{20}$$

Finally, note that equations (18) and (20) are identical except for the expressions for the Q values. Since the solution to the Bellman equation is unique, we must conclude that

$$Q^\pi(i, s, j) = Q^\pi(i, \chi(s), j).$$

We can rewrite the right-hand side to obtain

$$Q^\pi(i, s, j) = V^\pi(j, \chi(s)) + C^\pi(i, \chi(s), j),$$

where

$$C^\pi(i, x, j) = \sum_{x', N} P(x', N|x, j) \cdot \gamma^N [V^\pi(\pi(x'), x') + \tilde{R}_i(x') + C^\pi(i, x', \pi(x'))].$$

Q.E.D.

Of course we are primarily interested in being able to discover and represent the *optimal* policy at each node i. The following corollary shows that the optimal policy is an abstract policy, and hence, that it can be represented abstractly.

Corollary 2 *Consider the same conditions as Lemma 2, but with the change that the abstract hierarchical policy π is executed only by the descendents of node i, but not by node i. Let ω be an ordering over actions. Then the optimal ordered policy π^*_ω at node i is an abstract policy, and its action-value function can be represented abstractly.*

Proof: Define the policy ρ^*_ω to be the optimal ordered policy over the abstract MDP $\chi(M)$, and let $Q^*(i,x,j)$ be the corresponding optimal action-value function. Then by the same argument given above, Q^* is also a solution to the optimal Bellman equation for the original MDP. This means that the policy π^*_ω defined by $\pi^*_\omega(s) = \rho^*(\chi(s))$ is an optimal ordered policy, and by construction, it is an abstract policy. **Q.E.D.**

As stated, the Max node irrelevance condition appears quite difficult to satisfy, since it requires that the state transition probability distribution factor into X and Y components for all possible abstract hierarchical policies. However, in practice, this condition is often satisfied.

For example, let us consider the Navigate(t) subtask. The source and destination of the passenger are irrelevant to the achievement of this subtask. Any policy that successfully completes this subtask will have the same value function regardless of the source and destination locations of the passenger. By abstracting away the passenger source and destination, we obtain a huge savings in space. Instead of requiring 8000 values to represent the C functions for this task, we require only 400 values (4 actions, 25 locations, 4 possible values for t).

The advantages of this form of abstraction are similar to those obtained by Boutilier, Dearden and Goldszmidt (1995) in which belief network models of actions are exploited to simplify value iteration in stochastic planning. Indeed, one way of understanding the conditions of Definition 12 is to express them in the form of a decision diagram, as shown in Figure 7. The diagram shows that the irrelevant variables Y do not affect the rewards either directly or indirectly, and therefore, they do not affect either the value function or the optimal policy.

One rule for noticing cases where this abstraction condition holds is to examine the subgraph rooted at the given Max node i. If a set of state variables is irrelevant to the leaf state transition probabilities and reward functions and also to all pseudo-reward functions and termination conditions in the subgraph, then those variables satisfy the Max Node Irrelevance condition:

Lemma 3 *Let M be an MDP with associated MAXQ graph H, and let i be a Max node in H. Let X_i and Y_i be a partition of the state variables for M. A set of state variables Y_i is irrelevant to node i if*

- *For each primitive leaf node a that is a descendent of i,*

  $$P(x',y'|x,y,a) = P(y'|x,y,a)P(x'|x,a) \text{ and}$$
  $$R(x',y'|x,y,a) = R(x'|x,a),$$

- *For each internal node j that is equal to node i or is a descendent of i, $\tilde{R}_j(x',y') = \tilde{R}_j(x')$ and the termination predicate $T_j(x',y')$ is true iff $T_j(x')$.*

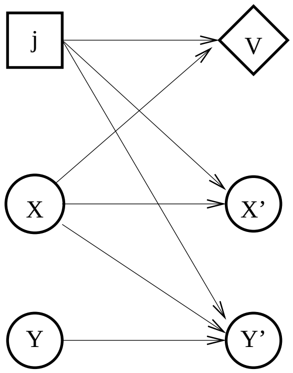


Figure 7: A dynamic decision diagram that represents the conditions of Definition 12. The probabilistic nodes X and Y represent the state variables at time t, and the nodes X' and Y' represent the state variables at a later time $t + N$. The square action node j is the chosen child subroutine, and the utility node V represents the value function $V(j, x)$ of that child action. Note that while X may influence Y', Y cannot affect X', and therefore, it cannot affect V.

Proof: We must show that any abstract hierarchical policy will give rise to an SMDP at node i whose transition probability distribution factors and whose reward function depends only on X_i. By definition, any abstract hierarchical policy will choose actions based only upon information in X_i. Because the primitive probability transition functions factor into an independent component for X_i and since the termination conditions at all nodes below i are based only on the variables in X_i, the probability transition function $P_i(x', y', N|x, y, a)$ must also factor into $P_i(y'|x, y, a)$ and $P_i(x', N|x, a)$. Similarly, all of the reward functions $V(j, x, y)$ must be equal to $V(j, x)$, because all rewards received within the subtree (either at the leaves or through pseudo-rewards) depend only on the variables in X_i. Therefore, the variables in Y_i are irrelevant for Max node i. **Q.E.D.**

In the Taxi task, the primitive navigation actions, North, South, East, and West only depend on the location of the taxi and not on the location of the passenger. The pseudo-reward function and termination condition for the MaxNavigate(t) node only depend on the location of the taxi (and the parameter t). Hence, this lemma applies, and the passenger source and destination are irrelevant for the MaxNavigate node.

5.1.2 Condition 2: Leaf Irrelevance

The second abstraction condition describes situations under which we can apply state abstractions to leaf nodes of the MAXQ graph. For leaf nodes, we can obtain a stronger result than Lemma 2 by using a slightly weaker definition of irrelevance.

Definition 13 (Leaf Irrelevance) *A set of state variables Y is* irrelevant for a primitive action *a of a MAXQ graph if for all states s the expected value of the reward function,*

$$V(a,s) = \sum_{s'} P(s'|s,a)R(s'|s,a)$$

does not depend on any of the values of the state variables in Y. In other words, for any pair of states s_1 and s_2 that differ only in their values for the variables in Y,

$$\sum_{s'_1} P(s'_1|s_1,a)R(s'_1|s_1,a) = \sum_{s'_2} P(s'_2|s_2,a)R(s'_2|s_2,a).$$

If this condition is satisfied at leaf a, then the following lemma shows that we can represent its value function $V(a,s)$ compactly.

Lemma 4 *Let M be an MDP with full-state MAXQ graph H, and suppose that state variables Y are irrelevant for leaf node a. Let $\chi(s) = x$ be the associated abstraction function that projects s onto the remaining relevant variables X. Then we can represent $V(a,s)$ for any state s by an abstracted value function $V(a,\chi(s)) = V(a,x)$.*

Proof: According to the definition of Leaf Irrelevance, any two states that differ only on the irrelevant state variables have the same value for $V(a,s)$. Hence, we can represent this unique value by $V(a,x)$. **Q.E.D.**

Here are two rules for finding cases where Leaf Irrelevance applies. The first rule shows that if the probability distribution factors, then we have Leaf Irrelevance.

Lemma 5 *Suppose the probability transition function for primitive action a, $P(s'|s,a)$, factors as $P(x',y'|x,y,a) = P(y'|x,y,a)P(x'|x,a)$ and the reward function satisfies $R(s'|s,a) = R(x'|x,a)$. Then the variables in Y are irrelevant to the leaf node a.*

Proof: Plug in to the definition of $V(a,s)$ and simplify.

$$\begin{aligned} V(a,s) &= \sum_{s'} P(s'|s,a)R(s'|s,a) \\ &= \sum_{x',y'} P(y'|x,y,a)P(x'|x,a)R(x'|x,a) \\ &= \sum_{y'} P(y'|x,y,a) \sum_{x'} P(x'|x,a)R(x'|x,a) \\ &= \sum_{x'} P(x'|x,a)R(x'|x,a) \end{aligned}$$

Hence, the expected reward for the action a depends only on the variables in X and not on the variables in Y. **Q.E.D.**

The second rule shows that if the reward function for a primitive action is constant, then we can apply state abstractions even if $P(s'|s,a)$ does not factor.

Lemma 6 *Suppose $R(s'|s,a)$ (the reward function for action a in MDP M) is always equal to a constant r_a. Then the entire state s is irrelevant to the primitive action a.*

Proof:

$$\begin{aligned} V(a,s) &= \sum_{s'} P(s'|s,a)R(s'|s,a) \\ &= \sum_{s'} P(s'|s,a)r_a \\ &= r_a. \end{aligned}$$

This does not depend on s, so the entire state is irrelevant to the primitive action a. **Q.E.D.**

This lemma is satisfied by the four leaf nodes North, South, East, and West in the taxi task, because their one-step reward is a constant (-1). Hence, instead of requiring 2000 values to store the V functions, we only need 4 values—one for each action. Similarly, the expected rewards of the Pickup and Putdown actions each require only 2 values, depending on whether the corresponding actions are legal or illegal. Hence, together, they require 4 values, instead of 1000 values.

5.1.3 Condition 3: Result Distribution Irrelevance

Now we consider a condition that results from "funnel" actions.

Definition 14 (Result Distribution Irrelevance). *A set of state variables Y_j is* irrelevant for the result distribution of action j *if, for all abstract policies π executed by node j and its descendents in the MAXQ hierarchy, the following holds: for all pairs of states s_1 and s_2 that differ only in their values for the state variables in Y_j,*

$$P^\pi(s', N|s_1, j) = P^\pi(s', N|s_2, j)$$

for all s' and N.

If this condition is satisfied for subtask j, then the C value of its parent task i can be represented compactly:

Lemma 7 *Let M be an MDP with full-state MAXQ graph H, and suppose that the set of state variables Y_j is irrelevant to the result distribution of action j, which is a child of Max node i. Let χ_{ij} be the associated abstraction function: $\chi_{ij}(s) = x$. Then we can define an abstract completion cost function $C^\pi(i, \chi_{ij}(s), j)$ such that for all states s,*

$$C^\pi(i, s, j) = C^\pi(i, \chi_{ij}(s), j).$$

Proof: The completion function for fixed policy π is defined as follows:

$$C^\pi(i, s, j) = \sum_{s', N} P(s', N|s, j) \cdot \gamma^N Q^\pi(i, s'). \tag{21}$$

Consider any two states s_1 and s_2, such that $\chi_{ij}(s_1) = \chi_{ij}(s_2) = x$. Under Result Distribution Irrelevance, their transition probability distributions are the same. Hence, the right-hand sides of (21) have the same value, and we can conclude that

$$C^\pi(i, s_1, j) = C^\pi(i, s_2, j).$$

Therefore, we can define an abstract completion function, $C^{\pi}(i, x, j)$ to represent this quantity. **Q.E.D.**

In undiscounted cumulative reward problems, the definition of result distribution irrelevance can be weakened to eliminate N, the number of steps. All that is needed is that for all pairs of states s_1 and s_2 that differ only in the irrelevant state variables, $P^{\pi}(s'|s_1, j) = P^{\pi}(s'|s_2, j)$ (for all s'). In the undiscounted case, Lemma 7 still holds under this revised definition.

It might appear that the result distribution irrelevance condition would rarely be satisfied, but we often find cases where the condition is true. Consider, for example, the Get subroutine for the taxi task. No matter what location the taxi has in state s, the taxi will be at the passenger's starting location when the Get finishes executing (i.e., because the taxi will have just completed picking up the passenger). Hence, the starting location is irrelevant to the resulting location of the taxi, and $P(s'|s_1, \mathsf{Get}) = P(s'|s_2, \mathsf{Get})$ for all states s_1 and s_2 that differ only in the taxi's location.

Note, however, that if we were maximizing discounted reward, the taxi's location would not be irrelevant, because the probability that Get will terminate in exactly N steps would depend on the location of the taxi, which could differ in states s_1 and s_2. Different values of N will produce different amounts of discounting in (21), and hence, we cannot ignore the taxi location when representing the completion function for Get.

But in the undiscounted case, by applying Lemma 7, we can represent $C(\mathsf{Root}, s, \mathsf{Get})$ using 16 distinct values, because there are 16 equivalence classes of states (4 source locations times 4 destination locations). This is much less than the 500 quantities in the unabstracted representation.

Note that although state variables Y may be irrelevant to the result distribution of a subtask j, they may be important *within* subtask j. In the Taxi task, the location of the taxi is critical for representing the value of $V(\mathsf{Get}, s)$, but it is irrelevant to the result state distribution for Get, and therefore it is irrelevant for representing $C(\mathsf{Root}, s, \mathsf{Get})$. Hence, the MAXQ decomposition is essential for obtaining the benefits of result distribution irrelevance.

"Funnel" actions arise in many hierarchical reinforcement learning problems. For example, abstract actions that move a robot to a doorway or that move a car onto the entrance ramp of a freeway have this property. The Result Distribution Irrelevance condition is applicable in all such situations as long as we are in the undiscounted setting.

5.1.4 Condition 4: Termination

The fourth condition is closely related to the "funnel" property. It applies when a subtask is guaranteed to cause its parent task to terminate in a goal state. In a sense, the subtask is funneling the environment into the set of states described by the goal predicate of the parent task.

Lemma 8 (Termination). *Let M_i be a task in a MAXQ graph such that for all states s where the goal predicate $G_i(s)$ is true, the pseudo-reward function $\tilde{R}_i(s) = 0$. Suppose there is a child task a and state s such that for all hierarchical policies π,*

$$\forall\, s'\ P_i^{\pi}(s', N|s, a) > 0 \ \Rightarrow\ G_i(s').$$

(i.e., every possible state s' that results from applying a in s will make the goal predicate, G_i, true.)

Then for any policy executed at node i, the completion cost $C(i, s, a)$ is zero and does not need to be explicitly represented.

Proof: When action a is executed in state s, it is guaranteed to result in a state s' such that $G_i(s)$ is true. By definition, goal states also satisfy the termination predicate $T_i(s)$, so task i will terminate. Because $G_i(s)$ is true, the terminal pseudo-reward will be zero, and hence, the completion function will always be zero. **Q.E.D.**

For example, in the Taxi task, in all states where the taxi is holding the passenger, the Put subroutine will succeed and result in a goal terminal state for Root. This is because the termination predicate for Put (i.e., that the passenger is at his or her destination location) implies the goal condition for Root (which is the same). This means that $C(\mathsf{Root}, s, \mathsf{Put})$ is uniformly zero, for all states s where Put is not terminated.

It is easy to detect cases where the Termination condition is satisfied. We only need to compare the termination predicate T_a of a subtask with the goal predicate G_i of the parent task. If the first implies the second, then the termination lemma is satisfied.

5.1.5 Condition 5: Shielding

The shielding condition arises from the structure of the MAXQ graph.

Lemma 9 (Shielding). *Let M_i be a task in a MAXQ graph and s be a state such that in all paths from the root of the graph down to node M_i there is a subtask j (possibly equal to i) whose termination predicate $T_j(s)$ is true, then the Q nodes of M_i do not need to represent C values for state s.*

Proof: In order for task i to be executed in state s, there must exist some path of ancestors of task i leading up to the root of the graph such that all of those ancestor tasks are not terminated. The condition of the lemma guarantees that this is false, and hence that task i cannot be executed in state s. Therefore, no C values need to be represented. **Q.E.D.**

As with the Termination condition, the Shielding condition can be verified by analyzing the structure of the MAXQ graph and identifying nodes whose ancestor tasks are terminated.

In the Taxi domain, a simple example of this arises in the Put task, which is terminated in all states where the passenger is not in the taxi. This means that we do not need to represent $C(\mathsf{Root}, s, \mathsf{Put})$ in these states. The result is that, when combined with the Termination condition above, we do not need to explicitly represent the completion function for Put at all!

5.1.6 Dicussion

By applying these five abstraction conditions, we obtain the following "safe" state abstractions for the Taxi task:

- North, South, East, and West. These terminal nodes require one quantity each, for a total of four values. (Leaf Irrelevance).

- Pickup and Putdown each require 2 values (legal and illegal states), for a total of four. (Leaf Irrelevance.)

- QNorth(t), QSouth(t), QEast(t), and QWest(t) each require 100 values (four values for t and 25 locations). (Max Node Irrelevance.)

- QNavigateForGet requires 4 values (for the four possible source locations). (The passenger destination is Max Node Irrelevant for MaxGet, and the taxi starting location is Result Distribution Irrelevant for the Navigate action.)

- QPickup requires 100 possible values, 4 possible source locations and 25 possible taxi locations. (Passenger destination is Max Node Irrelevant to MaxGet.)

- QGet requires 16 possible values (4 source locations, 4 destination locations). (Result Distribution Irrelevance.)

- QNavigateForPut requires only 4 values (for the four possible destination locations). (The passenger source and destination are Max Node Irrelevant to MaxPut; the taxi location is Result Distribution Irrelevant for the Navigate action.)

- QPutdown requires 100 possible values (25 taxi locations, 4 possible destination locations). (Passenger source is Max Node Irrelevant for MaxPut.)

- QPut requires 0 values. (Termination and Shielding.)

This gives a total of 632 distinct values, which is much less than the 3000 values required by flat Q learning. Hence, we can see that by applying state abstractions, the MAXQ representation can give a much more compact representation of the value function.

A key thing to note is that with these state abstractions, the value function is decomposed into a sum of terms such that no single term depends on the entire state of the MDP, even though the value function as a whole does depend on the entire state of the MDP. For example, consider again the state described in Figures 1 and 4. There, we showed that the value of a state s_1 with the passenger at R, the destination at B, and the taxi at (0,3) can be decomposed as

$$\begin{aligned} V(\mathsf{Root}, s_1) &= V(\mathsf{North}, s_1) + C(\mathsf{Navigate}(R), s_1, \mathsf{North}) + \\ &\quad C(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) + C(\mathsf{Root}, s_1, \mathsf{Get}) \end{aligned}$$

With state abstractions, we can see that each term on the right-hand side only depends on a subset of the features:

- $V(\mathsf{North}, s_1)$ is a constant

- $C(\mathsf{Navigate}(R), s_1, \mathsf{North})$ depends only on the taxi location and the passenger's source location.

- $C(\mathsf{Get}, s_1, \mathsf{Navigate}(R))$ depends only on the source location.

- $C(\mathsf{Root}, s_1, \mathsf{Get})$ depends only on the passenger's source and destination.

Without the MAXQ decomposition, no features are irrelevant, and the value function depends on the entire state.

What prior knowledge is required on the part of a programmer in order to identify these state abstractions? It suffices to know some qualitative constraints on the one-step reward functions, the one-step transition probabilities, and termination predicates, goal predicates, and pseudo-reward functions within the MAXQ graph. Specifically, the Max Node Irrelevance and Leaf Irrelevance conditions require simple analysis of the one-step transition function and the reward and pseudo-reward functions. Opportunities to apply the Result Distribution Irrelevance condition can be found by identifying "funnel" effects that result from the definitions of the termination conditions for operators. Similarly, the Shielding and Termination conditions only require analysis of the termination predicates of the various subtasks. Hence, applying these five conditions to introduce state abstractions is a straightforward process, and once a model of the one-step transition and reward functions has been learned, the abstraction conditions can be checked to see if they are satisfied.

5.2 Convergence of MAXQ-Q with State Abstraction

We have shown that state abstractions can be safely introduced into the MAXQ value function decomposition under the five conditions described above. However, these conditions only guarantee that the value function of any fixed *abstract* hierarchical policy can be represented—they do not show that recursively optimal policies can be represented, nor do they show that the MAXQ-Q learning algorithm will find a recursively optimal policy when it is forced to use these state abstractions. The goal of this section is to prove these two results: (a) that the ordered recursively-optimal policy is an abstract policy (and, hence, can be represented using state abstractions) and (b) that MAXQ-Q will converge to this policy when applied to a MAXQ graph with safe state abstractions.

Lemma 10 *Let M be an MDP with full-state MAXQ graph H and abstract-state MAXQ graph $\chi(H)$ where the abstractions satisfy the five conditions given above. Let ω be an ordering over all actions in the MAXQ graph. Then the following statements are true:*

- *The unique ordered recursively-optimal policy π_r^* defined by M, H, and ω is an abstract policy (i.e., it depends only on the relevant state variables at each node; see Definition 11),*
- *The C and V functions in $\chi(H)$ can represent the projected value function of π_r^*.*

Proof: The five abstraction lemmas tell us that if the ordered recursively-optimal policy is abstract, then the C and V functions of $\chi(H)$ can represent its value function. Hence, the heart of this lemma is the first claim. The last two forms of abstraction (Shielding and Termination) do not place any restrictions on abstract policies, so we ignore them in this proof.

The proof is by induction on the levels of the MAXQ graph, starting at the leaves. As a base case, let us consider a Max node i all of whose children are primitive actions. In this case, there are no policies executed *within* the children of the Max node. Hence if variables Y_i are irrelevant for node i, then we can apply our abstraction lemmas to represent the value function of any policy at node i—not just abstract policies. Consequently, the value function of any optimal policy for node i can be represented, and it will have the property that

$$Q^*(i, s_1, a) = Q^*(i, s_2, a) \tag{22}$$

for any states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$.

Now let us impose the action ordering ω to compute the optimal *ordered* policy. Consider two actions a_1 and a_2 such that $\omega(a_1, a_2)$ (i.e., ω prefers a_1), and suppose that there is a "tie" in the Q^* function at state s_1 such that the values

$$Q^*(i, s_1, a_1) = Q^*(i, s_1, a_2)$$

and they are the only two actions that maximize Q^* in this state. Then the optimal ordered policy must choose a_1. Now in all other states s_2 such that $\chi_i(s_1) = \chi_i(s_2)$, we have just established in (22) that the Q^* values will be the same. Hence, the same tie will exist between a_1 and a_2, and hence, the optimal ordered policy must make the same choice in all such states. Hence, the optimal ordered policy for node i is an abstract policy.

Now let us turn to the recursive case at Max node i. Make the inductive assumption that the ordered recursively-optimal policy is abstract within all descendent nodes and consider the locally optimal policy at node i. If Y is a set of state variables that are irrelevant to node i, Corollary 2 tells us that $Q^*(i, s_1, j) = Q^*(i, s_2, j)$ for all states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$. Similarly, if Y is a set of variables irrelevant to the result distribution of a particular action j, then Lemma 7 tells us the same thing. Hence, by the same ordering argument given above, the ordered optimal policy at node i must be abstract. By induction, this proves the lemma. **Q.E.D.**

With this lemma, we have established that the combination of an MDP M, an abstract MAXQ graph H, and an action ordering defines a unique recursively-optimal ordered abstract policy. We are now ready to prove that MAXQ-Q will converge to this policy.

Theorem 4 *Let $M = \langle S, A, P, R, P_0 \rangle$ be either an episodic MDP for which all deterministic policies are proper or a discounted infinite horizon MDP with discount factor $\gamma < 1$. Let H be an unabstracted MAXQ graph defined over subtasks $\{M_0, \ldots, M_k\}$ with pseudo-reward functions $\tilde{R}_i(s')$. Let $\chi(H)$ be a state-abstracted MAXQ graph defined by applying state abstractions χ_i to each node i of H under the five conditions given above. Let $\pi_x(i, \chi_i(s))$ be an abstract ordered GLIE exploration policy at each node i and state s whose decisions depend only on the "relevant" state variables at each node i. Let π_r^* be the unique recursively-optimal hierarchical policy defined by π_x, M, and $\tilde{R}$. Then with probability 1, algorithm MAXQ-Q applied to $\chi(H)$ converges to π_r^* provided that the learning rates $\alpha_t(i)$ satisfy Equation (15) and the one-step rewards are bounded.*

Proof: Rather than repeating the entire proof for MAXQ-Q, we will only describe what must change under state abstraction. The last two forms of state abstraction refer to states whose values can be inferred from the structure of the MAXQ graph, and therefore do not need to be represented at all. Since these values are not updated by MAXQ-Q, we can ignore them. We will now consider the first three forms of state abstraction in turn.

We begin by considering primitive leaf nodes. Let a be a leaf node and let Y be a set of state variables that are Leaf Irrelevant for a. Let $s_1 = (x, y_1)$ and $s_2 = (x, y_2)$ be two states that differ only in their values for Y. Under Leaf Irrelevance, the probability transitions $P(s'_1|s_1, a)$ and $P(s'_2|s_2, a)$ need not be the same, but the expected reward of performing a in both states must be the same. When MAXQ-Q visits an abstract state x, it does not "know" the value of y, the part of the state that has been abstracted away. Nonetheless, it draws a sample according to $P(s'|x, y, a)$, receives a reward $R(s'|x, y, a)$, and updates its estimate of $V(a, x)$ (line 4 of MAXQ-Q). Let $P_t(y)$ be the probability that MAXQ-Q is visiting (x, y) given that the unabstracted part of the state is x. Then Line 4 of MAXQ-Q is computing a stochastic approximation to

$$\sum_{s',N,y} P_t(y)P_t(s', N|x, y, a)R(s'|x, y, a).$$

We can write this as

$$\sum_y P_t(y) \sum_{s',N} P_t(s', N|x, y, a)R(s'|x, y, a).$$

According to Leaf Irrelevance, the inner sum has the same value for all states s such that $\chi(s) = x$. Call this value $r_0(x)$. This gives

$$\sum_y P_t(y)r_0(x),$$

which is equal to $r_0(x)$ for any distribution $P_t(y)$. Hence, MAXQ-Q converges under Leaf Irrelevance abstractions.

Now let us turn to the two forms of abstraction that apply to internal nodes: Max Node Irrelevance and Result Distribution Irrelevance. Consider the SMDP defined at each node i of the abstracted MAXQ graph at time t during MAXQ-Q. This would be an ordinary SMDP with transition probability function $P_t(x', N|x, a)$ and reward function $V_t(a, x) + \tilde{R}_i(x')$ except that when MAXQ-Q draws samples of state transitions, they are drawn according to the distribution $P_t(s', N|s, a)$ over the original state space. To prove the theorem, we must show that drawing (s', N) according to this second distribution is equivalent to drawing (x', N) according to the first distribution.

For Max Node Irrelevance, we know that for all abstract policies applied to node i and its descendents, the transition probability distribution factors as

$$P(s', N|s, a) = P(y'|x, y, a)P(x', N|x, a).$$

Because the exploration policy is an abstract policy, $P_t(s', N|s, a)$ factors in this way. This means that the Y_i components of the state cannot affect the X_i components, and hence, sampling from $P_t(s', N|s, a)$ and discarding the Y_i values gives samples for $P_t(x', N|x, a)$. Therefore, MAXQ-Q will converge under Max Node Irrelevance abstractions.

Finally, consider Result Distribution Irrelevance. Let j be a child of node i, and suppose Y_j is a set of state variables that are irrelevant to the result distribution of j. When the SMDP at node i wishes to draw a sample from $P_t(x', N|x, j)$, it does not "know" the current value of y, the irrelevant part of the current state. However, this does not matter, because Result Distribution Irrelevance means that for all possible values of y, $P_t(x', y', N|x, y, j)$ is the same. Hence, MAXQ-Q will converge under Result Distribution Irrelevance abstractions.

In each of these three cases, MAXQ-Q will converge to a locally-optimal ordered policy at node i in the MAXQ graph. By Lemma 10, this produces a locally-optimal ordered policy for the unabstracted SMDP at node i. Hence, by induction, MAXQ-Q will converge to the unique ordered recursively optimal policy π_r^* defined by MAXQ-Q H, MDP M, and ordered exploration policy π_x. **Q.E.D.**

5.3 The Hierarchical Credit Assignment Problem

There are still some situations where we would like to introduce state abstractions but where the five properties described above do not permit them. Consider the following modification of the taxi problem. Suppose that the taxi has a fuel tank and that each time the taxi moves one square, it costs one unit of fuel. If the taxi runs out of fuel before delivering the passenger to his or her destination, it receives a reward of -20, and the trial ends. Fortunately, there is a filling station where the taxi can execute a Fillup action to fill the fuel tank.

To solve this modified problem using the MAXQ hierarchy, we can introduce another subtask, Refuel, which has the goal of moving the taxi to the filling station and filling the tank. MaxRefuel is a child of MaxRoot, and it invokes Navigate(t) (with t bound to the location of the filling station) to move the taxi to the filling station.

The introduction of fuel and the possibility that we might run out of fuel means that we must include the current amount of fuel as a feature in representing every C value (for internal nodes) and V value (for leaf nodes) throughout the MAXQ graph. This is unfortunate, because our intuition tells us that the amount of fuel should have no influence on our decisions inside the Navigate(t) subtask. That is, either the taxi will have enough fuel to reach the target t (in which case, the chosen navigation actions do not depend on the fuel), or else the taxi will not have enough fuel, and hence, it will fail to reach t regardless of what navigation actions are taken. In other words, the Navigate(t) subtask should not need to worry about the amount of fuel, because even if there is not enough fuel, there is no action that Navigate(t) can take to get more fuel. Instead, it is the top-level subtasks that should be monitoring the amount of fuel and deciding whether to go refuel, to go pick up the passenger, or to go deliver the passenger.

Given this intuition, it is natural to try abstracting away the "amount of remaining fuel" within the Navigate(t) subtask. However, this doesn't work, because when the taxi runs out of fuel and a -20 reward is given, the QNorth, QSouth, QEast, and QWest nodes cannot "explain" why this reward was received—that is, they have no consistent way of setting their C tables to predict when this negative reward will occur, because their C values ignore the amount of fuel in the tank. Stated more formally, the difficulty is that the Max Node Irrelevance condition is not satisfied because the one-step reward function $R(s'|s,a)$ for these actions depends on the amount of fuel.

We call this the *hierarchical credit assignment problem.* The fundamental issue here is that in the MAXQ decomposition all information about rewards is stored in the leaf nodes of the hierarchy. We would like to separate out the basic rewards received for navigation (i.e., -1 for each action) from the reward received for exhausting fuel (-20). If we make the reward at the leaves only depend on the location of the taxi, then the Max Node Irrelevance condition will be satisfied.

One way to do this is to have the programmer manually decompose the reward function and indicate which nodes in the hierarchy will "receive" each reward. Let $R(s'|s,a) = \sum_i R(i,s'|s,a)$ be a decomposition of the reward function, such that $R(i,s'|s,a)$ specifies that part of the reward that must be handled by Max node i. In the modified taxi problem, for example, we can decompose the reward so that the leaf nodes receive all of the original penalties, but the out-of-fuel rewards must be handled by MaxRoot. Lines 15 and 16 of the MAXQ-Q algorithm are easily modified to include $R(i,s'|s,a)$.

In most domains, we believe it will be easy for the designer of the hierarchy to decompose the reward function. It has been straightforward in all of the problems we have studied. However, an interesting problem for future research is to develop an algorithm that can solve the hierarchical credit assignment problem autonomously.

6. Non-Hierarchical Execution of the MAXQ Hierarchy

Up to this point in the paper, we have focused exclusively on representing and learning hierarchical policies. However, often the optimal policy for a MDP is not strictly hierarchical. Kaelbling (1993) first introduced the idea of deriving a non-hierarchical policy from the value function of a hierarchical policy. In this section, we exploit the MAXQ decomposition to generalize her ideas and apply them recursively at all levels of the hierarchy. We will describe two methods for non-hierarchical execution.

The first method is based on the dynamic programming algorithm known as policy iteration. The policy iteration algorithm starts with an initial policy π^0. It then repeats the following two steps until the policy converges. In the *policy evaluation* step, it computes the value function V^{π_k} of the current policy π_k. Then, in the *policy improvement step*, it computes a new policy, π_{k+1} according to the rule

$$\pi_{k+1}(s) := \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma V^{\pi_k}(s')]. \tag{23}$$

Howard (1960) proved that if π_k is not an optimal policy, then π_{k+1} is guaranteed to be an improvement. Note that in order to apply this method, we need to know the transition probability distribution $P(s'|s,a)$ and the reward function $R(s'|s,a)$.

If we know $P(s'|s,a)$ and $R(s'|s,a)$, we can use the MAXQ representation of the value function to perform one step of policy iteration. We start with a hierarchical policy π and represent its value function using the MAXQ hierarchy (e.g., π could have been learned via MAXQ-Q). Then, we can perform one step of policy improvement by applying Equation (23) using $V^\pi(0,s')$ (computed by the MAXQ hierarchy) to compute $V^\pi(s')$.

Corollary 3 *Let $\pi^g(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma V^\pi(0,s)]$, where $V^\pi(0,s)$ is the value function computed by the MAXQ hierarchy and a is a primitive action. Then, if π was not an optimal policy, π^g is strictly better for at least one state in S.*

Proof: This is a direct consequence of Howard's policy improvement theorem. **Q.E.D.**

Unfortunately, we can't iterate this policy improvement process, because the new policy, π^g is very unlikely to be a hierarchical policy (i.e., it is unlikely to be representable in

Table 5: The procedure for executing the one-step greedy policy.

```
    procedure ExecuteHGPolicy(s)
1       repeat
2           Let $\langle V(0, s), a\rangle$ := EvaluateMaxNode$(0, s)$
3           execute primitive action $a$
4           Let $s$ be the resulting state
    end // ExecuteHGPolicy
```

terms of local policies for each node of the MAXQ graph). Nonetheless, one step of policy improvement can give very significant improvements.

This approach to non-hierarchical execution ignores the internal structure of the MAXQ graph. In effect, the MAXQ hierarchy is just viewed as a way to represent V^π—any other representation would give the same one-step improved policy π^g.

The second approach to non-hierarchical execution borrows an idea from Q learning. One of the great beauties of the Q representation for value functions is that we can compute one step of policy improvement without knowing $P(s'|s, a)$, simply by taking the new policy to be $\pi^g(s) := \operatorname{argmax}_a Q(s, a)$. This gives us the same one-step greedy policy as we computed above using one-step lookahead. With the MAXQ decomposition, we can perform these policy improvement steps *at all levels of the hierarchy.*

We have already defined the function that we need. In Table 3 we presented the function EvaluateMaxNode, which, given the current state s, conducts a search along all paths from a given Max node i to the leaves of the MAXQ graph and finds the path with the best value (i.e., with the maximum sum of C values along the path, plus the V value at the leaf). This is equivalent to computing the best action greedily at each level of the MAXQ graph. In addition, EvaluateMaxNode returns the primitive action a at the end of this best path. This action a would be the first primitive action to be executed if the learned hierarchical policy were executed starting in the current state s. Our second method for non-hierarchical execution of the MAXQ graph is to call EvaluateMaxNode in each state, and execute the primitive action a that is returned. The pseudo-code is shown in Table 5.

We will call the policy computed by ExecuteHGPolicy the *hierarchical greedy policy*, and denote it π^{hg*}, where the superscript * indicates that we are computing the greedy action at each time step. The following theorem shows that this can give a better policy than the original, hierarchical policy.

Theorem 5 *Let G be a MAXQ graph representing the value function of hierarchical policy π (i.e., in terms of $C^\pi(i, s, j)$, computed for all i, s, and j). Let $V^{hg}(0, s)$ be the value computed by* ExecuteHGPolicy *(line 2), and let π^{hg*} be the resulting policy. Define V^{hg*} to be the value function of π^{hg*}. Then for all states s, it is the case that*

$$V^\pi(s) \leq V^{hg}(0, s) \leq V^{hg*}(s). \tag{24}$$

Proof: (sketch) The left inequality in Equation (24) is satisfied by construction by line 6 of EvaluateMaxNode. To see this, consider that the original hierarchical policy, π, can be viewed as choosing a "path" through the MAXQ graph running from the root to one of the leaf nodes, and $V^{\pi}(0,s)$ is the sum of the C^{π} values along this chosen path (plus the V^{π} value at the leaf node). In contrast, EvaluateMaxNode performs a traversal of *all* paths through the MAXQ graph and finds the *best* path, that is, the path with the largest sum of C^{π} (and leaf V^{π}) values. Hence, $V^{hg}(0,s)$ must be at least as large as $V^{\pi}(0,s)$.

To establish the right inequality, note that by construction $V^{hg}(0,s)$ is the value function of a policy, call it π^{hg}, that chooses one action greedily at each level of the MAXQ graph (recursively), and then follows π thereafter. This is a consequence of the fact that line 6 of EvaluateMaxNode has C^{π} on its right-hand side, and C^{π} represents the cost of "completing" each subroutine by following π, not by following some other, greedier, policy. (In Table 3, C^{π} is written as C_t.) However, when we execute ExecuteHGPolicy (and hence, execute π^{hg*}), we have an opportunity to improve upon π and π^{hg} at each time step. Hence, $V^{hg}(0,s)$ is an underestimate of the actual value of π^{hg*}. **Q.E.D.**

Note that this theorem only works in one direction. It says that if we can find a state where $V^{hg}(0,s) > V^{\pi}(s)$, then the greedy policy, π^{hg*}, will be strictly better than π. However, it could be that π is not an optimal policy and yet the structure of the MAXQ graph prevents us from considering an action (either primitive or composite) that would improve π. Hence, unlike the policy improvement theorem of Howard (where all primitive actions are always eligible to be chosen), we do not have a guarantee that if π is suboptimal, then the hierarchically greedy policy is a strict improvement.

In contrast, if we perform one-step policy improvement as discussed at the start of this section, Corollary 3 guarantees that we will improve the policy. So we can see that in general, neither of these two methods for non-hierarchical execution is always better than the other. Nonetheless, the first method only operates at the level of individual primitive actions, so it is not able to produce very large improvements in the policy. In contrast, the hierarchical greedy method can obtain very large improvements in the policy by changing which actions (i.e., subroutines) are chosen near the root of the hierarchy. Hence, in general, hierarchical greedy execution is probably the better method. (Of course, the value functions of both methods could be computed, and the one with the better estimated value could be executed.)

Sutton, et al. (1999) have simultaneously developed a closely-related method for non-hierarchical execution of macros. Their method is equivalent to ExecuteHGPolicy for the special case where the MAXQ hierarchy has only one level of subtasks. The interesting aspect of ExecuteHGPolicy is that it permits greedy improvements at all levels of the tree to influence which action is chosen.

Some care must be taken in applying Theorem 5 to a MAXQ hierarchy whose C values have been learned via MAXQ-Q. Being an online algorithm, MAXQ-Q will not have correctly learned the values of *all* states at all nodes of the MAXQ graph. For example, in the taxi problem, the value of $C(\mathsf{Put}, s, \mathsf{QPutdown})$ will not have been learned very well except at the four special locations R, G, B, and Y. This is because the Put subtask cannot be executed until the passenger is in the taxi, and this usually means that a Get has just been completed, so the taxi is at the passenger's source location. During exploration, both children of Put will be tried in such states. The PutDown will usually fail (and receive a negative reward), whereas the Navigate will eventually succeed (perhaps after lengthy exploration) and take the taxi to the destination location. Now because of all-states updating, the values for $C(\mathsf{Put}, s, \mathsf{Navigate}(t))$ will have been learned at all of the states along the path to the passenger's destination, but the C values for the Putdown action will only be learned for the passenger's source and destination locations. Hence, if we train the MAXQ representation using hierarchical execution (as in MAXQ-Q), and then switch to hierarchically-greedy execution, the results will be quite bad. In particular, we need to introduce hierarchically-greedy execution early enough so that the exploration policy is still actively exploring. (In theory, a GLIE exploration policy never ceases to explore, but in practice, we want to find a good policy quickly, not just asymptotically).

Of course an alternative would be to use hierarchically-greedy execution from the very beginning of learning. However, remember that the higher nodes in the MAXQ hierarchy need to obtain samples of $P(s', N|s, a)$ for each child action a. If the hierarchical greedy execution interrupts child a before it has reached a terminal state (i.e., because at some state along the way, another subtask appears better to EvaluateMaxNode), then these samples cannot be obtained. Hence, it is important to begin with purely hierarchical execution during training, and make a transition to greedy execution at some point.

The approach we have taken is to implement MAXQ-Q in such a way that we can specify a number of primitive actions L that can be taken hierarchically before the hierarchical execution is "interrupted" and control returns to the top level (where a new action can be chosen greedily). We start with L set very large, so that execution is completely hierarchical—when a child action is invoked, we are committed to execute that action until it terminates. However, gradually, we reduce L until it becomes 1, at which point we have hierarchical greedy execution. We time this so that it reaches 1 at about the same time our Boltzmann exploration cools to a temperature of 0.1 (which is where exploration effectively has halted). As the experimental results will show, this generally gives excellent results with very little added exploration cost.

7. Experimental Evaluation of the MAXQ Method

We have performed a series of experiments with the MAXQ method with three goals in mind: (a) to understand the expressive power of the value function decomposition, (b) to characterize the behavior of the MAXQ-Q learning algorithm, and (c) to assess the relative importance of temporal abstraction, state abstraction, and non-hierarchical execution. In this section, we describe these experiments and present the results.

7.1 The Fickle Taxi Task

Our first experiments were performed on a modified version of the taxi task. This version incorporates two changes to the task described in Section 3.1. First, each of the four navigation actions is noisy, so that with probability 0.8 it moves in the intended direction, but with probability 0.1 it instead moves to the right (of the intended direction) and with probability 0.1 it moves to the left. The purpose of this change is to create a more realistic and more difficult challenge for the learning algorithms. The second change is that after the taxi has picked up the passenger and moved one square away from the passenger's source location, the passenger changes his or her destination location with probability 0.3. The purpose of this change is to create a situation where the optimal policy is not a hierarchical policy so that the effectiveness of non-hierarchical execution can be measured.

We compared four different configurations of the learning algorithm: (a) flat Q learning, (b) MAXQ-Q learning without any form of state abstraction, (c) MAXQ-Q learning with state abstraction, and (d) MAXQ-Q learning with state abstraction and greedy execution. These configurations are controlled by many parameters. These include the following: (a) the initial values of the Q and C functions, (b) the learning rate (we employed a fixed learning rate), (c) the cooling schedule for Boltzmann exploration (the GLIE policy that we employed), and (d) for non-hierarchical execution, the schedule for decreasing L, the number of steps of consecutive hierarchical execution. We optimized these settings separately for each configuration with the goal of matching or exceeding (with as few primitive training actions as possible) the best policy that we could code by hand. For Boltzmann exploration, we established an initial temperature and then a cooling rate. A separate temperature is maintained for each Max node in the MAXQ graph, and its temperature is reduced by multiplying by the cooling rate each time that subtask terminates in a goal state.

The process of optimizing the parameter settings for each algorithm is time-consuming, both for flat Q learning and for MAXQ-Q. The most critical parameter is the schedule for cooling the temperature of Boltzmann exploration: if this is cooled too rapidly, then the algorithms will converge to a suboptimal policy. In each case, we tested nine different cooling rates. To choose the different cooling rates for the various subtasks, we started by using fixed policies (e.g., either random or hand-coded) for all subtasks except the subtasks closest to the leaves. Then, once we had chosen schedules for those subtasks, we allowed their parent tasks to learn their policies while we tuned their cooling rates, and so on. One nice effect of our method of cooling the temperature only when a subtask terminates is that it naturally causes the subtasks higher in the MAXQ graph to cool more slowly. This meant that good results could often be obtained just by using the same cooling rate for all Max nodes.

The choice of learning rate is easier, since it is determined primarily by the degree of stochasticity in the environment. We only tested three or four different rates for each configuration. The initial values for the Q and C functions were set based on our knowledge of the problems—no experiments were required.

We took more care in tuning these parameters for these experiments than one would normally take in a real application, because we wanted to ensure that each method was compared under the best possible conditions. The general form of the results (particularly the speed of learning) is the same for wide ranges of the cooling rate and learning rate parameter settings.

The following parameters were selected based on the tuning experiments. For flat Q learning: initial Q values of 0.123 in all states, learning rate 0.25, and Boltzmann exploration with an initial temperature of 50 and a cooling rate of 0.9879. (We use initial values that end in .123 as a "signature" during debugging to detect when a weight has been modified.)

For MAXQ-Q learning without state abstraction, we used initial values of 0.123, a learning rate of 0.50, and Boltzmann exploration with an initial temperature of 50 and cooling rates of 0.9996 at MaxRoot and MaxPut, 0.9939 at MaxGet, and 0.9879 at MaxNavigate.

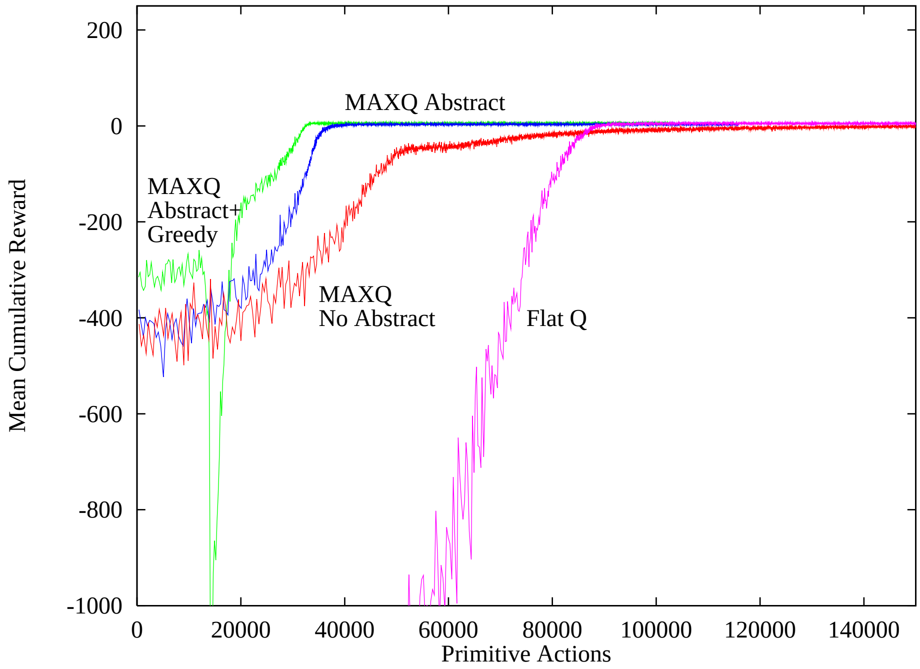


Figure 8: Comparison of performance of hierarchical MAXQ-Q learning (without state abstractions, with state abstractions, and with state abstractions combined with hierarchical greedy evaluation) to flat Q learning.

For MAXQ-Q learning with state abstraction, we used initial values of 0.123, a learning rate of 0.25, and Boltzmann exploration with an initial temperature of 50 and cooling rates of 0.9074 at MaxRoot, 0.9526 at MaxPut, 0.9526 at MaxGet, and 0.9879 at MaxNavigate.

For MAXQ-Q learning with non-hierarchical execution, we used the same settings as with state abstraction. In addition, we initialized L to 500 and decreased it by 10 with each trial until it reached 1. So after 50 trials, execution was completely greedy.

Figure 8 shows the averaged results of 100 training runs. Each training run involves performing repeated trials until convergence. Because the different trials execute different numbers of primitive actions, we have just plotted the number of primitive actions on the horizontal axis rather than the number of trials.

The first thing to note is that all forms of MAXQ learning have better initial performance than flat Q learning. This is because of the constraints introduced by the MAXQ hierarchy. For example, while the agent is executing a Navigate subtask, it will never attempt to pickup or putdown the passenger, because those actions are not available to Navigate. Similarly, the agent will never attempt to putdown the passenger until it has first picked up the passenger (and vice versa) because of the termination conditions of the Get and Put subtasks.

The second thing to notice is that without state abstractions, MAXQ-Q learning actually takes longer to converge, so that the Flat Q curve crosses the MAXQ/no abstraction curve. This shows that without state abstraction, the cost of learning the huge number of parameters in the MAXQ representation is not really worth the benefits. We suspect this is a consequence of the model-free nature of the MAXQ-Q algorithm. The MAXQ decomposition represents some information redundantly. For example, the cost of performing a Put subtask is computed both as $C(\mathsf{Root}, s, \mathsf{Get})$ and also as $V(\mathsf{Put}, s)$. A model-based algorithm could compute both of these from a learned model, but MAXQ-Q must learn each of them separately from experience.

The third thing to notice is that with state abstractions, MAXQ-Q converges very quickly to a hierarchically optimal policy. This can be seen more clearly in Figure 9, which focuses on the range of reward values in the neighborhood of the optimal policy. Here we can see that MAXQ with abstractions attains the hierarchically optimal policy after approximately 40,000 steps, whereas flat Q learning requires roughly twice as long to reach the same level. However, flat Q learning, of course, can continue onward and reach optimal performance, whereas with the MAXQ hierarchy, the best hierarchical policy is slow to respond to the "fickle" behavior of the passenger when he/she changes the destination.

The last thing to notice is that with greedy execution, the MAXQ policy is also able to attain optimal performance. But as the execution becomes "more greedy", there is a temporary drop in performance, because MAXQ-Q must learn C values in new regions of the state space that were not visited by the recursively optimal policy. Despite this drop in performance, greedy MAXQ-Q recovers rapidly and reaches hierarchically optimal performance faster than purely-hierarchical MAXQ-Q learning. Hence, there is no added cost—in terms of exploration—for introducing greedy execution.

This experiment presents evidence in favor of three claims: first, that hierarchical reinforcement learning can be much faster than flat Q learning; second, that state abstraction is required by MAXQ-Q learning for good performance; and third, that non-hierarchical execution can produce significant improvements in performance with little or no added exploration cost.

7.2 Kaelbling's HDG Method

The second task that we will consider is a simple maze task introduced by Leslie Kaelbling (1993) and shown in Figure 11. In each trial of this task, the agent starts in a randomly-chosen state and must move to a randomly-chosen goal state using the usual North, South, East, and West operators (we employed deterministic operators). There is a small cost for each move, and the agent must minimize the undiscounted sum of these costs.

Because the goal state can be in any of 100 different locations, there are actually 100 different MDPs. Kaelbling's HDG method starts by choosing an arbitrary set of landmark states and defining a Voronoi partition of the state space based on the Manhattan distances to these landmarks (i.e., two states belong to the same Voronoi cell iff they have the same nearest landmark). The method then defines one subtask for each landmark l. The subtask is to move from any state in the current Voronoi cell *or in any neighboring Voronoi cell* to the landmark l. Optimal policies for these subtasks are then computed.

Once HDG has the policies for these subtasks, it can solve the abstract Markov Decision Problem of moving from each landmark state to any other landmark state using the subtask solutions as macro actions (subroutines). So it computes a value function for this MDP.

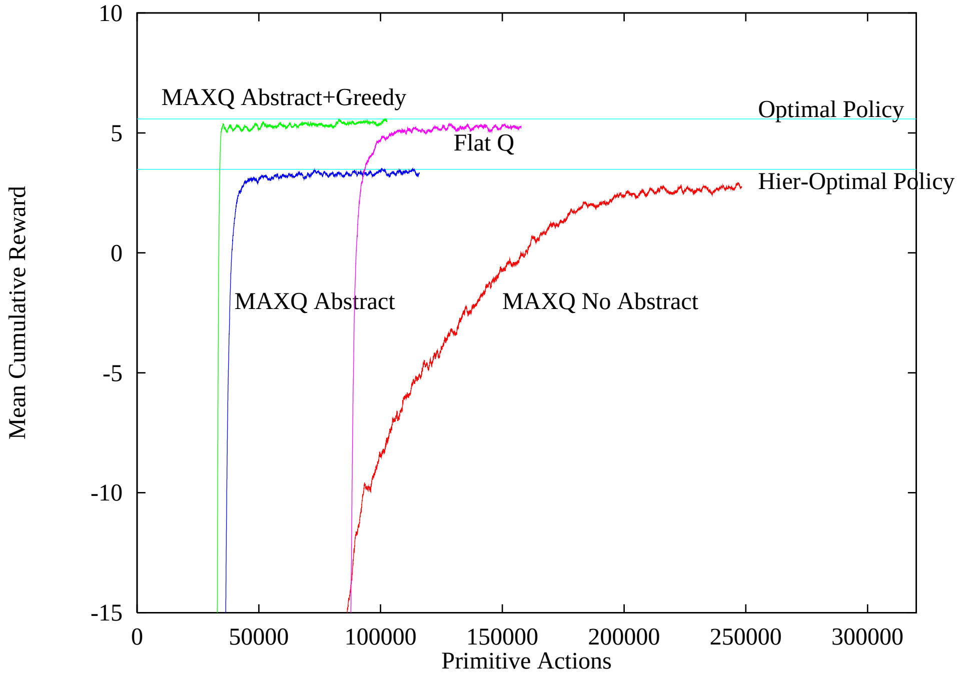


Figure 9: Close-up view of the previous figure. This figure also shows two horizontal lines indicating optimal performance and hierarchically optimal performance in this domain. To make this figure more readable, we have applied a 100-step moving average to the data points (which are themselves the average of 100 runs).

Finally, for each possible destination location g within a Voronoi cell for landmark l, the HDG method computes the optimal policy of getting from l to g.

By combining these subtasks, the HDG method can construct a good approximation to the optimal policy as follows. In addition to the value functions discussed above, the agent maintains two other functions: $NL(s)$, the name of the landmark nearest to state s, and $N(l)$, a list of the landmarks of the cells that are immediate neighbors of cell l. By combining these, the agent can build a list for each state s of the current landmark and the landmarks of the neighboring cells. For each such landmark, the agent computes the sum of three terms:

(t1) the expected cost of reaching that landmark,

(t2) the expected cost of moving from that landmark to the landmark in the goal cell, and

(t3) the expected cost of moving from the goal-cell landmark to the goal state.

Note that while terms (t1) and (t3) can be exact estimates, term (t2) is computed using the landmark subtasks as subroutines. This means that the corresponding path must pass through the intermediate landmark states rather than going directly to the goal landmark.

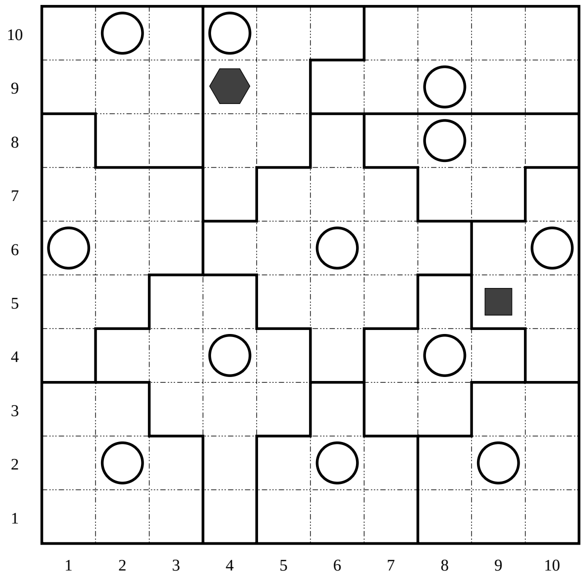


Figure 10: Kaelbling's 10-by-10 navigation task. Each circled state is a landmark state, and the heavy lines show the boundaries of the Voronoi cells. In each episode, a start state and a goal state are chosen at random. In this figure, the start state is shown by the black square, and the goal state is shown by the black hexagon.

Hence, term (t2) is typically an overestimate of the required distance. (Also note that (t3) is the same for all choices of the intermediate landmarks, so it does not need to be explicitly included in the computation of the best action until the agent enters the cell containing the goal.)

Given this information, the agent then chooses to move toward the best of the landmarks (unless the agent is already in the goal Voronoi cell, in which case the agent moves toward the goal state). For example, in Figure 10, term (t1) is the cost of reaching the landmark in row 6, column 6, which is 4. Term (t2) is the cost of getting from row 6, column 6 to the landmark at row 1 column 4 (by going from one landmark to another). In this case, the best landmark-to-landmark path is to go directly from row 6 column 6 to row 1 column 4. Hence, term (t2) is 6. Term (t3) is the cost of getting from row 1 column 4 to the goal, which is 1. The sum of these is $4 + 6 + 1 = 11$. For comparison, the optimal path has length 9.

In Kaelbling's experiments, she employed a variation of Q learning to learn terms (t1) and (t3), and she computed (t2) at regular intervals via the Floyd-Warshall all-sources shortest paths algorithm.

Figure 11 shows a MAXQ approach to solving this problem. The overall task Root, takes one argument g, which specifies the goal cell. There are three subtasks:

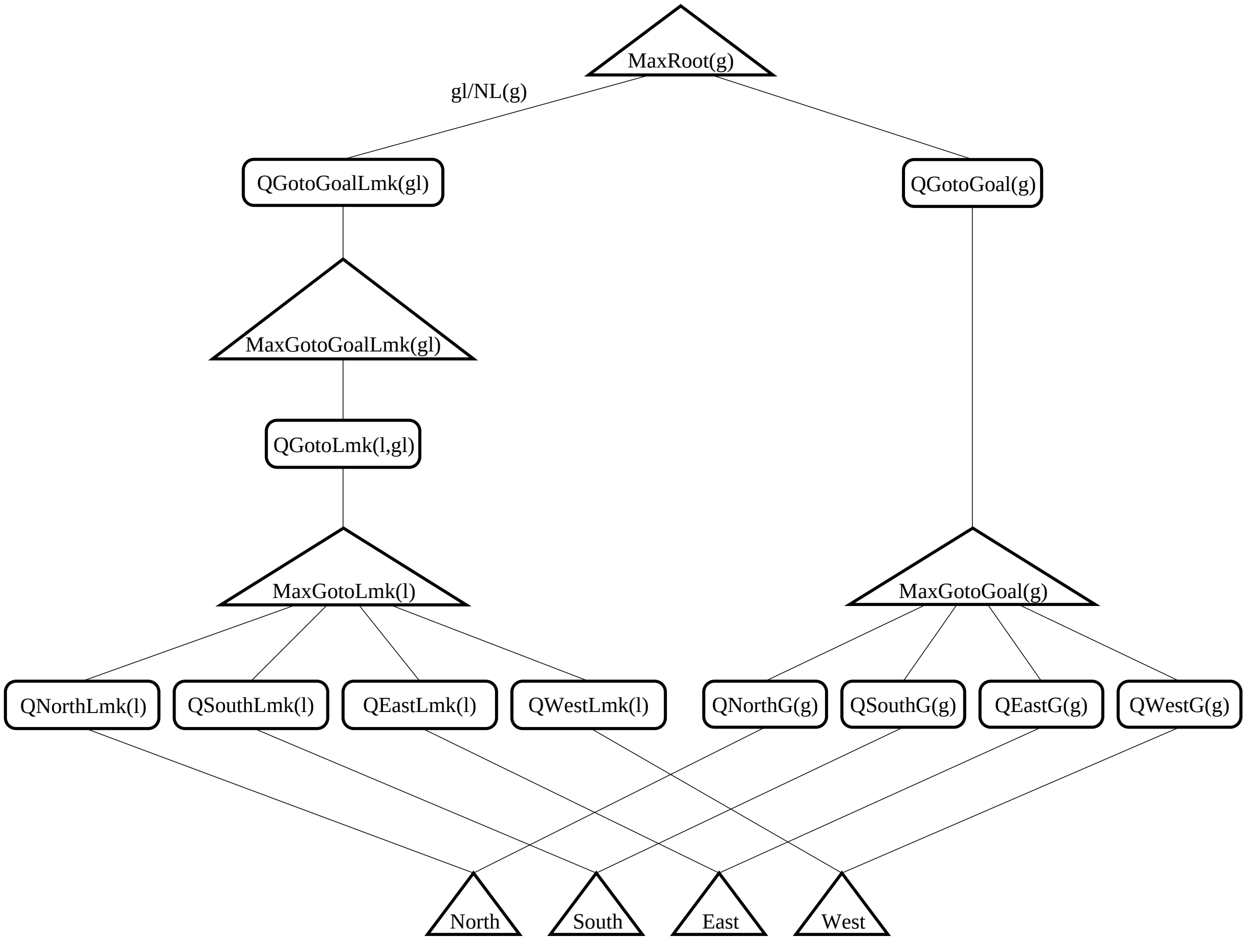


Figure 11: A MAXQ graph for the HDG navigation task.

- GotoGoalLmk, go to the landmark nearest to the goal location. The termination predicate for this subtask is true if the agent reaches the landmark nearest to the goal. The goal predicate is the same as the termination predicate.

- GotoLmk(l), go to landmark l. The termination predicate for this is true if either (a) the agent reaches landmark l or (b) the agent is outside of the region defined by the Voronoi cell for l and the neighboring Voronoi cells, $N(l)$. The goal predicate for this subtask is true only for condition (a).

- GotoGoal(g), go to the goal location g. The termination predicate for this subtask is true if either the agent is in the goal location or the agent is outside of the Voronoi cell $NL(g)$ that contains g. The goal predicate for this subtask is true if the agent is in the goal location.

The MAXQ decomposition is essentially the same as Kaelbling's method, but somewhat redundant. Consider a state where the agent is not inside the same Voronoi cell as the goal g. In such states, HDG decomposes the value function into three terms (t1), (t2), and (t3). Similarly, MAXQ also decomposes it into these same three terms:

- $V(\mathsf{GotoLmk}(l), s, a)$ the cost of getting to landmark l. This is represented as the sum of $V(a, s)$ and $C(\mathsf{GotoLmk}(l), s, a)$.
- $C(\mathsf{GotoGoalLmk}(gl), s, \mathsf{MaxGotoLmk}(l))$ the cost of getting from landmark l to the landmark gl nearest the goal.
- $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk}(gl))$ the cost of getting to the goal location after reaching gl (i.e., the cost of completing the Root task after reaching gl).

When the agent is inside the goal Voronoi cell, then again HDG and MAXQ store essentially the same information. HDG stores $Q(\mathsf{GotoGoal}(g), s, a)$, while MAXQ breaks this into two terms: $C(\mathsf{GotoGoal}(g), s, a)$ and $V(a, s)$ and then sums these two quantities to compute the Q value.

Note that this MAXQ decomposition stores some information twice—specifically, the cost of getting from the goal landmark gl to the goal is stored both as $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk}(gl))$ and as $C(\mathsf{GotoGoal}(g), s, a) + V(a, s)$.

Let us compare the amount of memory required by flat Q learning, HDG, and MAXQ. There are 100 locations, 4 possible actions, and 100 possible goal states, so flat Q learning must store 40,000 values.

To compute quantity (t1), HDG must store 4 Q values (for the four actions) for each state s with respect to its own landmark and the landmarks in $N(NL(s))$. This gives a total of 2,028 values that must be stored.

To compute quantity (t2), HDG must store, for each landmark, information on the shortest path to every other landmark. There are 12 landmarks. Consider the landmark at row 6, column 1. It has 5 neighboring landmarks which constitute the five macro actions that the agent can perform to move to another landmark. The nearest landmark to the goal cell could be any of the other 11 landmarks, so this gives a total of 55 Q values that must be stored. Similar computations for all 12 landmarks give a total of 506 values that must be stored.

Finally, to compute quantity (t3), HDG must store information, for each square inside each Voronoi cell, about how to get to each of the other squares inside the same Voronoi cell. This requires 3,536 values.

Hence, the grand total for HDG is 6,070, which is a huge savings over flat Q learning.

Now let's consider the MAXQ hierarchy with and without state abstractions.

- $V(a, s)$: This is the expected reward of each primitive action in each state. There are 100 states and 4 primitive actions, so this requires 400 values. However, because the reward is constant (-1), we can apply Leaf Irrelevance to store only a single value.
- $C(\mathsf{GotoLmk}(l), s, a)$, where a is one of the four primitive actions. This requires the same amount of space as (t1) in Kaelbling's representation—indeed, combined with $V(a, s)$, this represents exactly the same information as (t1). It requires 2,028 values. No state abstractions can be applied.

- $C(\mathsf{GotoGoalLmk}(gl), s, \mathsf{GotoLmk}(l))$: This is the cost of completing the GotoGoalLmk task after going to landmark l. If the primitive actions are deterministic, then $\mathsf{GotoLmk}(l)$ will always terminate at location l, and hence, we only need to store this for each pair of l and gl. This is exactly the same as Kaelbling's quantity (t2), which requires 506 values. However, if the primitive actions are stochastic—as they were in Kaelbling's original paper—then we must store this value for each possible terminal state of each GotoLmk action. Each of these actions could terminate at its target landmark l or in one of the states bordering the set of Voronoi cells that are the neighbors of the cell for l. This requires 6,600 values. When Kaelbling stores values only for (t2), she is effectively making the assumption that $\mathsf{GotoLmk}(l)$ will never fail to reach landmark l. This is an approximation which we can introduce into the MAXQ representation by our choice of state abstraction at this node.

- $C(\mathsf{GotoGoal}, s, a)$: This is the cost of completing the GotoGoal task after executing one of the primitive actions a. This is the same as quantity (t3) in the HDG representation, and it requires the same amount of space: 3,536 values.

- $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$: This is the cost of reaching the goal once we have reached the landmark nearest the goal. MAXQ must represent this for all combinations of goal landmarks and goals. This requires 100 values. Note that these values are the same as the values of $C(\mathsf{GotoGoal}(g), s, a) + V(a, s)$ for each of the primitive actions. This means that the MAXQ representation stores this information twice, whereas the HDG representation only stores it once (as term (t3)).

- $C(\mathsf{Root}, s, \mathsf{GotoGoal})$. This is the cost of completing the Root task after we have executed the GotoGoal task. If the primitive action are deterministic, this is always zero, because GotoGoal will have reached the goal. Hence, we can apply the Termination condition and not store any values at all. However, if the primitive actions are stochastic, then we must store this value for each possible state that borders the Voronoi cell that contains the goal. This requires 96 different values. Again, in Kaelbling's HDG representation of the value function, she is ignoring the probability that GotoGoal will terminate in a non-goal state. Because MAXQ is an exact representation of the value function, it does not ignore this possibility. If we (incorrectly) apply the Termination condition in this case, the MAXQ representation becomes a function approximation.

In the stochastic case, without state abstractions, the MAXQ representation requires 12,760 values. With safe state abstractions, it requires 12,361 values. With the approximations employed by Kaelbling (or equivalently, if the primitive actions are deterministic), the MAXQ representation with state abstractions requires 6,171 values. These numbers are summarized in Table 6. We can see that, with the unsafe state abstractions, the MAXQ representation requires only slightly more space than the HDG representation (because of the redundancy in storing $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$.

This example shows that for the HDG task, we can start with the fully-general formulation provided by MAXQ and impose assumptions to obtain a method that is similar to HDG. The MAXQ formulation guarantees that the value function of the hierarchical policy will be represented exactly. The assumptions will introduce approximations into the

Table 6: Comparison of the number of values that must be stored to represent the value function using the HDG and MAXQ methods.

HDG item	MAXQ item	HDG values	MAXQ no abs	MAXQ safe abs	MAXQ unsafe abs
	$V(a, s)$	0	400	1	1
(t1)	$C(\mathsf{GotoLmk}(l), s, a)$	2,028	2,028	2,028	2,028
(t2)	$C(\mathsf{GotoGoalLmk}, s, \mathsf{GotoLmk}(l))$	506	6,600	6,600	506
(t3)	$C(\mathsf{GotoGoal}(g), s, a)$	3,536	3,536	3,536	3,536
	$C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$	0	100	100	100
	$C(\mathsf{Root}, s, \mathsf{GotoGoal})$	0	96	96	0
Total Number of Values Required		6,070	12,760	12,361	6,171

value function representation. This might be useful as a general design methodology for building application-specific hierarchical representations. Our long-term goal is to develop such methods so that each new application does not require inventing a new set of techniques. Instead, off-the-shelf tools (e.g., based on MAXQ) could be specialized by imposing assumptions and state abstractions to produce more efficient special-purpose systems.

One of the most important contributions of the HDG method was that it introduced a form of non-hierarchical execution. As soon as the agent crosses from one Voronoi cell into another, the current subtask of reaching the landmark in that cell is "interrupted", and the agent recomputes the "current target landmark". The effect of this is that (until it reaches the goal Voronoi cell), the agent is always aiming for a landmark outside of its current Voronoi cell. Hence, although the agent "aims for" a sequence of landmark states, it typically does not visit many of these states on its way to the goal. The states just provide a convenient set of intermediate targets. By taking these "shortcuts", HDG compensates for the fact that, in general, it has overestimated the cost of getting to the goal, because its computed value function is based on a policy where the agent goes from one landmark to another.

The same effect is obtained by hierarchical greedy execution of the MAXQ graph (which was directly inspired by the HDG method). Note that by storing the NL (nearest landmark) function, Kaelbing's HDG method can detect very efficiently when the current subtask should be interrupted. This technique only works for navigation problems in a space with a distance metric. In contrast, ExecuteHGPolicy performs a kind of "polling", because it checks after each primitive action whether it should interrupt the current subroutine and invoke a new one. An important goal for future research on MAXQ is to find a general purpose mechanism for avoiding unnecessary "polling"—that is, a mechanism that can discover efficiently-evaluable interrupt conditions.

Figure 12 shows the results of our experiments with HDG using the MAXQ-Q learning algorithm. We employed the following parameters: for Flat Q learning, initial values of 0.123, a learning rate of 1.0, initial temperature of 50, and cooling rate of 0.9074; for MAXQ-Q without state abstractions: initial values of -25.123, learning rate of 1.0, initial

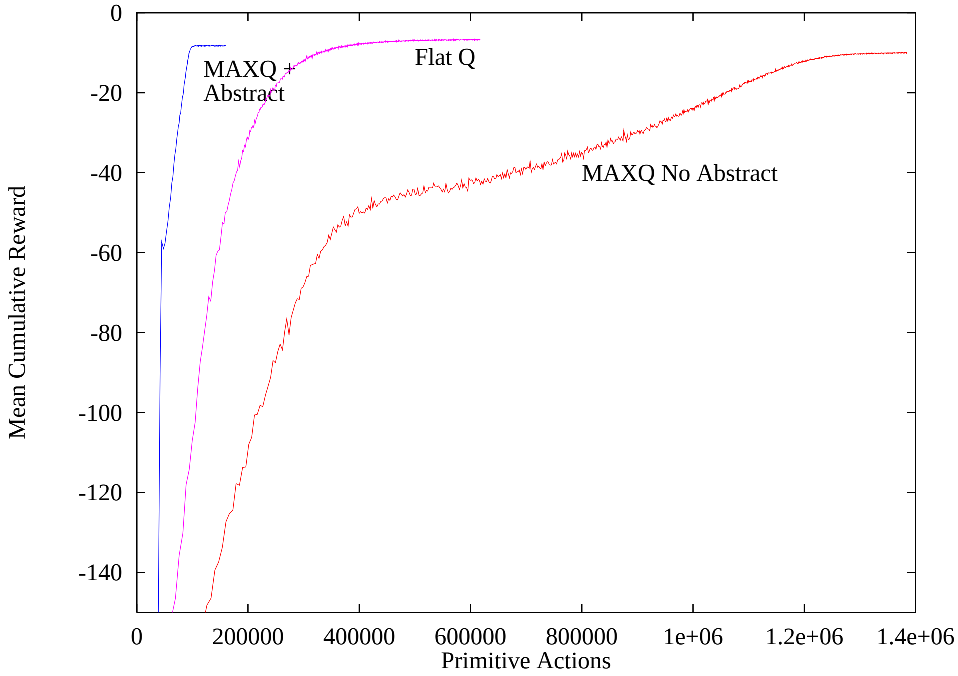


Figure 12: Comparison of Flat Q learning with MAXQ-Q learning with and without state abstraction. (Average of 100 runs.)

temperature of 50, and cooling rates of 0.9074 for MaxRoot, 0.9999 for MaxGotoGoalLmk, 0.9074 for MaxGotoGoal, and 0.9526 for MaxGotoLmk; for MAXQ-Q with state abstractions: initial values of -20.123, learning rate of 1.0, initial temperature of 50, and cooling rates of 0.9760 for MaxRoot, 0.9969 for MaxGotoGoal, 0.9984 for MaxGotoGoalLmk, and 0.9969 for MaxGotoLmk. Hierarchical greedy execution was introduced by starting with 3000 primitive actions per trial, and reducing this every trial by 2 actions, so that after 1500 trials, execution is completely greedy.

The figure confirms the observations made in our experiments with the Fickle Taxi task. Without state abstractions, MAXQ-Q converges much more slowly than flat Q learning. With state abstractions, it converges roughly three times as fast. Figure 13 shows a close-up view of Figure 12 that allows us to compare the differences in the final levels of performance of the methods. Here, we can see that MAXQ-Q with no state abstractions was not able to reach the quality of our hand-coded hierarchical policy—presumably even more exploration would be required to achieve this, whereas with state abstractions, MAXQ-Q is able to do slightly better than our hand-coded policy. With hierarchical greedy execution, MAXQ-Q is able to reach the goal using one fewer action, on the average—so that it approaches the performance of the best hierarchical greedy policy (as computed by value iteration). Notice however, that the best performance that can be obtained by hierarchical greedy execution of the best recursively-optimal policy cannot match optimal performance. Hence, Flat Q

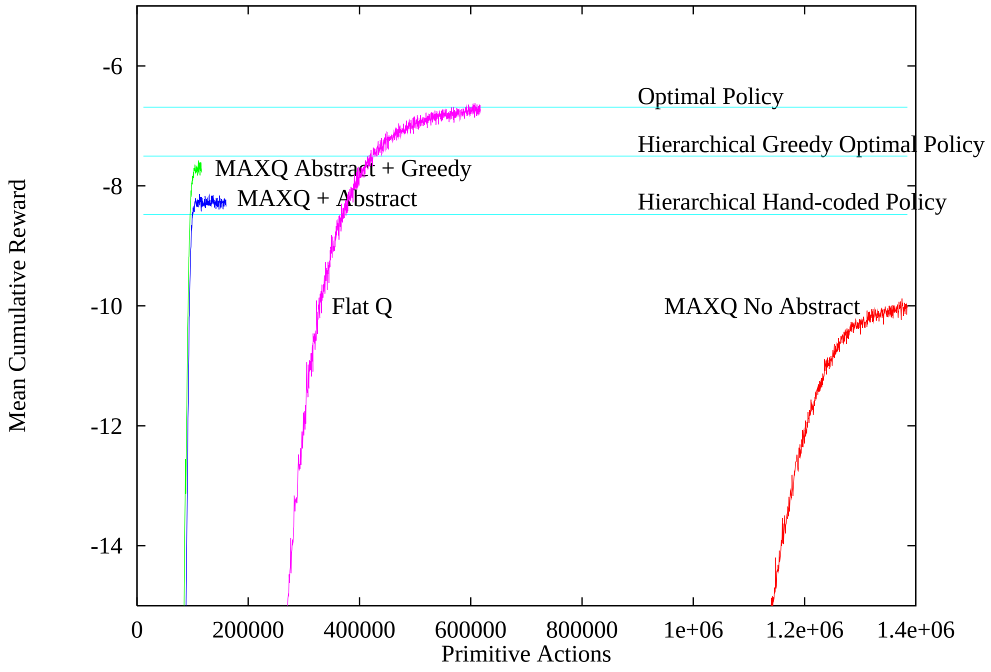


Figure 13: Expanded view comparing Flat Q learning with MAXQ-Q learning with and without state abstraction and with and without hierarchical greedy execution. (Average of 100 runs.)

learning achieves a policy that reaches the goal state, on the average, with about one fewer primitive action. Finally notice that as in the taxi domain, there was no added exploration cost for shifting to greedy execution.

Kaelbling's HDG work has recently been extended and generalized by Moore, Baird and Kaelbling (1999) to any sparse MDP where the overall task is to get from any given start state to any desired goal state. The key to the success of their approach is that each landmark subtask is guaranteed to terminate in a single resulting state. This makes it possible to identify a *sequence* of good intermediate landmark states and then assemble a policy that visits them in sequence. Moore, Baird and Kaelbling show how to construct a hierarchy of landmarks (the "airport" hierarchy) that makes this planning process efficient. Note that if each subtask did not terminate in a single state (as in general MDPs), then the airport method would not work, because there would be a combinatorial explosion of potential intermediate states that would need to be considered.

7.3 Parr and Russell: Hierarchies of Abstract Machines

In his (1998b) dissertation work, Ron Parr considered an approach to hierarchical reinforcement learning in which the programmer encodes prior knowledge in the form of a hierarchy of finite-state controllers called a HAM (Hierarchy of Abstract Machines). The hierarchy

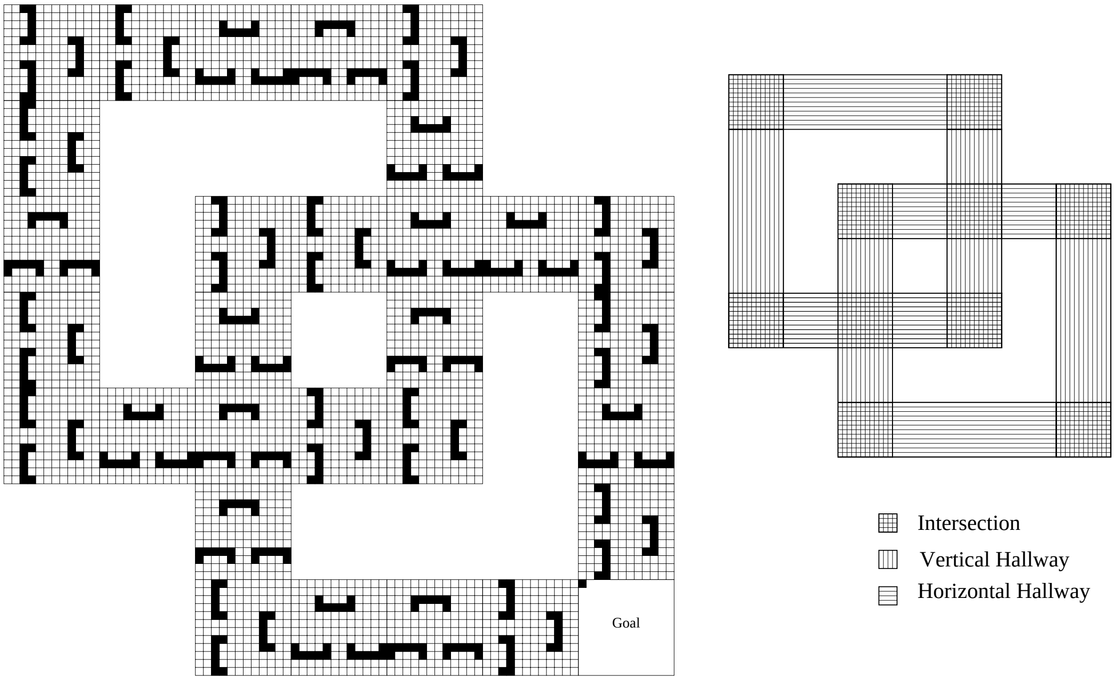


Figure 14: Parr's maze problem (on left). The start state is in the upper left corner, and all states in the lower right-hand room are terminal states. The smaller diagram on the right shows the hallway and intersection structure of the maze.

is executed using a procedure-call-and-return discipline, and it provides a *partial policy* for the task. The policy is partial because each machine can include non-deterministic "choice" machine states, in which the machine lists several options for action but does not specify which one should be chosen. The programmer puts "choice" states at any point where he/she does not know what action should be performed. Given this partial policy, Parr's goal is to find the best policy for making choices in the choice states. In other words, his goal is to learn a hierarchical value function $V(\langle s, m \rangle)$, where s is a state (of the external environment) and m contains all of the internal state of the hierarchy (i.e., the contents of the procedure call stack and the values of the current machine states for all machines appearing in the stack). A key observation is that it is only necessary to learn this value function at choice states $\langle s, m \rangle$. Parr's algorithm does not learn a decomposition of the value function. Instead, it "flattens" the hierarchy to create a new Markov decision problem over the choice states $\langle s, m \rangle$. Hence, it is hierarchical primarily in the sense that the programmer structures the prior knowledge hierarchically. An advantage of this is that Parr's method can find the optimal hierarchical policy subject to constraints provided by the programmer. A disadvantage is that the method cannot be executed "non-hierarchically" to produce a better policy.

Parr illustrated his work using the maze shown in Figure 14. This maze has a large-scale structure (as a series of hallways and intersections), and a small-scale structure (a series of obstacles that must be avoided in order to move through the hallways and intersections).

In each trial, the agent starts in the top left corner, and it must move to any state in the bottom right corner room. The agent has the usual four primitive actions, North, South, East, and West. The actions are stochastic: with probability 0.8, they succeed, but with probability 0.1 the action will move to the "left" and with probability 0.1 the action will move to the "right" instead (e.g., a North action will move east with probability 0.1 and west with probability 0.1). If an action would collide with a wall or an obstacle, it has no effect.

The maze is structured as a series of "rooms", each containing a 12-by-12 block of states (and various obstacles). Some rooms are parts of "hallways", because they are connected to two other rooms on opposite sides. Other rooms are "intersections", where two or more hallways meet.

To test the representational power of the MAXQ hierarchy, we want to see how well it can represent the prior knowledge that Parr is able to represent using the HAM. We begin by describing Parr's HAM for his maze task, and then we will present a MAXQ hierarchy that captures much of the same prior knowledge.[3]

Parr's top level machine, MRoot, consists of a loop with a single choice state that chooses among four possible child machines: MGo($East$), MGo($South$), MGo($West$), and MGo($North$). The loop terminates when the agent reaches a goal state. MRoot will only invoke a particular machine if there is a hallway in the specified direction. Hence, in the start state, it will only consider MGo($South$) and MGo($East$).

The MGo(d) machine begins executing when the agent is in an intersection. So the first thing it tries to do is to exit the intersection into a hallway in the specified direction d. Then it attempts to traverse the hallway until it reaches another intersection. It does this by first invoking an MExitIntersection(d) machine. When that machine returns, it then invokes an MExitHallway(d) machine. When that machine returns, MGo also returns.

The MExitIntersection and MExitHallway machines are identical except for their termination conditions. Both machines consist of a loop with one choice state that chooses among four possible subroutines. To simplify their description, suppose that MGo($East$) has chosen MExitIntersection($East$). Then the four possible subroutines are MSniff($East, North$), MSniff($East, South$), MBack($East, North$), and MBack($East, South$).

The MSniff(d, p) machine always moves in direction d until it encounters a wall (either part of an obstacle or part of the walls of the maze). Then it moves in perpendicular direction p until it reaches the end of the wall. A wall can "end" in two ways: either the agent is now trapped in a corner with walls in both directions d and p or else there is no longer a wall in direction d. In the first case, the MSniff machine terminates; in the second case, it resumes moving in direction d.

The MBack(d, p) machine moves one step backwards (in the direction opposite from d) and then moves five steps in direction p. These moves may or may not succeed, because the actions are stochastic and there may be walls blocking the way. But the actions are carried out in any case, and then the MBack machine returns.

The MSniff and MBack machines also terminate if they reach the end of a hall or the end of an intersection.

[3] The author thanks Ron Parr for providing the details of the HAM for this task.

These finite-state controllers define a highly constrained partial policy. The MBack, MSniff, and MGo machines contain no choice states at all. The only choice points are in MRoot, which must choose the direction in which to move, and in MExitIntersection and MExitHall, which must decide when to call MSniff, when to call MBack, and which "perpendicular" direction to tell these machines to try when they cannot move forward.

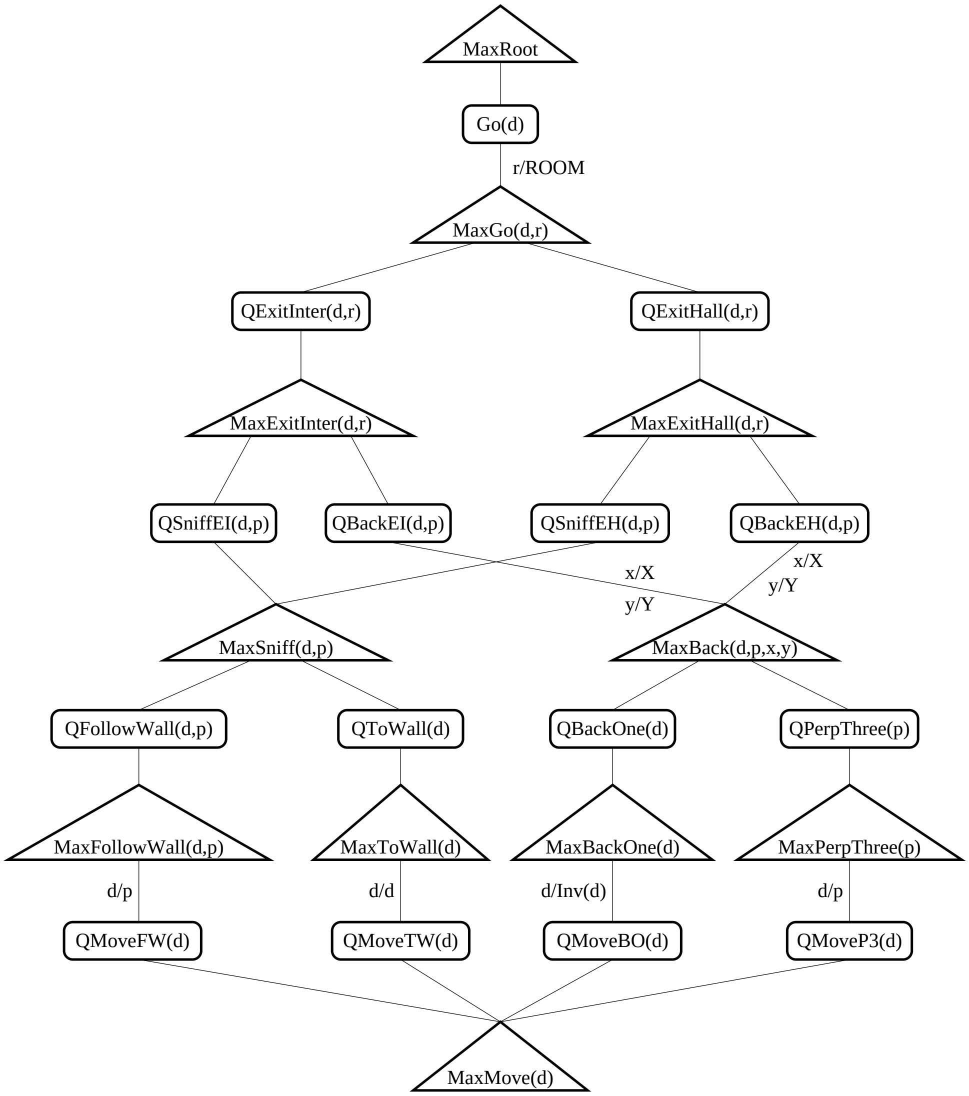


Figure 15: MAXQ graph for Parr's maze task.

Figure 15 shows a MAXQ graph that encodes a similar set of constraints on the policy. The subtasks are defined as follows:

- Root. This is exactly the same as the MRoot machine. It must choose a direction d and invoke Go. It terminates when the agent enters a terminal state. This is also its goal condition (of course).

- Go(d,r). (Go in direction d leaving room r.) The parameter r is bound to an identification number corresponding to the current 12-by-12 "room" in which the agent is located. Go terminates when the agent enters the room at the end of the hallway in direction d or when it leaves the desired hallway (e.g., in the wrong direction). The goal condition for Go is satisfied only if the agent reaches the desired intersection.

- ExitInter(d,r). This terminates when the agent has exited room r. The goal condition is that the agent exit room r in direction d.

- ExitHall(d,r). This terminates when the agent has exited the current hall (into some intersection). The goal condition is that the agent has entered the desired intersection in direction d.

- Sniff(d,r). This encodes a subtask that is equivalent to the MSniff machine. However, Sniff must have two child subtasks, ToWall and FollowWall, that were simply internal states of MSniff. This is necessary, because a subtask in the MAXQ framework cannot contain any internal state, whereas a finite-state controller in the HAM representation can contain as many internal states as necessary. In particular, it can have one state for when it is moving forward and another state for when it is following a wall sideways.

- ToWall(d). This is equivalent to one part of MSniff. It terminates when there is a wall in "front" of the agent in direction d. The goal condition is the same as the termination condition.

- FollowWall(d,p). This is equivalent to the other part of MSniff. It moves in direction p until the wall in direction d ends (or until it is stuck in a corner with walls in both directions d and p). The goal condition is the same as the termination condition.

- Back(d,p,x,y). This attempts to encode the same information as the MBack machine, but this is a case where the MAXQ hierarchy cannot capture the same information. MBack simply executes a sequence of 6 primitive actions (one step back, five steps in direction p). But to do this, MBack must have 6 internal states, which MAXQ does not allow. Instead, the Back subtask has the subgoal of moving the agent at least one square backwards and at least 3 squares in the direction p. In order to determine whether it has achieved this subgoal, it must remember the x and y position where it started to execute, so these are bound as parameters to Back. Back terminates if it achieves the desired change in position or if it runs into walls that prevent it from achieving the subgoal. The goal condition is the same as the termination condition.

- BackOne(d,x,y). This moves the agent one step backwards (in the direction opposite to d. It needs the starting x and y position in order to tell when it has succeeded. It terminates if it has moved at least one unit in direction d or if there is a wall in this direction. Its goal condition is the same as its termination condition.

- PerpThree(p, x, y). This moves the agent three steps in the direction p. It needs the starting x and y positions in order to tell when it has succeeded. It terminates when it has moved at least three units in the direction p or if there is a wall in that direction. The goal condition is the same as the termination condition.

- Move(d). This is a "parameterized primitive" action. It executes one primitive move in direction d and terminates immediately.

From this, we can see that there are three major differences between the MAXQ representation and the HAM representation. First, a HAM finite-state controller can contain internal states. To convert them into a MAXQ subtask graph, we must make a separate subtask for each internal state in the HAM. Second, a HAM can terminate based on an "amount of effort" (e.g., performing 5 actions), whereas a MAXQ subtask must terminate based on some change in the state of the world. It is impossible to define a MAXQ subtask that performs k steps and then terminate regardless of the effects of those steps (i.e., without adding some kind of "counter" to the state of the MDP). Third, it is more difficult to formulate the termination conditions for MAXQ subtasks than for HAM machines. For example, in the HAM, it was not necessary to specify that the MExitHallway machine terminates when it has entered a *different* intersection than the one where the MGo was executed. However, this is important for the MAXQ method, because in MAXQ, each subtask learns its own value function and policy—independent of its parent tasks. For example, without the requirement to enter a *different* intersection, the learning algorithms for MAXQ will always prefer to have MaxExitHall take one step backward and return to the room in which the Go action was started (because that is a much easier terminal state to reach). This problem does not arise in the HAM approach, because the policy learned for a subtask depends on the whole "flattened" hierarchy of machines, and returning to the state where the Go action was started does not help solve the overall problem of reaching the goal state in the lower right corner.

To construct the MAXQ graph for this problem, we have introduced three programming tricks: (a) binding parameters to aspects of the current state (in order to serve as a kind of "local memory" for where the subtask began executing), (b) having a parameterized primitive action (in order to be able to pass a parameter value that specifies which primitive action to perform), and (c) employing "inheritance of termination conditions"—that is, each subtask in this MAXQ graph (but not the others in this paper) inherits the termination conditions of all its ancestor tasks. Hence, if the agent is in the middle of executing a ToWall action when it leaves an intersection, the ToWall subroutine terminates because the ExitInter termination condition is satisfied. This behavior is very similar to the standard behavior of MAXQ. Ordinarily, when an ancestor task terminates, all of its descendent tasks are forced to return *without updating their C values.* With inheritance of termination conditions, on the other hand, the descendent tasks are forced to terminate, but *after updating their C values.* In other words, the termination condition of each child task is the logical disjuntion of all of the termination conditions of its ancestors (plus its own termination condition). This inheritance made it easier to write the MAXQ graph, because the parents did not need to pass down to their children all of the information necessary for the children to define the complete termination and goal predicates.

There are essentially no opportunities for state abstraction in this task, because there are no irrelevant features of the state. There are some opportunities to apply the Shielding and Termination properties, however. In particular, ExitHall(d) is guaranteed to cause its parent task, MaxGo(d), to terminate, so it does not require any stored C values. There are many states where some subtasks are terminated (e.g., Go($East$) in any state where there is a wall on the east side of the room), and so no C values need to be stored.

Nonetheless, even after applying the state elimination conditions, the MAXQ representation for this task requires much more space than a flat representation. An exact computation is difficult, but after applying MAXQ-Q learning, the MAXQ representation required 52,043 values, whereas flat Q learning requires fewer than 16,704 values. Parr states that his method requires only 4,300 values.

To test the relative effectiveness of the MAXQ representation, we compare MAXQ-Q learning with flat Q learning. Because of the very large negative values that some states acquire (particularly during the early phases of learning), we were unable to get Boltzmann exploration to work well—one very bad experience would cause an action to receive such a low Q value, that it would never be tried again. Hence, we experimented with both ϵ-greedy exploration and counter-based exploration. The ϵ-greedy exploration policy is an ordered, abstract GLIE policy in which a random action is chosen with probability ϵ, and ϵ is gradually decreased over time. The counter-based exploration policy keeps track of how many times each action a has been executed in each state s. To choose an action in state s, it selects the action that has been executed the fewest times until all actions have been executed T times. Then it switches to greedy execution. Hence, it is not a genuine GLIE policy. Parr employed counter-based exploration policies in his experiments with this task.

As in the other domains, we conducted several experimental runs (e.g., testing Boltzmann, ϵ-greedy, and counter-based exploration) to determine the best parameters for each algorithm. For Flat Q learning, we chose the following parameters: learning rate 0.50, ϵ-greedy exploration with initial value for ϵ of 1.0, ϵ decreased by 0.001 after each successful execution of a Max node, and initial Q values of -200.123. For MAXQ-Q learning, we chose the following parameters: counter-based exploration with $T = 10$, learning rate equal to the reciprocal of the number of times an action had been performed, and initial values for the C values selected carefully to provide underestimates of the true C values. For example, the initial values for QExitInter were -40.123, because in the worst case, after completing an ExitInter task, it takes about 40 steps to complete the subsequent ExitHall task and hence, complete the Go parent task. Performance was quite sensitive to these initial C values, which is a potential drawback of the MAXQ approach.

Figure 16 plots the results. We can see that MAXQ-Q learning converges about 10 times faster than Flat Q learning. We do not know whether MAXQ-Q has converged to a recursively optimal policy. For comparison, we also show the performance of a hierarchical policy that we coded by hand, but in our hand-coded policy, we used knowledge of contextual information to choose operators, so this policy is surely better than the best recursively optimal policy. HAMQ learning should converge to a policy equal to or slightly better than our hand-coded policy.

This experiment demonstrates that the MAXQ representation can capture most—but not all—of the prior knowledge that can be represented by the HAMQ hierarchy. It also

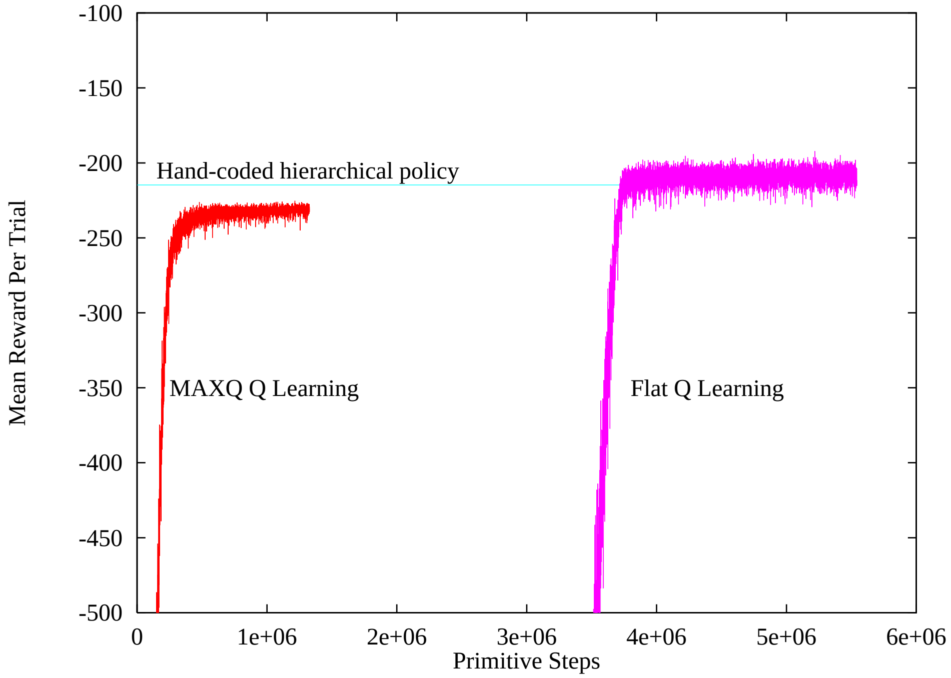


Figure 16: Comparison of Flat Q learning and MAXQ-Q learning in the Parr maze task.

shows that the MAXQ representation requires much more care in the design of the goal conditions for the subtasks.

7.4 Other Domains

In addition to the three domains discussed above, we have developed MAXQ graphs for Singh's (1992) "flag task", the treasure hunter task described by Tadepalli and Dietterich (1997), and Dayan and Hinton's (1993) Feudal-Q learning task. All of these tasks can be easily and naturally placed into the MAXQ framework—indeed, all of them fit more easily than the Parr and Russell maze task.

MAXQ is able to exactly duplicate Singh's work and his decomposition of the value function—while using exactly the same amount of space to represent the value function. MAXQ can also duplicate the results from Tadepalli and Dietterich—however, because MAXQ is not an explanation-based method, it is considerably slower and requires substantially more space to represent the value function.

In the Feudal-Q task, MAXQ is able to give better performance than Feudal-Q learning. The reason is that in Feudal-Q learning, each subroutine makes decisions using only a Q function learned at its own level of the hierarchy—that is, without information about the estimated costs of the actions of its descendents. In contrast, the MAXQ value function decomposition permits each Max node to make decisions based on the sum of its completion function, $C(i, s, j)$, and the costs estimated by its descendents, $V(j, s)$. Of course, MAXQ also supports non-hierarchical execution, which is not possible for Feudal-Q, because it does not learn a value function decomposition.

8. Discussion

Before concluding this paper, we wish to discuss two issues: (a) design tradeoffs in hierarchical reinforcement learning and (b) methods for automatically learning (or at least improving) MAXQ hierarchies.

8.1 Design Tradeoffs in Hierarchical Reinforcement Learning

In the introduction to this paper, we discussed four issues concerning the design of hierarchical reinforcement learning architectures: (a) the method for defining subtasks, (b) the use of state abstraction, (c) non-hierarchical execution, and (d) the design of learning algorithms. In this subsection, we want to highlight a tradeoff between the first two of these issues.

MAXQ defines subtasks using a termination predicate T_i and a pseudo-reward function $\tilde{R}$. There are at least two drawbacks of this method. First, it can be hard for the programmer to define T_i and $\tilde{R}$ correctly, since this essentially requires guessing the value function of the optimal policy for the MDP at all states where the subtask terminates. Second, it leads us to seek a recursively optimal policy rather than a hierarchically optimal policy. Recursively optimal policies may be much worse than hierarchically optimal ones, so we may be giving up substantial performance.

However, in return for these two drawbacks, MAXQ obtains a very important benefit: the policies and value functions for subtasks become *context-free*. In other words, they do not depend on their parent tasks or the larger context in which they are invoked. To understand this point, consider again the MDP shown in Figure 6. It is clear that the *optimal* policy for exiting the left-hand room (the Exit subtask) depends on the location of the goal. If it is at the top of the right-hand room, then the agent should prefer to exit via the upper door, whereas if it is at the bottom of the right-hand room, the agent should prefer to exit by the lower door. However, if we define the subtask of exiting the left-hand room using a pseudo-reward of zero for both doors, then we obtain a policy that is not optimal in either case, but a policy that we can re-use in both cases. Furthermore, this policy *does not depend on the location of the goal.* Hence, we can apply Max node irrelevance to solve the Exit subtask using only the location of the robot and ignore the location of the goal.

This example shows that we obtain the benefits of subtask reuse and state abstraction because we define the subtask using a termination predicate and a pseudo-reward function. The termination predicate and pseudo-reward function provide a barrier that prevents "communication" of value information between the Exit subtask and its context.

Compare this to Parr's HAM method. The HAMQ algorithm finds the best policy consistent with the hierarchy. To achieve this, it must permit information to propagate "into" the Exit subtask (i.e., the Exit finite-state controller) from its environment. But this means that if any state that is reached after leaving the Exit subtask has different values depending on the location of the goal, then these different values will propagate back into the Exit subtask. To represent these different values, the Exit subtask must know the location of the goal. In short, to achieve a hierarchically optimal policy within the Exit subtask, we must (in general) represent its value function using the *entire* state space. State abstractions cannot be employed without losing hierarchical optimality.

We can see, therefore, that there is a direct tradeoff between achieving hierarchical optimality and employing state abstractions. Methods for hierarchical optimality have more freedom in defining subtasks (e.g., using partial policies, as in the HAM approach). But they cannot (safely) employ state abstractions within subtasks, and in general, they cannot reuse the solution of one subtask in multiple contexts. Methods for recursive optimality, on the other hand, must define subtasks using some method (such as pseudo-reward functions for MAXQ or fixed policies for the options framework) that isolates the subtask from its context. But in return, they can apply state abstraction and the learned policy can be reused in many contexts (where it will be more or less optimal).

It is interesting that the iterative method described by Dean and Lin (1995) can be viewed as a method for moving along this tradeoff. In the Dean and Lin method, the programmer makes an initial guess for the values of the terminal states of each subtask (i.e., the doorways in Figure 6). Based on this initial guess, the locally optimal policies for the subtasks are computed. Then the locally optimal policy for the parent task is computed—while holding the subtask policies fixed (i.e., treating them as options). At this point, their algorithm has computed the recursively optimal solution to the original problem, given the initial guesses. Instead of solving the various subproblems sequentially via an offline algorithm as Dean and Lin suggested, we could use the MAXQ-Q learning algorithm.

But the method of Dean and Lin does not stop here. Instead, it computes new values of the terminal states of each subtask based on the learned value function for the entire problem. This allows it to update its "guesses" for the values of the terminal states. The entire solution process can now be repeated to obtain a new recursively optimal solution, based on the new guesses. They prove that if this process is iterated indefinitely, it will converge to the hierarchically optimal policy (provided, of course, that no state abstractions are used within the subtasks).

This suggests an extension to MAXQ-Q learning that adapts the $\tilde{R}$ values online. Each time a subtask terminates, we could update the $\tilde{R}$ function based on the computed value of the terminated state. To be precise, if j is a subtask of i, then when j terminates in state s', we should update $\tilde{R}_j(s')$ to be equal to $\tilde{V}(i, s') = \max_{a'} \tilde{Q}(i, s', a')$. However, this will only work if $\tilde{R}_j(s')$ is represented using the full state s'. If subtask j is employing state abstractions, $x = \chi(s)$, then $\tilde{R}_j(x')$ will need to be the average value of $\tilde{V}(i, s')$, where the average is taken over all states s' such that $x' = \chi(s')$ (weighted by the probability of visiting those states). This is easily accomplished by performing a stochastic approximation update of the form

$$\tilde{R}_j(x') = (1 - \alpha_t)\tilde{R}_j(x') + \alpha_t \tilde{V}(i, s')$$

each time subtask j terminates. Such an algorithm could be expected to converge to the best hierarchical policy consistent with the given state abstractions.

This also suggests that in some problems, it may be worthwhile to first learn a recursively optimal policy using very aggressive state abstractions and then use the learned value function to initialize a MAXQ representation with a more detailed representation of the states. These progressive refinements of the state space could be guided by monitoring the degree to which the values of $\tilde{V}(i, x')$ vary for each abstract state x'. If they have a large variance, this means that the state abstractions are failing to make important distinctions in the values of the states, and they should be refined.

Both of these kinds of adaptive algorithms will take longer to converge than the basic MAXQ method described in this paper. But for tasks that an agent must solve many times in its lifetime, it is worthwhile to have learning algorithms that provide an initial useful solution but then gradually improve that solution until it is optimal. An important goal for future research is to find methods for diagnosing and repairing errors (or sub-optimalities) in the initial hierarchy so that ultimately the optimal policy will be discovered.

8.2 Automated Discovery of Abstractions

The approach taken in this paper has been to rely upon the programmer to design the MAXQ hierarchy including the termination conditions, pseudo-reward functions, and state abstractions. But the results of this paper, particularly concerning state abstraction, suggest ways in which we might be able to automate the construction of the hierarchy.

The main purpose of the hierarchy is to create opportunities for subtask sharing and state abstraction. These are actually very closely related. In order for a subtask to be shared in two different regions of the state space, it must be the case that the value function in those two different regions is identical except for an additive offset. In the MAXQ framework, that additive offset would be the difference in the C values of the parent task. So one way to find reusable subtasks would be to look for regions of state space where the value function exhibits these additive offsets.

A second way would be to search for structure in the one-step probability transition function $P(s'|s, a)$. A subtask will be useful if it enables state abstractions such as Max Node Irrelevance. We can formulate this as the problem of identifying some region of state space such that, conditioned on being in that region, $P(s'|s, a)$ factors according to Equation 17. A top-down divide-and-conquer algorithm similar to decision-tree algorithms might be able to do this.

A third way would be to search for funnel actions by looking for bottlenecks in the state space through which all policies must travel. This would be useful for discovering cases of Result Distribution Irrelevance.

In some ways, the most difficult kinds of state abstractions to discover are those in which arbitrary subgoals are introduced to constrain the policy (and sacrifice optimality). For example, how could an algorithm automatically decide to impose landmarks onto the HDG task? Perhaps by detecting a large region of state space without bottlenecks or variations in the reward function?

The problem of discovering hierarchies is an important challenge for the future, but at least this paper has provided some guidelines for what constitute good state abstractions, and these can serve as objective functions for guiding the automated search for abstractions.

9. Concluding Remarks

This paper has introduced a new representation for the value function in hierarchical reinforcement learning—the MAXQ value function decomposition. We have proved that the MAXQ decomposition can represent the value function of any hierarchical policy under both the finite-horizon undiscounted, cumulative reward criterion and the infinite-horizon discounted reward criterion. This representation supports subtask sharing and re-use, because the overall value function is decomposed into value functions for individual subtasks.

The paper introduced a learning algorithm, MAXQ-Q learning, and proved that it converges with probability 1 to a recursively optimal policy. The paper argued that although recursive optimality is weaker than either hierarchical optimality or global optimality, it is an important form of optimality because it permits each subtask to learn a locally optimal policy while ignoring the behavior of its ancestors in the MAXQ graph. This increases the opportunities for subtask sharing and state abstraction.

We have shown that the MAXQ decomposition creates opportunities for state abstraction, and we identified a set of five properties (Max Node Irrelevance, Leaf Irrelevance, Result Distribution Irrelevance, Shielding, and Termination) that allow us to ignore large parts of the state space within subtasks. We proved that MAXQ-Q still converges in the presence of these forms of state abstraction, and we showed experimentally that state abstraction is important in practice for the successful application of MAXQ-Q learning—at least in the Taxi and HDG tasks.

The paper presented two different methods for deriving improved non-hierarchical policies from the MAXQ value function representation, and it has formalized the conditions under which these methods can improve over the hierarchical policy. The paper verified experimentally that non-hierarchical execution gives improved performance in the Fickle Taxi Task (where it achieves optimal performance) and in the HDG task (where it gives a substantial improvement).

Finally, the paper has argued that there is a tradeoff governing the design of hierarchical reinforcement learning methods. At one end of the design spectrum are “context free” methods such as MAXQ-Q learning. They provide good support for state abstraction and subtask sharing but they can only learn recursively optimal policies. At the other end of the spectrum are “context-sensitive” methods such as HAMQ, the options framework, and the early work of Dean and Lin. These methods can discover hierarchically optimal policies (or, in some cases, globally optimal policies), but their drawback is that they cannot easily exploit state abstractions or share subtasks. Because of the great speedups that are enabled by state abstraction, this paper has argued that the context-free approach is to be preferred—and that it can be relaxed as needed to obtain improved policies.

Acknowledgements

The author gratefully acknowledges the support of the National Science Foundation under grant number IRI-9626584, the Office of Naval Research under grant number N00014-95-1-0557, the Air Force Office of Scientific Research under grant number F49620-98-1-0375, and the Spanish government under their program of Estancias de Investigadores Extranjeros en Regimen de Año Sabatico en España. In addition, the author is indebted to many colleagues for helping develop and clarify the ideas in this paper including Valentina Zubek, Leslie Kaelbling, Bill Langford, Wes Pinchot, Rich Sutton, Prasad Tadepalli, and Sebastian Thrun. I particularly want to thank Eric Chown for encouraging me to study Feudal reinforcement learning, Ron Parr for providing the details of his HAM machines, and Sebastian Thrun for encouraging me to write a single comprehensive paper. I also thank Andrew Moore (the action editor), Valentina Zubek, and the two sets of anonymous reviewers of previous drafts of this paper for their suggestions and careful reading, which have improved the paper immeasurably.

References

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1104–1111.

Currie, K., & Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, *52*(1), 49–86.

Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems, 5*, pp. 271–278. Morgan Kaufmann, San Francisco, CA.

Dean, T., & Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. Tech. rep. CS-95-10, Department of Computer Science, Brown University, Providence, Rhode Island.

Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Fifteenth International Conference on Machine Learning*, pp. 118–126. Morgan Kaufmann.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17–37.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, pp. 220–229 San Francisco, CA. Morgan Kaufmann Publishers.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

Jaakkola, T., Jordan, M. I., & Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, *6*(6), 1185–1201.

Kaelbling, L. P. (1993). Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 167–173 San Francisco, CA. Morgan Kaufmann.

Kalmár, Z., Szepesvári, C., & Lörincz, A. (1998). Module based reinforcement learning for a real robot. *Machine Learning*, *31*, 55–85.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 923–928 Boston, MA. AAAI Press.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, *26*(1), 35–77.

Lin, L.-J. (1993). *Reinforcement learning for robots using neural networks.* Ph.D. thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA.

Moore, A. W., Baird, L., & Kaelbling, L. P. (1999). Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1316–1323 San Francisco. Morgan Kaufmann.

Parr, R. (1998a). Flexible decomposition algorithms for weakly coupled Markov decision problems. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, pp. 422–430 San Francisco, CA. Morgan Kaufmann Publishers.

Parr, R. (1998b). *Hierarchical control and learning for Markov decision processes.* Ph.D. thesis, University of California, Berkeley, California.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, Vol. 10, pp. 1043–1049 Cambridge, MA. MIT Press.

Pearl, J. (1988). *Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA.

Rummery, G. A., & Niranjan, M. (1994). Online Q-learning using connectionist systems. Tech. rep. CUED/FINFENG/TR 166, Cambridge University Engineering Department, Cambridge, England.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*(2), 115–135.

Singh, S., Jaakkola, T., Littman, M. L., & Szepesvári, C. (1998). Convergence results for single-step on-policy reinforcement-learning algorithms. Tech. rep., University of Colorado, Department of Computer Science, Boulder, CO. To appear in *Machine Learning.*

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*, 323–339.

Sutton, R. S., Singh, S., Precup, D., & Ravindran, B. (1999). Improved switching among temporally abstract actions. In *Advances in Neural Information Processing Systems*, Vol. 11, pp. 1066–1072. MIT Press.

Sutton, R., & Barto, A. G. (1998). *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA.

Sutton, R. S., Precup, D., & Singh, S. (1998). Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. Tech. rep., University of Massachusetts, Department of Computer and Information Sciences, Amherst, MA. To appear in *Artificial Intelligence.*

Tadepalli, P., & Dietterich, T. G. (1997). Hierarchical explanation-based reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 358–366 San Francisco, CA. Morgan Kaufmann.

Tambe, M., & Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, *68*(1), 155–199.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Ph.D. thesis, King's College, Oxford. (To be reprinted by MIT Press.).

Watkins, C. J., & Dayan, P. (1992). Technical note Q-Learning. *Machine Learning*, *8*, 279.

On Replacement Models Via a Fuzzy Set Theoretic Framework

Augustine O. Esogbue, *Senior Member, IEEE,* and Warren E. Hearnes, II

***Abstract*—Uncertainty is present in virtually all replacement decisions due to unknown future events, such as revenue streams, maintenance costs, and inflation. Fuzzy sets provide a mathematical framework for explicitly incorporating imprecision into the decision making model, especially when the system involves human subjectivity. This paper illustrates the use of fuzzy sets and possibility theory to explicitly model uncertainty in replacement decisions via fuzzy variables and numbers. In particular, a fuzzy set approach to *economic life of an asset* calculation as well as a finite-horizon single asset replacement problem with multiple challengers is discussed. Because the use of triangular fuzzy numbers provides a compromise between computational efficiency and realistic modeling of the uncertainty, this discussion emphasizes fuzzy numbers. The algorithms used to determine the optimal replacement policy incorporate fuzzy arithmetic, dynamic programming (DP) with fuzzy rewards, the vertex method, and various ranking methods for fuzzy numbers. A brief history of replacement analysis, current conventional techniques, the basic concepts of fuzzy sets and possibility theory, and the advantages of the fuzzy generalization are also discussed.**

***Index Terms*—Decision making under uncertainty, fuzzy numbers, fuzzy sets, possibility theory, replacement analysis.**

I. Economic Decision Analysis

ECONOMIC decision analysis is a useful tool, offering individuals and organizations the techniques to model economic decision making problems, such as maintenance and replacement decisions, and determine an optimal decision. However, the accuracy of the model determines the validity of the conclusion. In many cases, the assumption of certainty in many models is made not so much for validity but for the need to obtain simpler and more readily solvable formulations. Essentially, the tradeoff is between an *inaccurate but solvable* model and a *more accurate but potentially unsolvable* one. In most real-world systems, however, there are elements of uncertainty in the process or its parameters, which may lack precise definition or precise measurement, especially when the system involves human subjectivity.

When developing a model of a system with uncertainty, the decision maker can either *ignore* the uncertainty, *implicitly acknowledge* it, or *explicitly model* it. Ignoring the uncertainty usually results in a deterministic model of the process with precise values for all parameters. Implicitly acknowledging the uncertainty may still result in a deterministic model in which sensitivity analysis or discount factors can be used to get an idea of how this uncertainty affects the outcome. Lastly, the decision maker can explicitly model the uncertainty using specific paradigms, such as interval analysis, possibility theory, probability theory, or evidence theory [3].

The proper paradigm depends on the nature of the uncertainty. When the probabilities are specified for the outcomes, the theory of Von Neumann and Morgenstern [34] provides the tools necessary to determine the optimal decision. However, in many cases, these probabilities are neither defined nor directly attainable. Under these circumstances, other theories are needed. The most common choice is the use of subjective probability distributions and the theory of choice due to Savage [34]. However, considerable debate on the use of subjective probabilities exists and is well documented in the literature [6], [16], [23], [25], [27]. From a psychological standpoint, the methods used to elicit these subjective probabilities and the validity of the subjective probabilities themselves have been the focus of research led by Tversky and Kahneman [37]–[39]. Their studies show that the heuristics employed to assess probabilities and predict values can sometimes lead to "severe and systematic errors" [38].

Because humans do not think naturally in probabilistic terms, they tend to find the notions of fuzzy sets and their linguistic based approaches more user-friendly and appealing. We may view fuzzy set theory as a generalization of classical set theory since it provides us with a mathematical tool for describing sets that have no sharp transition from membership to nonmembership. Membership in a fuzzy set is defined by a generalized version of the classical indicator function called a membership function. Fuzzy sets allow the definition of vague or imprecise concepts, such as "*approximately* 1000," where, for example, 1000 would have a membership of 1.0 and 975 would have a membership of 0.5 (see Fig. 1). This theory has been developed and successfully applied to numerous areas, such as control and decision making, engineering, and medicine. Its application to economic analysis is natural due to the uncertainty inherent in many financial and investment decisions. As noted earlier, it provides a precise mathematical language to model uncertainty due to vagueness and imprecision in events or statements describing a system. More information on fuzzy set theory, particularly fundamental concepts, such as fuzzy numbers, which are invoked in our presentation, is included in the Appendix.

Manuscript received May 7, 1997; revised December 17, 1997. This work was supported in part by the National Science Foundation under Grant ECS-9216004, the National Aeronautics and Space Administration under Grant NAG 9-726, and the Electric Power Research Institute under Grant RP 8030-16.

The authors are with the Intelligent Systems and Control Laboratory, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0205 USA (e-mail: aesogbue@isye.gatech.edu).

Publisher Item Identifier S 1094-6977(98)07473-2.

II. Replacement Analysis

One of the most practical and topical areas of engineering economics is replacement analysis. Mathematical models and analysis methods are used to determine the sequence of replacement decisions that provides a required service for a specified time horizon in an optimal manner. It is assumed that maintenance and replacement decisions occur on a periodic basis. The decision maker chooses from various options, such as to keep, overhaul, or perform preventive maintenance on the existing asset or replace it with a new/used asset. Any sequence of decisions is called a *replacement policy*, and any sequence that optimizes some performance measure, such as net present value or annual equivalent cost, is an *optimal replacement policy*.

In replacement analysis, the *economic life of an asset* determines the replacement cycle that gives the minimum annual equivalent cost (MAEC) of operating a single asset over an infinite horizon [35]. Dynamic programming (DP), with discounting to put more emphasis on short-term income, is an acknowledged tool for the determination of the optimal replacement policy for more general replacement models [18]. Using the DP approach, some of the restrictive assumptions of the economic life method can be relaxed and still produce a computationally feasible solution algorithm. Early pioneers in the use of DP for finite-horizon equipment replacement problems were Bellman and Wagner. Bellman [4], [5] was the first to formulate the replacement problem as a dynamic program. Optimal replacement policies were proposed first for the case with no technological change and later assuming technological improvement. Bellman formulated a discounted DP version of the *economic life of an asset* model and determined analytically the optimal age T to replace the asset.

In the more challenging technological improvement version, the revenue, upkeep costs, and replacement costs are assumed to be functions of the date the asset is installed as well as its age with respect to installation. Wagner [41] formulated the replacement problem as a network and solved for the shortest path, which corresponded to the minimum outlay. Terborgh [36] included linear obsolescence in his formulation, while Alchian [1] allowed operating revenues and operating costs to increase linearly with time. Oakford [30] modeled increasing revenues and data. Dreyfus [31] modeled technological change in revenue, maintenance, and replacement costs using bounded exponential functions. The Dynamic Replacement Economy Decision Model (DREDM) developed in [31] is a generalization of Wagner's DP model that allows for multiple challengers and time-varying parameters.

Replacement models of great interest and relevance to this paper are those that model *uncertainty*. Dreyfus and Law [31] treat the replacement problem in which determinism yields to stochasticity. Their model includes the probability of a catastrophic failure in the asset being used as well as an uncertain net operating cost that is modeled by another probability distribution. The DP algorithm determines the minimum expected cost for the process. The Stochastic Replacement Economy Decision Model (SRE) presented by Lohmann [29] is a stochastic generalization of the DREDM. The assumption that the cash flows and relationships are known with certainty was relaxed, and the component cash flows are modeled as triangular probability distributions based on the decision maker's subjective judgment. The solution for this model is generated through Monte Carlo simulation, which determines the probability that each asset is the optimal choice at time zero as well as the probability distribution of the optimal net present value of the policy.

A tacit assumption implicit in the foregoing models is that uncertainty in the replacement decision can be fully modeled either deterministically or stochastically. This is not, however, always the case. Limiting replacement models to these two approaches either ignores the uncertainty or assumes that all uncertainty is probabilistic in nature and that the probabilistic information is fully known. Categorically classifying all uncertainty as randomness may not be reasonable or adequate.

In recent times, the debate concerning the use of nonprobabilistic uncertainty and, specifically, fuzzy sets has surfaced in the area of economic analysis [3], [8]–[11], [19], [20], [42]. The replacement decisions made at each time period are based not only on the current cash flows, but also on projected future cash flows of all possible assets [29]. Therefore, uncertainty in these cash flows can have a pronounced effect on the optimal replacement policy. A fuzzy set theoretic approach, as described in the sequel, may lead to more informed replacement decisions when the assumptions for a probabilistic approach are not met. For the deterministic case, a fuzzy set theoretic approach using fuzzy numbers is equivalent to multivariable sensitivity analysis and immediately provides both the deterministic optimal value and the possible range due to uncertainty.

III. Fuzzy Concepts in Cash Flow Analysis

Cash flows, the basic variable in replacement decisions, are used by managers and financial analysts to measure the streams of money going into and flowing out of a particular organization's operation [35]. Traditionally, cash flows are treated as either deterministic or stochastic. However, as shown in simulation studies [32], uncertain information in estimating these cash flows can limit the value of the analysis. Errors in deterministic cash flow estimations can skew the results of the analysis. Similarly, the use of subjective probability distributions as the only measure of uncertainty is of concern. In addition to failing to capture all forms of uncertainty, they generally cannot be verified and the required historical information for generating frequency-based probability distributions is not always available.

The fundamental types of uncertainty, *nonspecificity*, *fuzziness*, and *strife*, are examined by Klir and Yuan in [24]. Uncertainty occurs in replacement and maintenance decisions in various ways. Of particular interest are nonspecificity and fuzziness that may factor into the estimates of the aggregate cash flows, purchase prices/salvage values, minimum attractive rate of return (MARR), or physical lifetimes of the assets. This is especially true when these variables are based on the estimates provided by experts via the use of such natural language statements as "*approximately* $1000." Using fuzzy

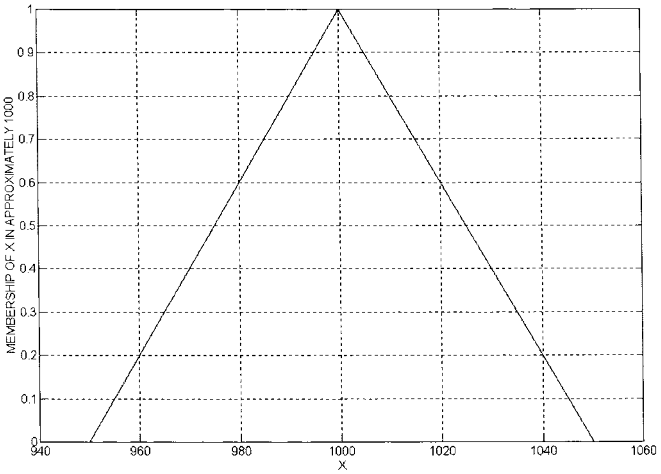


Fig. 1. TFN representing the expression "*approximately* 1000."

variables, we can represent this vagueness and imprecision. In this presentation, however, these vague quantities will be represented using triangular fuzzy numbers (TFN's).

Several basic concepts of fuzzy sets applicable to the foregoing forms of uncertainty include *fuzzy sets*, *α-level sets*, *convexity*, and *TFN's*. For completeness, these definitions and the foundations of fuzzy numbers and possibility theory are briefly reviewed in the Appendix. Refer to [45] and [46] for a more complete review. Fuzzy numbers are fuzzy sets defined on the set of real numbers, generally used to represent vague expressions, such as "*about* 20" or "*approximately* 1000," used often in the description of uncertain economic decision systems. TFN's are a special type fuzzy number that simplify the arithmetic operations considerably and are used in the models developed in this paper. Fig. 1 is a TFN representation of the expression "*approximately* 1000." A TFN is a fuzzy number $\tilde{M}_T = [l, m, r]$ with the membership function

$$\mu_{\tilde{M}_T} = \begin{cases} (1/(m-l))(x-l), & \text{for } x \in (l, m] \\ (1/(r-m))(r-x), & \text{for } x \in (m, r) \\ 0, & \text{otherwise.} \end{cases} \qquad (1)$$

A. Nonprobabilistic Methods in Cash Flow Analysis

Research into fuzzy versions of cash flows began with Ward [42], who defines them as trapezoidal fuzzy numbers in solving a fuzzy present worth problem. Buckley [8] used fuzzy numbers to develop fuzzy net present value (FPV) and fuzzy net future value (FFV) with fuzzy interest rates over a period of n years, where n may also be fuzzy set. Buckley developed fuzzy equivalents to continuous interest payments, the effective rate of interest, and regular annuities as well. Restricting the fuzzy cash flows to positive fuzzy numbers allows the multiplication operation to be distributive over addition. The fuzzy number of time periods produces nonlinearities that make computations more complex. Li Calzi [28] provided an axiomatic development for the fuzzy extension of financial mathematics with a desire to maintain consistency in the calculations. He examined two classes of fuzzy quantities, compact fuzzy intervals and invertible fuzzy intervals, and proved general theorems regarding consistency.

Two of the most recent and practical applications of nonprobabilistic uncertainty to economic analysis are given in [10] and [11]. Choobineh and Behrens [11] call attention to the use of intervals and possibility theory in economic analysis. The weak distributivity of interval arithmetic is noted, but a technique called the vertex method [12] is utilized to bypass this problem in interval and fuzzy arithmetic. Their approach to modeling cash flows as fuzzy intervals is similar to Ward's. Chiu and Park [10] use fuzzy numbers in cash flow analysis and provide a good survey of the major methods for ranking mutually exclusive fuzzy projects. The cash flows are modeled as TFN's, and the linear approximation to the product of two TFN's is investigated. The present worth of a fuzzy project is also examined. Their resultant formulation of a fuzzy present worth is

$$\mathrm{PW} = \sum_{t=0}^{n} \left\{ \frac{P_t}{\prod_{s=0}^{t} (1 + R_s)} \right\} \qquad (2)$$

where P_t is a positive or negative TFN representing the cash flow at the end of time t, n is the number of evaluation periods, and R_s is the nonnegative TFN representing the discount rate at the end of time s. Extending these ideas to replacement analysis, Hearnes [20] formulated fuzzy versions of the economic life of an asset model and the finite single asset replacement problem. This work is used as a point of departure for the discussions that follow.

B. Fuzzy Arithmetic and Interval Analysis

Fuzzy numbers represent vague notions of precise quantities. It is essential to be able to perform algebraic operations on them. Fuzzy arithmetic is based on the extension principle introduced by Zadeh [44]. The arithmetic operations of addition, subtraction, multiplication, and division developed in [14] and [15] are particularly useful when modeling and analyzing cash flows. The algebraic operations for TFN's are specifically reviewed. The choice of TFN's is in part due to their simplified algebraic operations. However, the set of TFN's is not closed under the operations of multiplication and division. The effect of using a linear (TFN) approximation, which is studied thoroughly in [22], is not significant. The TFN approximation to the multiplication operation is used in the following models for computational simplicity.

Fuzzy numbers are a family of nested intervals [11] that correspond to levels of "confidence" by the decision maker and therefore are closely related to interval analysis. However, like interval arithmetic, the multiplication operation for fuzzy numbers is only weakly distributive over addition. This presents a problem when modeling with fuzzy numbers since the outcome can depend on the form of the equation used. There are some special cases, however, where the multiplication operation is distributive over addition. If $\tilde{M}_1, \tilde{M}_2, \tilde{M}_3$ are fuzzy numbers, the multiplication operation is distributive over addition (i.e., $\tilde{M}_1 \cdot (\tilde{M}_2 + \tilde{M}_3) = \tilde{M}_1 \cdot \tilde{M}_2 + \tilde{M}_1 \cdot \tilde{M}_3$) when [15]:

1) $\tilde{M}_1$ is a real number, i.e., scalar multiplication is distributive over addition;
2) $\tilde{M}_2$ and $\tilde{M}_3$ are both positive or both negative;
3) $\tilde{M}_2 = -\tilde{M}_2$ and $\tilde{M}_3 = -\tilde{M}_3$.

When conditions 1)–3) are not met, a procedure called the *vertex method* [12] preserves the distributivity of multiplication over addition. The price, however, is an exponential increase in the number of computations.

C. Some Sources of Uncertainty in Replacement Analysis

Several aspects of replacement analysis contain imprecision and vagueness that warrant further discussion. We postulate that some of these variables, such as the physical lifetime of an asset, aggregate cash flow estimates, and MARR, significantly impact the optimal replacement policy.

The physical lifetime of some assets may not be known with certainty, yet this is tacitly assumed and treated as deterministic in many models. In some situations, if enough information is known, it is appropriate to treat it probabilistically. However, in cases in which the asset is a new technology or model, this information is not generally available, and such an approach must be considered suspect. In this case, it is instructive to treat the uncertainty in this variable through the use of fuzzy DP models but particularly those that allow stochastic or fuzzy termination times [8], [17], [21]. For example, for an older asset, the historical information about failures may be known and a probability distribution for failure can be derived. However, for an asset with new technology, an estimate of physical lifetime may be a fuzzy set, such as "*about* five years" or "*more or less* ten years." In each of these cases, the decision space has an uncertain boundary that affects the overall decision policy. Stochastic and fuzzy DP [17] provide methods for dealing with this type of uncertainty. Similarly, there may be uncertainty in the actual horizon N of the project that may be either stochastic or fuzzy. For example, the project duration or asset life may be determined if the state of the asset reaches some imprecisely definable point.

Another source of uncertainty in replacement and maintenance decisions is the estimation of the aggregate cash flow for each time period. In previous models, aggregate cash flows were treated as either deterministic or stochastic variables, but errors in these can lead to skewed analysis [32]. Subjective probability distributions generally cannot be verified, while the required historical information for generating frequency-based probability distributions is not generally available. In these cases, it may be more appropriate to define the aggregate cash flows as possibility distributions based on a decision maker's opinion or expert judgment.

The MARR, which is usually used for project evaluation and comparison, is also another variable that may realistically possess forms of uncertainty [35]. However, in classical approaches, this either is not addressed or erroneously assumed to be well known or deterministic. The selection of the proper MARR plays an important role in the outcome of the maintenance and replacement decisions. There are a number of ways to determine a corporation's MARR, such as the use of the Delphi method involving its directors or some chosen mathematical formula. However, the MARR can be investment or management dependent. Because of the uncertainties characteristic of investment and management decision processes, it is inevitable that any MARR thus determined is imprecise or fraught with uncertainties. Variation in the MARR and its effect on the optimal policy are vital pieces of information to decision makers. These may be better modeled as a fuzzy variable or fuzzy number.

A number of engineering economic studies discuss the incorporation of inflation and inflation rate in their models. It is tacitly assumed or conceded that the measurement of this variable is precise. This, however, is not the case. We know that there is a considerable degree of uncertainty due to the way that it is measured. For example, the Consumer Price Index presently used by the United States Government is now under review due to the concern expressed by certain economists that the "basket" of goods and services it uses may not accurately reflect the *true* inflation (see, for example, [33]). The lack of specificity or precision involved in the measurement of the inflation rate may necessitate the injection of fuzzy modeling, such as the use of fuzzy numbers to represent it.

IV. Economic Life of an Asset Model

In some replacement decisions, an asset is required for a long period of time. In these cases, an infinite horizon can be assumed and the decision variable becomes the life of the asset, commonly called the *economic life of an asset*. The chosen replacement cycle is the cycle corresponding to the minimum annual equivalent (AE) cost of owning and operating the asset [35]. An infinite sequence of replacements and stationary cash flows (with respect to installation time) is assumed.

The general deterministic n-period replacement cycle gives the following AE cost:

$$\begin{aligned}\mathrm{AE}_n(i) = &(P - S_n)(A/P, i, n) + S_n i \\ &+ (A/P, i, n) \sum_{n'=1}^{n} (C_{n'}(P/F, i, n'))\end{aligned} \tag{3}$$

where

i	MARR;
P	initial purchase price;
S_n	salvage value at end of period n;
$C_{n'}$	aggregate cash outflow at end of period n';
$(A/P, i, n)$	capital recovery factor;
$(P/F, i, n)$	present worth factor.

Two significant factors determining the optimal replacement cycle are the *aggregate cash flows* at each time period and MARR. The MARR is set by the organization and is considered a crisp (deterministic) number in the model. The future aggregate cash flows and salvage values, however, are a source of considerable uncertainty and are modeled as TFN's. These parameters are represented as fuzzy versions of their original counterparts by $\tilde{C}_{n'}$ and $\tilde{S}_n$, respectively. The decision maker determines a best, worst, and most likely estimate for each. This method of elicitation, which is quick, has been used previously in replacement analysis [29].

Generalizing to consider fuzzy cash flows and salvage values, we obtain the Possibilistic Economic Life of an Asset Model (PELAM) [20]. The fuzzy economic life of an asset is defined as the replacement cycle n corresponding to the minimum *fuzzy AE* (FAE) of all possible replacement cycles. The traditional model in (3) is manipulated into a *proper representation*—such that all fuzzy numbers appear only once in the equation

$$\begin{aligned}\mathrm{FAE}_n(i) = {} & P(A/P,i,n) + \tilde{S}_n(i - (A/P,i,n)) \\ & + (A/P,i,n) \sum_{n'=1}^{n} (\tilde{C}_{n'}(P/F,i,n') \qquad (4)\end{aligned}$$

where

$\tilde{S}_n$ TFN representing the "salvage value at end of period n;"

$\tilde{C}_{n'}$ TFN representing the "aggregate cash flow at end of period n'."

The operations used in PELAM are scalar multiplication, addition, and subtraction. Therefore, the use of TFN's to model the cash flows gives TFN's as a result. However, if the MARR is also modeled as a fuzzy number the result is not a TFN and the linear approximation to TFN multiplication and the vertex method must be used.

A. Example Problem

Suppose that an asset which is to perform service A is required by XYZ Corporation indefinitely. Asset B can be purchased for \$50 (all dollar amounts are in thousands) and has a physical life of five years. The aggregate cash flows (operating costs—operating revenues) for each year of the life of asset B are \$3, \$4, \$6, \$10, and \$12, respectively, for $n = 1, \cdots, 5$. If the asset is sold at the end of the year, its salvage value is \$35, \$30, \$27, \$23, and \$20, respectively, for $n = 1, \cdots, 5$. Assume a MARR of 10%. The problem is to determine the economic life of Asset B.

The assumption of certainty in future cash flows is unrealistic, except in some cases, such as when the asset is covered by a service contract. Likewise, future salvage values are dependent on the state of the equipment at that time, possible technological breakthroughs that have occurred, and numerous other uncertain events. The deterministic data are treated as the "most likely" estimates. The local expert or decision maker provides additional information, namely, the "best" and "worst" estimates.

Being pessimistic, the decision maker believes that the cash flows (operating costs less revenues) might be much higher than the "most likely" estimates and the salvage values might be much lower. Therefore, the high estimates for the cash flows are \$5, \$7, \$10, \$15, and \$18, respectively, for $n = 1, \cdots, 5$. The low estimates remain near the "most likely" estimates—\$2, \$3, \$5, \$8, and \$10. The salvage values high estimates are \$38, \$32, \$29, \$27, and \$25. The low estimates are \$32, \$24, \$21, \$18, and \$15.

Table I gives the calculated FAE costs for each replacement cycle $n = 1, \cdots, 5$, while Fig. 2 gives a graphical representation. Of these, the "minimum" must be chosen.

TABLE I
RESULTS OF PELAM FOR EXAMPLE PROBLEM

Replacement Cycle (Years)	Fuzzy AE Cost of Replacement Cycle
1	[19.00, 23.00, 28.00]
2	[16.05, 18.00, 23.33]
3	[14.58, 16.19, 20.94]
4	[14.22, 16.30, 20.75]
5	[14.30, 16.46, 21.09]

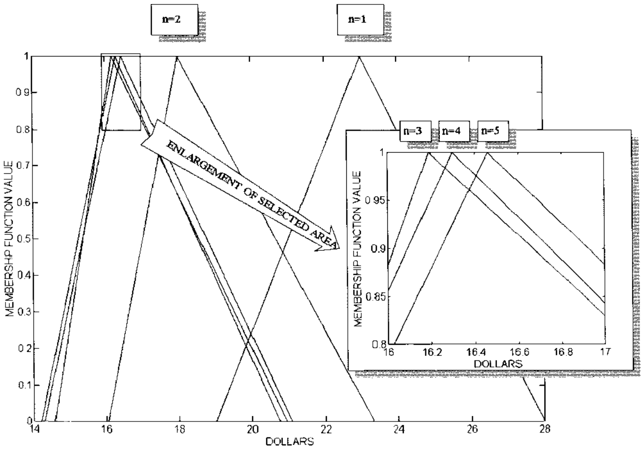


Fig. 2. TFN's representing the annual equivalent costs for the five possible replacement cycles for the example problem.

TABLE II
RANKING OF FAE COSTS IN EXAMPLE PROBLEM BY VARIOUS METHODS

Ranking Method with Ranking
Adamo, $\alpha = 0.9$
$FAE_3 \sim FAE_4 \sim FAE_5 < FAE_2 < FAE_1$
Chang
$FAE_3 < FAE_4 < FAE_5 < FAE_2 < FAE_1$
Chiu and Park, $w = 0.3$
$FAE_4 < FAE_3 < FAE_5 < FAE_2 < FAE_1$
Choobineh and Behrens
$FAE_4 < FAE_3 < FAE_5 < FAE_2 < FAE_1$
Kaufmann and Gupta
$FAE_4 < FAE_3 < FAE_5 < FAE_2 < FAE_1$
Traditional model
$AE_3 < AE_4 < AE_5 < AE_2 < AE_1$

To compare the alternatives described by fuzzy numbers, we need a ranking method. All ranking methods reported in the literature suffer from some pathological examples in which the result is counterintuitive [7], [10]. The rankings of selected methods are exhibited in Table II. Note that the rankings are not all in agreement, as shown in Table II. However, several of them do agree with each other (Chiu and Park, Choobineh and Behrens, and Kaufmann and Gupta). All three of these methods use a ranking function based on the $l, m,$ and r values of the TFN. In the remainder of this paper, the Kaufmann and Gupta method is used as the preferred ranking function for the reasons that follow.

The Kaufmann and Gupta method is a hierarchical test. It can basically be described as follows. Let $\tilde{M}_1 = [l_1, m_1, r_1]$ and $\tilde{M}_2 = [l_2, m_2, r_2]$ be two different TFN's.

TEST 1: Compare the *ordinal numbers*.

- IF $(l_1+2m_1+r_1) < (l_2+2m_2+r_2)$, THEN $\tilde{M}_1 < \tilde{M}_2$.
- ELSE IF $(l_1+2m_1+r_1) > (l_2+2m_2+r_2)$, THEN $\tilde{M}_1 > \tilde{M}_2$.
- ELSE go to TEST 2.

TEST 2: Compare the *modal values*.

- IF $m_1 < m_2$, THEN $\tilde{M}_1 < \tilde{M}_2$.
- ELSE IF $m_1 > m_2$, THEN $\tilde{M}_1 >; \tilde{M}_2$.
- ELSE go to TEST 3.

TEST 3: Compare the *divergence*.

- IF $(r_1 - l_1) < (r_2 - l_2)$, THEN $\tilde{M}_1 < \tilde{M}_2$.
- ELSE $\tilde{M}_1 > \tilde{M}_2$.

This method is usually used because 1) it is relatively easy to compute and 2) it always chooses a maximum when the two TFN's are not equal. The latter property is especially important in models based on DP in which a unique optimal value at each stage is desired.

B. Optimal Replacement Cycle for the Example Problem

We now show the determination of the optimal replacement policy for the example problem. Using the Kaufmann and Gupta ranking method for reasons discussed above, the optimal replacement cycle for Asset B is four years with a fuzzy AE cost of [14.22, 16.30, 20.75]. The modal values of the TFN's for each replacement cycle correspond to the *deterministic* AE costs from the traditional economic life of an asset model. Therefore, the traditional optimal replacement cycle is immediately available from the fuzzy solution and the fuzzy solution using fuzzy numbers is equivalent to performing sensitivity analysis on all uncertain variables. It is also interesting to note that using the traditional model the optimal replacement cycle is three years, which is different from the four years determined by PELAM. Additionally, PELAM determines the optimal replacement cycle based on the decision maker's estimates of the uncertainty and therefore provides a more informative answer than the traditional model.

V. General Single Asset Replacement

When a service is required for only a finite period or the aggregate cash flows are nonstationary with respect to installation time, a more general approach than PELAM is needed. The general single asset replacement problem is widely studied in the literature [2], [13], [29], [31]. It may be defined as follows:

a_n	asset in use at time period n;
N	number of time periods that service is required;
A_n	number of challenging assets at time period n.

The time periods $n = 0, 1, \cdots, N$ represent the periodic replacement decisions. If N is finite, the problem is a finite-horizon replacement problem, and if N is infinite, the problem is an infinite horizon problem. The existing asset is known as the *defender* and can only be placed into service at period zero. The A_0 assets available for replacement at time zero are known as *current challengers*, and the A_n assets available at future periods are known as *future challengers*. For each period n in the lifetime of each asset, there are three component cash flows describing the *installation cost and/or salvage value*, *operating costs*, and *operating revenues* at period n. The component cash flows of the future challengers are related to the corresponding component cash flow of a current challenger by a scalar function $f(a, n, C)$, where n is the time period in which asset a is installed and $C \in 1, 2, 3$ represents the respective component cash flow. This function allows for the modeling of inflation, technological improvements, and other time-dependent effects on cash flows. For example, to model a crisp 3% inflation rate, then $f(a, n, C) \equiv 1.03^n$ for all a, C. Other more complicated variations can be defined to model a wide range of factors. The component cash flows and relation functions are either known with certainty or estimated by the decision maker and may also be a fuzzy variable. The NPV of any sequence of cash flows $\{F_0, F_1, \cdots, F_N\}$ received at time periods $0, 1, \cdots, n$ with respect to MARR i is

$$\sum_{n=0}^{N} \frac{F_j}{(1+i)^n}. \tag{5}$$

The problem is to find the sequence(s) of keep/replace decisions that maximizes NPV$_i$, the net present value given some MARR i.

The Possibilistic Model for Single Asset Replacement via DP (PMSAR) [20] is a generalization of SREDM in [31] to allow for fuzzy parameters, such as aggregate cash flows, inflation, or technological change. SREDM used the "best," "worst," and "most likely" estimates of the parameters, as in PERT analysis, to create triangular probability distributions. Using those probability distributions, Monte Carlo simulation provided estimates of the probability of each asset being the optimal choice at period zero. Under such conditions of estimating the distributions, it is arguable that a possibility theory approach is more appropriate. Like PELAM, PMSAR uses TFN's for cash flows. However, there is also the possibility of having uncertainty in technological improvements, inflation, or other aspects of the future challengers, and this is modeled as a TFN through a relation function f. The solution technique is a forward dynamic program that uses the Kaufmann and Gupta ranking method to determine the optimal decision and functional equation value at each time period. The problem is to find the sequence(s) of keep/replace decisions that maximizes FPV_i, the fuzzy net present value given some MARR i.

We now present a fuzzy analog of the SREDM model. For ease of exposition, we define the following variables of the model.

1) Let k be the state of the system $k = 1, 2, \cdots, N$, which represents the number of periods of required service.
2) There are two decision variables: n, the period to place an asset into service, and a, the asset to place into service.
3) The immediate reward $r_i(a, n, k)$ is the FPV_i generated by placing asset a into service at period n and keeping it in service until period k.
4) The transition function $\tau(a, n, k)$ for placing asset a into service at time n for the remaining $k - n$ time periods is $\tau(a, n, k) = n$.

We define the function $\text{FPV}_i(a, n, C, k)$ as the fuzzy net present value with respect to MARR i of the *installation cost and/or salvage value*, *operating revenues*, and *operating costs* for $C = 1, 2, 3$, respectively, of placing asset a into service at period n for the remaining $k - n$ time periods. Collectively, the aggregate fuzzy net present value for all three component cash flows is

$$\text{FPV}_i(a, n, k) = \sum_{C=1}^{3} \text{FPV}_i(a, n, C, k). \tag{6}$$

Furthermore, define the functional $\text{FPV}_i^*(k)$ for this process as the value obtained using an optimal replacement policy from state zero to state k. Invoking Bellman's Principle of Optimality results in the following functional equation of a forward dynamic program:

$$\text{FPV}_i^*(k) = \max_{n,a} \{r_i(a, n, k) + \text{FPV}_i^*(n)\} \tag{7}$$

where

$$n \in \{0, 1, \cdots, k - 1\} \tag{8}$$

and

$$a \in \begin{cases} \{0, 1, \cdots, A_0\}, & \text{if } n = 0 \\ \{1, 2, \cdots, A_n\}, & \text{if } n > 0. \end{cases} \tag{9}$$

A boundary condition of $\text{FPV}_i^*(0) = [0, 0, 0]$ is assigned. The $\max$ operation is performed through the Kaufmann and Gupta ranking method on TFN's. Defining L_a as the physical lifetime, in time periods, of asset a gives

$$r_i(a, n, k) = \begin{cases} \text{FPV}_i(a, n, k), & \text{if } k - n \leq L_a \\ -M, & \text{otherwise} \end{cases} \tag{10}$$

where $M \gg 0$ is some sufficiently large number.

Relating the parameters of future challengers to the parameters of current challengers via a fuzzy relation function $f(a, n, C)$ is a desirable feature since there may exist considerable uncertainty of the nonprobabilistic nature in future events. This addition is not without its price, however. The model requires multiplication of two fuzzy numbers, which is only weakly distributive and is not closed over TFN's. This problem may be readily circumvented by the adroit use of the vertex method [12] and a TFN approximation to the product of two TFN's [22].

A. Example Problem

Let us now consider an adaptation of a replacement problem discussed by Lohmann [31] in which we specifically incorporate fuzzy uncertainty and use it as a vehicle for illustrating our point.

Three current challengers $A_n = 1, 2, 3$ for $n = 1, \cdots, 15$ can replace the defender $a = 0$. The time horizon of $N = 15$ years is established. Each challenger has a physical lifetime of five years. The defender has a remaining life of three years. For capital transfers, the cash flow at period zero is the purchase cost and the cash flows at periods $n > 0$ are salvage values. The component cash flows for the most likely estimates plus or minus a percentage are listed in Table III. Assume a MARR of 10%. Suppose the tax rate d is 50% and MACRS depreciation tables for a seven-year recovery period [14.29%, 24.49%, 17.49%, 12.49%, 8.92%, 8.92%, 8.92%, 4.46%] are utilized in the determination of depreciation tax shield.

TABLE III
DATA FOR EXAMPLE PROBLEM

a	n	$C = 1$	$C = 2$	$C = 3$
0	0	5423	0	0
	1	4194 ±5%	-15791 ±5%	19200 ±5%
	2	3355 ±5%	-17685 ±5%	19200 ±5%
	3	2684 ±10%	-19808 ±10%	19200 ±10%
1	0	20000	0	0
	1	16100 ±5%	-8000 ±5%	19200 ±5%
	2	13200 ±5%	-8960 ±5%	19200 ±5%
	3	10840 ±10%	-10035 ±10%	19200 ±10%
	4	8192 ±10%	-11239 ±10%	19200 ±10%
	5	6554 ±10%	-12588 ±10%	19200 ±10%
2	0	21000	0	0
	1	16800 ±5%	-7500 ±5%	19200 ±5%
	2	13440 ±5%	-8400 ±5%	19200 ±5%
	3	10252 ±10%	-9408 ±10%	19200 ±10%
	4	8602 ±10%	-10537 ±10%	19200 ±10%
	5	6881 ±10%	-11801 ±10%	19200 ±10%
3	0	22000	0	0
	1	17600 ±25%	-7250 ±25%	19200 ±5%
	2	14080 ±25%	-8120 ±25%	19200 ±5%
	3	11264 ±25%	-9094 ±25%	19200 ±10%
	4	9011 ±25%	-10186 ±25%	19200 ±10%
	5	7209 ±25%	-11408 ±25%	19200 ±10%

The optimal sequence of decisions is determined via forward DP with rewards modeled as fuzzy numbers. Four primary cash flows create the *installation cost and/or salvage value* component cash flow (see Fig. 3), as follows.

- Purchase cost of the asset: $\tilde{P}$.
- Tax on capital gains at the sale of the asset at time $k - n$

  $$d \cdot (\tilde{S}_{k-n} - \tilde{B}_{k-n})$$

  where the book-value $\tilde{B}$ is

  $$\tilde{B} = \tilde{P}\left(1 - \sum_{j=1}^{k-n} \text{MACRS}_j\right).$$

- Tax shield from depreciation of the asset during each time period

  $$d \cdot \tilde{P} \cdot \text{MACRS}_{k-n-j}$$

  for $j = 0, \cdots, k - n - 1$.
- Salvage value $k - n$:$\tilde{S}_{k-n}$ of the asset at time.

The *operating costs* component cash flow consists of a single cash outflow representing after-tax operating costs: $\tilde{C}_j$ for $j = 1, \cdots, k-n$. Similarly, the *operating revenues* component cash flow consists of a single cash inflow representing after-tax operating revenues: $\tilde{R}_j$ for $j = 1, \cdots, k - n$. Therefore, $\text{FPV}_i(a, n, 1, k)$ is as in (11), shown at the bottom of the next page. For $C = 2$, $\text{FPV}_i(a, n, 2, k)$ is

$$-\frac{(1-d)\displaystyle\sum_{j=1}^{k-n} \frac{\tilde{C}_j}{(1+i)^j}}{(1+i)^n} \tag{12}$$

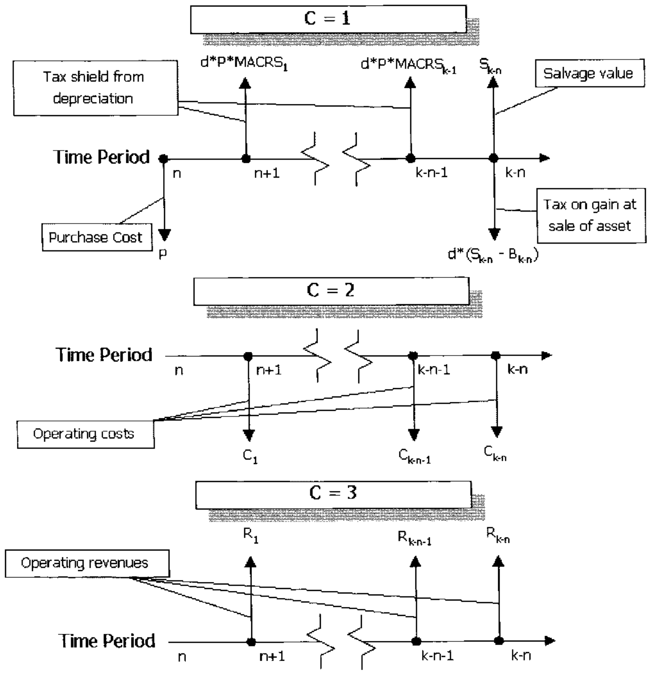


Fig. 3. Standard cash flow diagrams for each component cash flow in PMSAR.

and for $C = 3$, $\text{FPV}_i(a, n, 3, k)$ is

$$\frac{(1-d)\displaystyle\sum_{j=1}^{k-n}\frac{\tilde{R}_j}{(1+i)^j}}{(1+i)^n}. \tag{13}$$

Using the data from Table III, the calculations for $a = n = 0, k = 1$ are as follows:

$$\begin{aligned}\text{FPV}_{0.1}(0,0,1,1) &= [-1146.95, -1051.64, -956.32]\\ \text{FPV}_{0.1}(0,0,2,1) &= [-7536.61, -7177.73, -6818.84]\\ \text{FPV}_{0.1}(0,0,3,1) &= [8290.91, 8727.27, 9163.64]\end{aligned}$$

which yields

$$\text{FPV}_{0.1}(0,0,1) = [-392.66, 497.91, 1388.48].$$

Similar calculations for $a = 1, 2, 3$ yield

$$\begin{aligned}\text{FPV}_{0.1}(1,0,1) &= [515.91, 1500.00, 2484.09]\\ \text{FPV}_{0.1}(2,0,1) &= [511.36, 1500, 2488.64]\\ \text{FPV}_{0.1}(3,0,1) &= [-228.41, 1431.82, 3092.04].\end{aligned}$$

Note that the salvage and purchase values and functional relation (if used) appear more than once in the FPV_i calculations for $C = 1$; therefore, the vertex method is used to generate the correct FPV_i. This entails computing deterministically the problem for the modal (most likely values) as well as an additional 2^3 times for all possible combinations of "best" and "worst" estimates for these parameters. Thus, (7) gives the following for a one-year horizon $k = 1$:

TABLE IV
DYNAMIC PROGRAMMING RESULTS FOR BASIC EXAMPLE PROBLEM

N	Optimal Policy (asset, time installed)	$FPV^{\star}_{0.1}(N)$
1	(1,0)	[515.91,1500.00,2484.09]
2	(1,0)	[1686.70,3159.42,4632.15]
3	(1,0)	[2054.50,4759.95,7465.39]
4	(2,0)	[2799.62,6360.66,9921.71]
5	(3,0)	[413.48,7715.23,15017.00]
6	(3,0) (2,5)	[731.00,8646.61,16562.20]
7	(3,0) (1,5)	[1460.79,9676.98,17893.20]
8	(2,0) (2,4)	[4711.79,10705.10,16698.40]
9	(3,0) (2,5)	[2151.82,11664.70,21177.60]
10	(3,0) (3,5)	[670.22,12505.80,24341.40]
11	(3,0) (3,5) (2,10)	[867.37,13084.10,25300.80]
12	(3,0) (3,5) (1,10)	[1320.51,13723.90,26127.20]
13	(3,0) (2,5) (2,9)	[3339.13,14362.20,25385.40]
14	(3,0) (3,5) (2,10)	[1749.59,14958.10,28166.60]
15	(3,0) (3,5) (3,10)	[829.63,15480.30,30131.10]

$$\begin{aligned}\text{FPV}^*_{0.1}(1) &= \max_{n,a}\{\text{FPV}_{0.1}(a,n,1) + \text{FPV}^*_{0.1}(0)\}\\ &= [515.91, 1500.00, 2484.09] \quad \text{for } a = 1. \end{aligned} \tag{14}$$

The same calculations are performed for the remaining stages, up to $k = N$, recording both the functional equation value and the optimal decision for each stage as in Table IV.

B. Optimal Replacement Policy

As in the previous example, we show the determination of the optimal replacement problem for the model example under fuzziness. Solving the functional equation given in (7) for $N = 15$ results in the following optimal replacement policy II:

$$(3,0)(3,5)(3,10)$$

with

$$\text{FPV}^*_{0.1}(15) = [829.63, 15480.30, 30131.10].$$

This can be interpreted as buying asset three at time zero and again at times five and ten. In this solution, the modal values of each $\text{FPV}^*_i(k)$ represent the $\text{NPV}^*_i(k)$ of the deterministic model, while the lower and upper values indicate the overall uncertainty in the decision. The uncertainty signified by the width of the base of each $\text{FPV}^*_i(k)$ also is equivalent to the range of possible values determined via multivariable sensitivity analysis. Thus, a fuzzy model immediately provides both the traditional deterministic NPV for this policy as well as the range of values the NPV may take due to uncertainty in the parameters.

$$\frac{-\tilde{P} + \dfrac{\tilde{S}_{k-n}}{(1+i)^{k-n}} + d\tilde{P}\displaystyle\sum_{j=1}^{k-n}\frac{\text{MACRS}_j}{(1+i)^j} - \dfrac{d(\tilde{S}_{k-n} - \tilde{B}_{k-n})}{(1+i)^{k-n}}}{(1+i)^n} \tag{11}$$

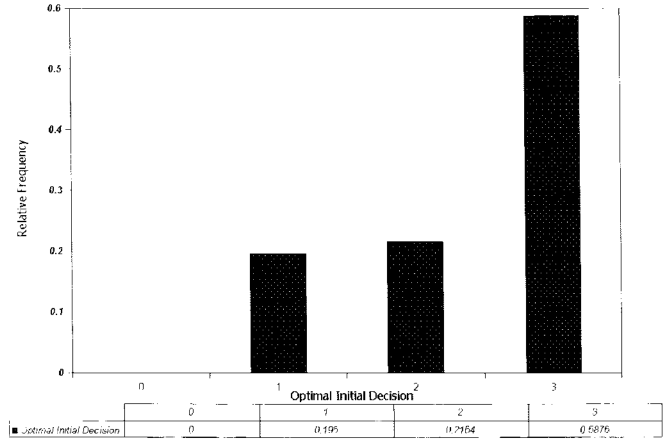


Fig. 4. Monte Carlo simulation results for the example problem depicting the probability distribution for the optimal first choice.

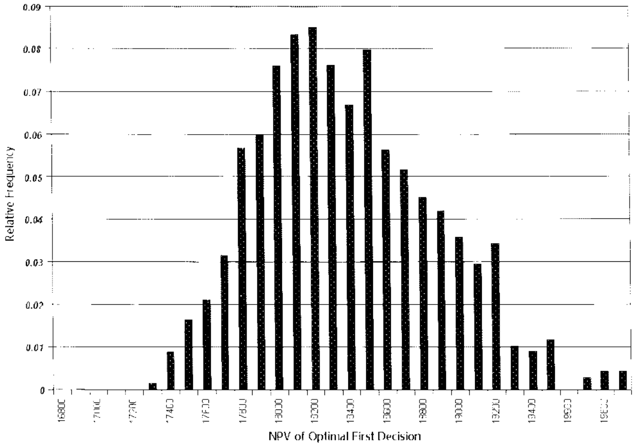


Fig. 5. Monte Carlo simulation results for the example problem depicting the probability distribution for the optimal NPV.

Other uncertain factors may be introduced via the relation function $f(a, n, C)$. If, for example, a moderate inflation increase of [1%, 2%, 3%] per year were expected while the other parameters remained the same, the optimal replacement policy Π becomes

$$(3,0)(2,5)(2,10)$$

with

$$\text{FPV}^*_{0.1}(15) = [-969.74, 16558.10, 34313.60].$$

Contrast this with the traditional stochastic model SREDM where the solution is derived using Monte Carlo simulation; a technique that generates a large number of realizations of the uncertain (and assumed random) variables and solves each set of them deterministically [31]. From this large sample, the probability that each alternative current asset is the optimal first choice can be estimated (see Fig. 4) as well as the corresponding cumulative probability distributions of the following:

1) economic life of each current asset;
2) NPV of the optimal sequence of challengers for a finite horizon (see Fig. 5);
3) equivalent finite horizon time for infinite horizon problems.

We note that the probability distributions generated by SREDM are *subjectively* interpreted to determine the optimal *current* decision and no information regarding future decisions is available.

VI. Maintenance Decisions

Not all replacement decisions are of the form in which the only decisions are to "keep" or "replace" the existing asset. However, the DP formulation is sufficiently general to allow for other options. For example, a third option may be the "purchase of a used machine." Bellman describes the DP formulation of the replacement model with the option to purchase a used machine as well as that of a model with an "overhaul" option [5]. The overhaul option can be done either in a general manner in which the cash flows and other variables are functions of both the installation time and the overhaul time or by allowing the overhauled asset (t years old) to maintain the characteristics of a younger asset ($t' < t$ years old).

The effects of the maintenance may also be uncertain variables that can be appropriately modeled as fuzzy numbers. Modeling an overhaul or maintenance option in PMSAR can be done either by defining functions that relate an overhauled asset's aggregate cash flows and salvage value to its installation time and overhaul time or by defining challenging assets that represent the costs and characteristics of an overhauled machine.

We return to the example problem for the PMSAR above to illustrate a maintenance option with an imprecise or vague effect. Suppose that the maintenance option is modeled as a "ghost" asset, which is defined as follows. The purchase cost P' of the "ghost" asset a' is a function of the cost of the maintenance option on the asset a currently in place and the number of years n' since a was installed

$$P' = ([0.15, 0.20, 0.25] \cdot P_a)(1.03)^{n'}$$

where P_a is the purchase price of asset a. This particular function states that the base maintenance cost is a fuzzy number that is [15%, 20%, 25%] of the original cost of the asset with an increase of 3% per year since installation. The result of the maintenance is also fuzzy, a [6%, 10%, 12%] reduction in the original operating costs for the following year. The component cash flows corresponding to this maintenance challenger a' are a function of the time n' that the maintenance occurs as well as the original asset and its installation time. Let us denote performing maintenance on the existing asset at period n as (M, n). The resulting optimal maintenance and replacement policy Π for $N = 15$ then becomes

$$\begin{aligned}&(M,0)(M,1)\\&(1,2)(M,3)(M,4)(M,5)(M,6)\\&(1,7)(M,8)(M,9)(M,10)(M,11)\\&(2,12)(M,13)(M,14)\end{aligned}$$

with an optimal fuzzy present value $\text{FPV}^*_{0.1}(15) = [25\,709.00, 40\,749.90, 54\,887.80]$. This result can be translated to performing maintenance on the *current defender* at periods zero and one, purchasing asset one at period two and performing maintenance on this asset each period until asset one is purchased again in time seven. The maintenance at each period continues on asset one until asset two is purchased at period 12. Maintenance on asset two is then performed at each remaining period. As can be seen, the fuzzy present value with this maintenance option, in this example, has risen significantly.

VII. Summary

Probability theory has been used as the traditional approach for modeling uncertainty in economic analysis. This is acceptable only to the extent that uncertainty is satisfactorily equated with randomness. However, there exist other types of uncertainty that are especially relevant to economic decision analysis. Thus, there is a role to be played by nonprobabilistic uncertainty, as shown in this effort. Many approaches have been shown possible. A brief survey of replacement analysis, focusing on the use of nonprobabilistic uncertainty, is given. The use of TFN's provides a compromise between computational efficiency and realistic modeling of the uncertainty. Thus, this discussion emphasizes fuzzy numbers.

In the extension to the *economic life of an asset* model, the uncertainty in the parameters is explicitly modeled. By only a threefold increase in the number of computations, the optimal choice based on the decision maker's best estimates of these parameters is easily obtained. The traditional deterministic models are a special case of this new possibilistic model. In effect, PELAM performs multivariable sensitivity analysis on all uncertain parameters *concurrently* and incorporates this uncertainty into the determination of the optimal decision. The benefits for PMSAR are virtually the same, except that there is a greater increase in the number of computations due to the vertex method. Contrast this with the large number of repetitions that Monte Carlo simulation requires, as well as the subjective interpretation of the results, and this disadvantage is not severe.

When probability distributions are not known nor can be correctly assumed, or when a stochastic model is too difficult to solve, fuzzy sets and possibility theory offer an efficient alternative in replacement analysis. There are a number of benefits for modeling the uncertainty in the replacement problem via fuzzy numbers. We recount a few of them, as follows.

1) Use of fuzzy uncertainty may be more appropriate when modeling systems with human subjectivity. The only existing technique in replacement analysis that modeled general uncertainty in the replacement decision was a Monte Carlo simulation method [29].
2) Creating a triangular distribution from the best, worst, and most likely estimates of an expert is more appropriate for possibility theory than probability theory due to the lack of probabilistic information.
3) Results of each model can be easily plotted to provide a graphical representation of the effects of the uncertain parameters and can provide this information without a significant increase in the computational complexity. The algorithms are easily implemented on a personal computer.
4) Use of TFN's retains the triangular distribution throughout the solution algorithm and conveniently shows the best, worst, and most likely NPV's of each possible decision.
5) Use of fuzzy numbers corresponds to performing sensitivity analysis on all uncertain parameters simultaneously as well as to identifying the answer to the deterministic version of the problem that corresponds to the modal value of each fuzzy number.

We have illustrated the benefits of explicitly modeling uncertainty in engineering economy decisions using a fuzzy set-theoretic framework. As in traditional replacement analysis, the age of the asset served as the surrogate variable from which all cash flows could be determined. With all factors influencing the decision in the *models* being measured in terms of profit, the DP approach takes advantage of recurring subproblems to efficiently determine the optimal policy. Other factors, such as operating efficiency, machine safety, and machine reliability, may affect the replacement and maintenance decisions. Yet, these attributes are not easily quantified, especially in terms of profit. One approach in practice is to convert maintenance down-time into lost sales, examine safety issues in terms of insurance costs and worker's compensation, and use other obtainable measures as surrogates for these attributes. However, it is arguable that the decision making problem may not be adequately modeled in that framework. A fuzzy set-theoretic approach that identifies different measures for levels of repair, safety, maintenance up- and down-time, and other imprecisely defined attributes may provide a more informative replacement policy. Modeling in this framework may require a multiple attribute decision making approach that incorporates the unquantifiable, incomplete, or unobtainable information. The reader who wishes to explore this possibility is referred to [26] for fuzzy approaches to multiple attribute and multiple objective decision making problems.

Appendix

Fuzzy set theory, a generalization of classical set theory, was developed in [43]. Fuzziness describes sets that have no sharp transition from membership to nonmembership. Traditional modeling methods assume certain and unambiguous structures and parameters, but uncertainty is inherent in most real-world systems. Fuzzy set theory provides a strict mathematical theory to describe this inherent characteristic.

Let $\boldsymbol{X}$ be a collection of objects denoted generically by $\boldsymbol{x}$.

Definition 1: (A fuzzy set (or fuzzy subset) $\tilde{A}$ in $\boldsymbol{X}$ is a set of ordered pairs

$$\tilde{A} = \{\boldsymbol{x}, \mu_{\tilde{A}}(\boldsymbol{x}) | \boldsymbol{x} \in \boldsymbol{X}\}$$

where $\mu_{\tilde{A}}(\boldsymbol{x})$ is the membership function of $\boldsymbol{x}$ in $\tilde{A}$, which maps $\boldsymbol{X}$ to the membership space $\boldsymbol{M}$. If $\boldsymbol{M}$ is the closed interval [0, 1], then $\tilde{A}$ is called a normal fuzzy set [43].

Definition 2: (The α-level set of a fuzzy set $\tilde{A}$ is the crisp set of elements that belong to $\tilde{A}$ at least to the degree $\alpha \in [0, 1]$ [43].

Definition 3: A fuzzy set $\tilde{A}$ is convex if

$$\mu_{\tilde{A}}(\lambda \boldsymbol{x}_1 + (1-\lambda)\boldsymbol{x}_2) \geq \min(\mu_{\tilde{A}}(\boldsymbol{x}_1), \mu_{\tilde{A}}(\boldsymbol{x}_2))$$

for $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \boldsymbol{X}$ and $\lambda \in [0, 1]$. An alternative definition is a fuzzy set is convex if all α-level sets are convex [43].

The basis for generalizing crisp (nonfuzzy) mathematical concepts to fuzzy sets is the extension principle.

Definition 4: Let $\boldsymbol{X}$ be a Cartesian product of universes $\boldsymbol{X} = \boldsymbol{X}_1, \cdots, \boldsymbol{X}_r$, and $\tilde{A}_1, \cdots, \tilde{A}_r$ be r fuzzy sets in $\boldsymbol{X}_1, \cdots, \boldsymbol{X}_r$, respectively. Let f be a mapping from $\boldsymbol{X}$ to a universe $\boldsymbol{Y}, y = f(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_r)$. The extension principle allows the definition of a fuzzy set $\tilde{B}$ in $\boldsymbol{Y}$ by

$$\tilde{B} = \{(y, \mu_{\tilde{B}}(y)) | y = f(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_r), (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_r) \in \boldsymbol{X}\}$$

where $\mu_{\tilde{B}}(y)$ is

$$\sup_{(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_r) \in f^{-1}(y)} \min\{\mu_{\tilde{A}_1}(x_1), \cdots, \mu_{\tilde{A}_r}(x_r)\}$$

if $f^{-1}(y) \neq \emptyset$ and zero otherwise [44].

Fuzzy numbers are a particular kind of fuzzy sets.

Definition 5: A fuzzy number $\tilde{M}$ is a convex normal fuzzy set on the real line R such that 1) there exists exactly one $\boldsymbol{x}_0 \in R$ with $\mu_{\tilde{M}}(\boldsymbol{x}) = 1$ and 2) $\mu_{\tilde{M}}(\boldsymbol{x})$ is piecewise continuous. Denote the $\boldsymbol{x}_0 \in R$ that satisfies 1) as the modal value of the fuzzy number [46].

Fuzzy arithmetic is based on the extension principle. The arithmetic operations of addition, subtraction, multiplication, and division were developed by Dubois and Prade [14] and an excellent overview is given in [15].

Definition 6: A fuzzy number $\tilde{M}$ is positive (negative) if $\mu_{\tilde{M}}(\boldsymbol{x}) = 0$ for all $x < 0$ $(x > 0)$ [14].

Though the addition, subtraction, multiplication, and division operations are defined for general fuzzy numbers, we focus on the operations as they apply to TFN's. Let $\tilde{M}_1 = [l_1, m_1, u_1]$ and $\tilde{M}_2 = [l_2, m_2, u_2]$ be two TFN's with respective lower, most likely, and upper estimates. The sum, defined as

$$\tilde{M}_1 + \tilde{M}_2 = [l_1 + l_2, m_1 + m_2, u_1 + u_2]$$

is associative and commutative. The subtraction operation is simply the addition operation on two fuzzy numbers, one of which has been multiplied by the scalar -1. Scalar multiplication for TFN's is

$$a\tilde{M}_1 = \{\begin{cases} [al_1, am_1, au_1], & \text{for } a \geq 0 \\ [au_1, am_1, al_1], & \text{for } a < 0. \end{cases}$$

The difference of two TFN's is therefore

$$\tilde{M}_1 - \tilde{M}_2 = [l_1 - u_2, m_1 - m_2, u_1 - l_2]$$

which is also associative and commutative.

The multiplication operation for TFN's is only weakly distributive over addition. Therefore, the solution procedure must either use the vertex method [12] or it may give a different outcome. For example, the following property that holds true in all cases:

$$\tilde{M}_1(\tilde{M}_2 + \tilde{M}_3) \subseteq \tilde{M}_1\tilde{M}_2 + \tilde{M}_1\tilde{M}_3$$

is only equality under special circumstances or through the use of the vertex method. The vertex method for TFN's is straightforward. Each of the three TFN's above has two extreme points l_i and u_i for $i = 1, 2, 3$. All 2^3 combinations are calculated for the above expression, and the minimum and maximum values are chosen. The drawback is that there is an exponential number of calculations under this algorithm.

ACKNOWLEDGMENT

The authors are grateful to the referees for their valuable comments.

REFERENCES

[1] A. A. Alchian, "Economic replacement policy," The RAND Corp., Santa Monica, CA, Tech. Rep. Publ. R-224, 1952.

[2] J. C. Bean, J. R. Lohmann, and R. L Smith, "A dynamic infinite horizon replacement economy decision model," *Eng. Econom.*, vol. 30, no. 2, pp. 99–120, Winter 1985.

[3] A. M. Behrens and F. Choobineh, "Can economic uncertainty always be described by randomness?" in *Proc. 1989 Int. Ind. Eng. Conf.*, pages 116–120.

[4] R. E. Bellman, "Equipment replacement policy," *SIAM J. Appl. Math.*, vol. 3, pp. 133–136, 1955.

[5] R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*. Princeton, NJ: Princeton Univ. Press, 1962.

[6] J. Bezdek, "Fuzziness versus probability—again (!?)," *IEEE Trans. Fuzzy Syst.*, vol. 2, pp. 1–3, Jan. 1994.

[7] G. Bortolan and R. Degani, "A review of some methods for ranking fuzzy subsets," *Fuzzy Sets Syst.*, vol. 15, pp. 1–19, 1985.

[8] J. J. Buckley, "The fuzzy mathematics of finance," *Fuzzy Sets Syst.*, vol. 21, pp. 257–273, 1987.

[9] ——, "Solving fuzzy equations in economics and finance," *Fuzzy Sets Syst.*, vol. 48, pp. 289–296, 1992.

[10] C. Y. Chiu and C. S. Park, "Fuzzy cash flow analysis using present worth criterion," *Eng. Econ.*, vol. 39, no. 2, pp. 113–138, 1994.

[11] F. Choobineh and A. Behrens, "Use of intervals and possibility distributions in economic analysis," *J. Oper. Res. Soc.*, vol. 43, no. 9, pp. 907–918, 1992.

[12] W. Dong and H. C. Shah, "Vertex method for computing functions of fuzzy variables," *Fuzzy Sets Syst.*, vol. 14, pp. 65–78, 1987.

[13] S. E. Dreyfus and A. M. Law, *The Art and Theory of Dynamic Programming*. New York: Academic, 1977.

[14] D. Dubois and H. Prade, "Operations on fuzzy numbers," *Int. J. Syst. Sci.*, vol. 9, pp. 612–626, 1978.

[15] ——, "Fuzzy numbers: An overview," in *Analysis of Fuzzy Information*, vol. 1, J. C. Bezdek, Ed. Boca Raton, FL: CRC, 1987, pp. 3–39.

[16] ——, "Fuzzy sets–A convenient fiction for modeling vagueness and possibility," *IEEE Trans. Fuzzy Syst.*, vol. 2, pp. 16–21, Jan. 1994.

[17] A. O. Esogbue and J. Kacprzyk, "Fuzzy dynamic programming: A survey of main developments and applications," *Arch. Contr. Sci.*, vol. 5, nos. 1-2, pp. 39–59, 1996.

[18] B. Gluss, *An Elementary Introduction to Dynamic Programming: A State Equation Approach*. New York: Allyn and Bacon, 1972.

[19] C. P. Gupta, "A note on the transformation of possibilistic information into probabilistic information for investment decisions," *Fuzzy Sets Syst.*, vol. 56, pp. 175–182, 1993.

[20] W. E. Hearnes, II, "Modeling with possibilistic uncertainty in the single asset replacement problem," in *Proc. Int. Joint Conf. Inf. Sci., 4th Annu. Conf. Fuzzy Theory Technol.*, Wrightsville Beach, NC, Sept. 28–Oct. 1 1995, pp. 552–555.

[21] J. Kacprzyk, "Decision-making in a fuzzy environment with fuzzy termination time," *Fuzzy Sets Syst.*, vol. 1, pp. 169–179, 1978.

[22] A. Kaufmann and M. M. Gupta, *Fuzzy Mathematical Models in Engineering and Management Science*. Amsterdam, The Netherlands: Elsevier, 1988.

[23] G. J. Klir, "On the alleged superiority of probabilistic representation of uncertainty," *IEEE Trans. Fuzzy Syst.*, vol. 2, pp. 27–31, Jan. 1994.

[24] G. J. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Upper River Saddle, NJ: Prentice-Hall PTR, 1995.

[25] B. Kosko, "The probability monopoly," *IEEE Trans. Fuzzy Syst.*, vol. 2, p. 32–33, Jan. 1994.

[26] Y.-J. Lai and C.-L. Hwang, *Fuzzy Multiple Objective Decision Making*. Berlin, Germany: Springer-Verlag, 1994.

[27] M. Laviolette and J. W. Seaman, Jr., "The efficacy of fuzzy representations of uncertainty," *IEEE Trans. Fuzzy Syst.*, vol. 2, pp. 4–15, Jan. 1994.

[28] M. Li Calzi, "Toward a general setting for the fuzzy mathematics of finance," *Fuzzy Sets Syst.*, vol. 35, pp. 265–280, 1990.

[29] J. R. Lohmann, "A stochastic replacement economic decision model," *IIE Trans.*, vol. 18, pp. 182–194, 1986.

[30] R. V. Oakford, *Capital Budgeting: A Quantitative Evaluation of Investment Alternatives*. New York: Ronald, 1970.

[31] R. V. Oakford, J. R. Lohmann, and A. Salazar, "A dynamic replacement economy decision model," *IIE Trans.*, vol. 16, pp. 65–72, 1984.

[32] R. V. Oakford, A. Salazaar, and H. A. DiGiulio, "The long term effectiveness of expected net present value maximization in an environment of incomplete and uncertain information," *AIIE Trans.*, vol. 13, pp. 265–276, 1981.

[33] R. J. Samuelson, "Imperfect vision: A flawed CPI makes it harder for us to read the past and plan for the future," *Newsweek*, vol. 55, 1996.

[34] L. Savage, *The Foundations of Statistics*. New York: Wiley, 1954.

[35] G. P. Sharp-Bette and C. S. Park, *Advanced Engineering Economics*. New York: Wiley, 1990.

[36] G. Terborgh, *Dynamic Equipment Policy*. Washington, DC: Machinery Appl. Products Inst., 1949.

[37] A. Tversky and D. Kahneman, "Belief in the law of small numbers," *Psychol. Bull.*, vol. 2, pp. 105–110, 1971.

[38] ——, "Judgment under uncertainty: Heuristics and biases," *Science*, vol. 185, pp. 1124–1131, 1974.

[39] ——, "The framing of decisions and the psychology of choice," *Science*, vol. 211, pp. 3–8, 1981.

[40] J. Von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton, NJ: Princeton Univ. Press, 1944.

[41] H. M. Wagner, *Principles of Operations Research*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

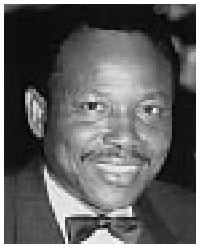
[42] T. L. Ward, "Discounted fuzzy cash flow analysis," in *Proc. 1985 Fall Ind. Eng. Conf.*, pp. 476–481.

[43] L. A. Zadeh, "Fuzzy sets," *Inf. Contr.*, vol. 8, pp. 338–353, 1965.

[44] ——, "The concept of a linguistic variable and its application to approximate reasoning i, ii, iii," *Inf. Sci.* , pp. 8-9:8:199–251; 8:301:357; 9:43–80, 1975.

[45] ——, "Fuzzy sets as a basis for a theory of possibility," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-1, pp. 3–28, Jan. 1978.

[46] H. J. Zimmerman, *Fuzzy Set Theory and Its Applications*, 2nd ed. Boston, MA: Kluwer, 1991.



Augustine O. Esogbue (S'65–M'66–SM'98) received the B.S. degree in electrical engineering and mathematics from the University of California, Los Angeles, in 1964, the M.S. degree in industrial engineering and operations research from Columbia University, New York, in 1965, and the Ph.D. degree in industrial and systems engineering (operations research) from the University of Southern California, Los Angeles, in 1968.

He was an Assistant Professor in the Department of Operations Research and Systems Research, Case Western Reserve University, Cleveland, OH, from 1968 to 1972. He then joined the faculty of the School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, as an Associate Professor. He was promoted to Professor in 1977 and is currently the Director of the Intelligent Systems and Controls Laboratory. His research interests include operations research, particularly, dynamic programming, computational intelligence, decision making and control in a fuzzy environment, and operations research with application to sociotechnical problems, such as healthcare, water resource management, and disaster control planning.

Dr. Esogbue is a Fellow of the American Association for the Advancement of Science, a Member of the New York Academy of Sciences, a Consultant for several agencies, and serves on several panels of the National Research Council. He is active on the program committees for international conferences and serves on the editorial boards of several journals, including the *International Journal of Advanced Computational Intelligence*, *Journal of Mathematical Analysis and Applications*, and *Fuzzy Sets and Systems*. He also actively promotes the National Society of Black Engineers and is a member of its National Advisory Board.

Warren E. Hearnes, II, received the B.S. degree in general engineering from the United States Military Academy, West Point, NY, in 1989. He is currently pursuing the Ph.D. degree from the Georgia Institute of Technology, Atlanta.

His research interests include engineering economy, decision analysis, dynamic programming, and intelligent control.

NEURO-FUZZY ADAPTIVE CONTROL: STRUCTURE, PERFORMANCE, AND APPLICATIONS

Augustine O. Esogbue
Intelligent Systems and Control Lab
School of Industrial and Systems Engineering
Georgia Institute of Technology

1 Introduction

Solution to control problems in which there is considerable uncertainty or lack of knowledge about the process is the focal concern of our research mission. The class of problems used as the leitmotif of our work may be subsumed under the rubric of what is generally known as the general set-point regulator problem. This group encompasses the usual dynamical system plants as well as discrete state space problems which arise in the control of operations and manufacturing. Our work has focused on the development and demonstration of the potential of the proposed Statistical Fuzzy Associative Learning Controller in simulated applications to three representatives of this class of problems, namely, process control of particular second-order dynamical systems, message routing in communication networks and the power system stabilization and control problem . In each, there are complexities and uncertainties which call for intelligence, but intelligence which can be automated for speed and repeatability.

Classical analytical and optimization approaches generally require some type of model of the plant and specific knowledge of what constitutes desirable plant behavior. For cases where the lack of knowledge is of a high order, a variety of intelligent methods have been suggested to provide more acceptable performance than can be achieved with the classical methods. In our review of existing approaches(Esogbue and Murrell 1994), it was shown that many of them consist of off-line empirical modeling using a training set of data. There are many instances in which these approaches work well for control of uncertain systems. However, a training set generally requires some knowledge of what constitutes good control, and in uncertain systems, this is often lacking. Also, many systems are based on an explicit gradient-type of approach, such as the well-known backpropagation neural networks with their attendant drawbacks. It is also noteworthy that some of these problems are framed in terms of discrete spaces, do not have objective functions which are analytically differentiable, or any known objective function formula at all. Hence, gradient methods which proliferate in the literature may not be applicable.

An alternative approach to control which has had notable success is that of fuzzy controllers, which mathematically mimic the imprecise reasoning processes which are an important part of what makes humans so successful at solving problems with large uncertainties.

However, the problem has been to find effective ways to capture or acquire the knowledge or skill required for a particular problem. Recent research has been devoted to ways of integrating learning algorithms with fuzzy control to automate the knowledge acquisition. Most of the attempts to combine the power of fuzzy reasoning with learning capability have taken either of two approaches. One approach has been to implement fuzzy reasoning with AI-type rule-based methods. The resulting controller models can become quite complicated and cumbersome as they are scaled up to larger systems. The second approach is to use gradient-type neural networks combined with fuzzy logic. Although these have been used successfully in some applications, they share the limitations of gradient methods.

It is now believed that learning systems (in the sense of mathematical learning theory) may be the most appropriate for the problems with a high order of uncertainty. A reinforcement learning trial system can learn on-line from its own experience with the process it is designed to control. Although such systems have been suggested since the 1960's, their applicability has been significantly inhibited by the almost exponential growth in complexity as it is scaled up to larger systems due to the necessity of discretizing the spaces. Combining these methods with fuzzy discretization and fuzzy logic can make learning trial methods a viable, practical approach. However, until the research first reported in Esogbue and Murrell(1993), and then furthered by Murrell(1994) and Esogbue, Hearnes and Song (1995), very little development of this idea has been seen in the literature.

These problems are second-order linear and nonlinear dynamical systems of the type which often occur in industrial process control. In these problems, uncertainty about the true plant model is often introduced by the environment, or by frequent changes in configuration necessitated for example by flexible manufacturing practices. Additionally, unknown, time-varying and nonlinear dynamics complicate the use of the plant model required by classical control theory methods. These concerns, necessitate the pursuit of novel approaches of the sort dissussed in the sequel.

2 The Self-Learning Fuzzy-Neuro Controller

In response to the quest for an intelligent controller that can learn online, does not require an exact model of the plant, operates without the benefit of what is considered optimal, does not need a set of predefined control rules, and uses feedback in an instructive way without the attendant expensive computational costs, Esogbue and Murrell [12] reported the development of a self-learning fuzzy-neuro controller with unique features and capabilities. This controller consists of five parts: (1) the Statistical Fuzzy Discretization Network (SFDN) which employs a variation of the Kohonen self-organizing map (SOM) to fuzzify the state space of the plant; (2) the Fuzzy Correlation Network (FCN) which implements the learned fuzzy control rules as fuzzy relation; (3) the Stochastic Learning Correlation Network (SLCN) which maps a particular fuzzy state to a set of fuzzy control actions through an adaptive stochastic algorithm; (4) the Control Activation Network (CAN) which defuzzifies the fuzzy control to a crisp control signal; and (5) the Performance Evaluation System (PES) which provides feedback reinforcement signals to the learning algorithm based on the effectiveness of the control action. A block diagram of the controller is shown in Figure 1.

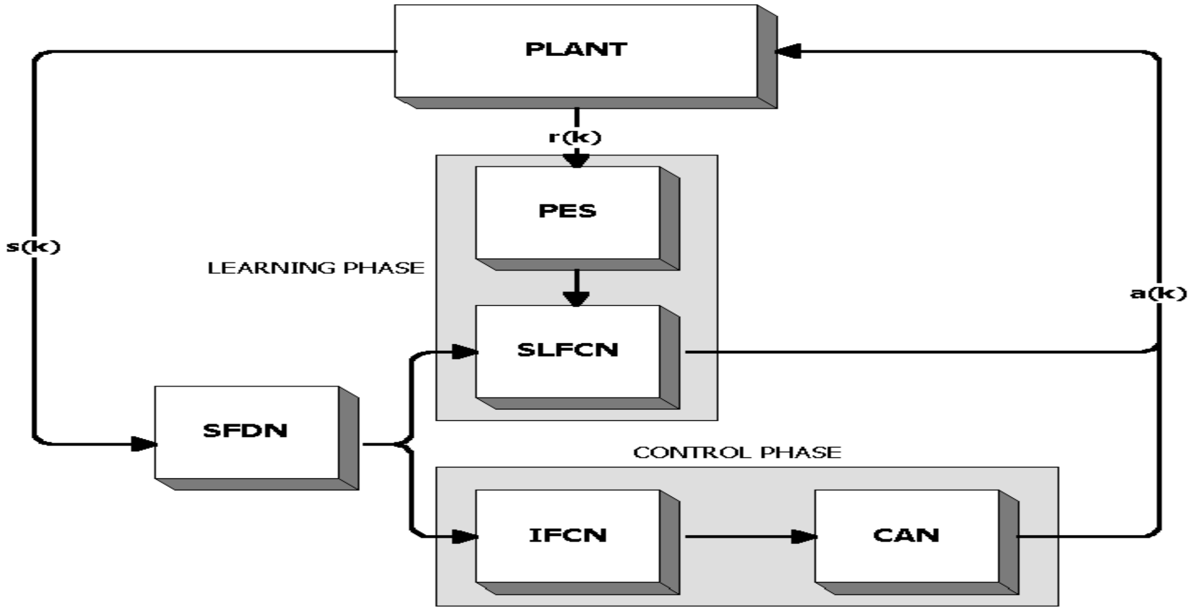


Figure 1: Block Diagram of Fuzzy-Neuro Controller

2.1 Statistical Fuzzy Discretization Network (SFDN)

This subsystem consists of a network of automata nodes (“neurons”) arranged in a grid (see Each node receives as its input the current process state vector. Every time a vector is input, each node computes an output that represents the degree of membership in the fuzzy subset of the input space that corresponds to that node. This output, called the node activation, is a measure of the degree of similarity of the input state vector to the ideal or prototype member of that fuzzy set. It is computed as some combination of the state vector and a location parameter vector associated with that node which represents the prototype process state for the corresponding fuzzy set. Also associated with each node is a parameter that encodes the degree of dispersion or spread of its fuzzy set membership function used to calculate the node activation. For a particular membership function form, the location parameters and spread parameter together define a fuzzy set. The SFDN thus performs a fuzzy discretization, inducing a fuzzy partition of the state space X into reference fuzzy subsets $X_1, X_2, \dots, X_r$, each represented by a node in the grid.

The network described here is an extension of Kohonen's self-organizing feature map to fuzzy characterization of dynamic plant states. The output of the i^{th} node in the map is

$$a_i = \exp(-\| \ x - m_i \| \ / s_i) \tag{1}$$

where x is the state vector input, m_i is the vector of location parameters and s_i the spread parameter for the i^{th} node, and it is assumed that the choice of similarity measure is the Euclidean metric and the functional form of the membership functions is a gaussian function. Thus, the vector a of node activations is the fuzzy hyperstate due to input state vector x:

$$a = (\pi_{X_1}(x), \dots, \pi_{X_r}(x))^T \tag{2}$$

where $\pi_{X_i}(\bullet)$ is the membership function for fuzzy set X_i given by the foregoing equation.

A sequence of state vectors is input to the network over time, and an adjustment algorithm adapts the location parameters to reflect the actual clustering of the state vectors by which the aggregation into fuzzy subsets is determined. A simplified version of the update rule of the j^th component of m_i for this example is

$$m_{ij,k+1} = \begin{cases} m_{ij,k} + \alpha_k(x_{j,k} - m_{ij,k}) & \text{for } i \in T_{c_k} \\ m_{ij,k} & \text{otherwise} \end{cases} \tag{3}$$

where k indexes the time step of the algorithm, T_{c_k} is a small neighborhood of nodes in the grid within a radius c_k around the node most activated by x_k, and α_k and c_k are decreasing functions of k. The basic concept for updating the spread parameter is given by

$$s_{i,k+1} = \begin{cases} s_{i,k} + \eta_k(|x_{j,k} - m_{ij,k}|^2 - s_{i,k}) & \text{for } i \in T_{c_k} \\ s_{i,k} & \text{otherwise} \end{cases} \tag{4}$$

where η_k is also a decreasing function of k.

The SFDN provides a means of aggregating similar plant states, thus permitting implementation of the control as a discrete relation. The adaptation update equations as initially configured are of the simple delta-rule type, which is more easily implemented in real time than clustering algorithms and permits the parallel distributed computation of neural networks.

2.2 Fuzzy Correlation Network (FCN)

The Fuzzy Correlation Network implements the fuzzy control rules as a fuzzy relation G (learned by the SLCN) which associates the collection of fuzzy sets $X_1, \ldots, X_r$ for input vectors $x \in X$ with the fuzzy sets $U_1, \ldots, U_s$ for the controls $u \in U$. This is accomplished with a fuzzy associative memory (FAM) or correlation network. The i^th node on one side represents the degree to which X_i has been selected, given by the SFDN node output a_i for state x. Each of these is linked to every node on the other side. The output b_j of the j^th node on the other side indicates the degree to which fuzzy output set U_j is the correct choice given the activations of the X_i's, or the "firing strength" of each rule for which that fuzzy control is the consequent. The connection weight parameter g_ij indicates the degree to which X_i relates to U_j. The control rule "If x is X_i then u is U_j" is represented by a strong link g_ij between node X_i and node U_j. The network connection weight matrix $G = \{g_{ij}\}$ specfies a fuzzy relation on $\{X_1, \ldots, X_r\} \times \{U_1, \ldots, U_s\}$. Thus, given input activation a, the fuzzy hyperstate, the fuzzy control vector is given by

$$b^T = \sigma(a^T G) \tag{5}$$

where σ is a vector-valued function whose components are $b_j = \sigma_j(a^T G\bullet, j), G\bullet, j$ is the j^th column of G, and each σ_j is some type of limiting function, (i.e., σ_i satisfies $\sigma_i(\alpha) \to 0$ as $\alpha \to -\infty$ and $\sigma_i(\alpha) \to 1$ as $\alpha \to \infty$), such as a sigmoid function. Thus, this implementation uses the product and limited-sum logic operators which is more easily implemented with a neural network associative memory than the common min-max logic.

2.3 Stochastic Learning Correlation Network (SLCN)

The purpose of this subsystem is to test and learn the effectiveness of pairing a particular control vector fuzzy set with each given state vector fuzzy set, using the performance evaluation provided by the Performance Evaluation System (PES), then use this knowledge base to generate the fuzzy control relation used by the FCN. Both the SLCN and the FCN receive as input the fuzzy hyperstate a output by the state map grid of the SFDN. The first phase of operation of the controller is a learning phase in which the fuzzy control vector b is generated by the SLCN. In the second phase of operation, the fuzzy relation learned by the SLCN is used by the FCN to generate b.

Initially, nothing is known about what control vector gives the best response when the process is in a given state. So, a fuzzy output is just picked and the performance measure indicates how good the selection was. If it was not that good, the controller will be less inclined to pick that control again the next time the system enters that fuzzy state. If the selection was good, that control action is reinforced, so that it is more likely to be selected next time. The network that implements this is akin to Narendra's stochastic learning automata(SLA).

The SLCN consists of a matrix of nodes where each row corresponds to a particular fuzzy input state and each column to a particular fuzzy control action. The degree of activation of a node indicates the fuzzy degree to which it selects the control fuzzy subset to which it is assigned. Each node has a spread parameter h_ij for the i^th fuzzy state and the j^th fuzzy control. The location parameters λ_ij (scalar) are adjusted so that they always fall at the center of the spread function. In this case, a box-shaped function defined on a bounded interval is used. Thus, the output of the node for the i^th fuzzy state and the j^th fuzzy control is given by

$$c_{ij,t} = \begin{cases} 1 & \text{if} \lambda_{ij,t} - h_{ij,t} < \zeta_t < \lambda_{ij,t} + h_{ij,t} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$\lambda_{ij,t} = \sum_{k=1}^{j-1} h_{ij,t} + \frac{1}{2} h_{ij,t} \tag{7}$$

where $h_{ij,t}$ and $\lambda_{ij,t}$ are location and spread parameters, respectively, at time t for node (i, j), and $\zeta_t \in [0, 1]$ is generated by a chaotic or pseudo-random process and serves as the input to the node. The node with the largest spread parameter is then the one that will have the maximum activation most often. The fuzzy control vector $\mathbf{b}$ is given by

$$b_{j,t} = \begin{cases} c_{ij,t} & \text{for } i = \text{ argmax } (a_i) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

When i is the index of the most activated input fuzzy set, then the most activated node in the i^{th} row of the SLCN node matrix selects the control fuzzy set. The update algorithm for h_ij is given as

$$h_{ij,t+1} = (h_{ij,t} + r_t a_{i,t} b_{j,t} \gamma_{ij,t})/(1 + \sum_{j^r t} a_{i,t} b_{j,t} \gamma_{ij,t}) \tag{9}$$

$$\gamma_{ij,t} = \begin{cases} h_{ij,t} & \text{if } r_t < 0 \\ 1 - h_{ij,t} & \text{if } r_t \geq 0 \end{cases} \tag{10}$$

where r_t is the reinforcement which is a function of the performance measure p_t, and $a_{i,t}$ and $b_{i,t}$ are the input and output activations, respectively, for the i^{th} fuzzy state and j^{th} fuzzy control at time t. The product $a_{i,t}\, b_{j,t}$ is the association or correlation between the state and control fuzzy sets. The quantities p_t and r_t are computed by the PES subsystem.

2.4 Control Activation Network (CAN)

The input to the Control Activation Network is the fuzzy control vector b. This fuzzy control is defuzzified to produce a crisp output quantity u, which is a vector for multivariable control processes. Each CAN node has its location parameter vector set to the desired control vector prototype levels. Its input is the vector b, and its output is a crisp control vector u. The nodes can be set up as a map network to adapt the fuzzy sets according to the control vectors that are actually being output from the controller.

Using the max criterion defuzzification method, the B_i node with the largest activation (degree of truth) triggers the activation of the CAN node whose output is the prototype value $\bar{u}_j$ corresponding to the fuzzy set B_j. Alternatively, in the center of area method, u is calculated as the normalized weighted sum over the fuzzy sets B_i in which the weights are the activations b_j, given as

$$u = \left(\sum_{k=1}^{|U|} b_k s_k \bar{u}_k \right) / \sum_{k=1}^{|U|} b_k \tag{11}$$

where s_k are the spread parameters for the output fuzzy set membership functions. If the membership function form is symmetrical, then the effect of the spread is trivial.

2.5 Performance Evaluation System (PES)

The particular nature of the plant or process to be controlled and the characterization of the desired performance dictate the details of how the performance evaluation network is configured. When a performance measure is available which is an analytical function of the plant states or output, then the reinforcement signal r_t is simply a normalization of the performance measure p_t to lie in the interval [-1,1] or [0,1]. It is often the case, however, that complex processes which have no known well-defined plant model also do not have a well-defined formula for computing performance. Rather, there is a certain qualitative goal or objective to be reached, but it is not known what the values of the plant variables should be when that goal is reached. Even when a formula in terms of the variables is known, there is often an unknown delay between the control action taken and the effect on the plant variables, so that the result of the current control is not known until some future time. In such cases, various methods of estimating the performance evaluation function must be used. Our investigation into performance function estimation methods are reported elsewhere. In particular, we note the use of various dynamic programming-like algorithms such as the temporal difference (TD) algorithm by Murrell(1994) and both TD and Q-Learning by Esogbue et alia(1995).

The most straight-forward approach to the situation in which a qualitative determination of reaching the goal is given only after a period of many time intervals is described here.

Reaching the goal is indicated by $p_t = 1$ (success) and reaching forbidden states (such as plant shutdown due to variables out-of-bounds) is indicated by -1 (failure). For each state entered at each time, a control action is taken. The average performance over time of this state-control pair is computed and updated whenever there is a success or failure. The reinforcement signal r_t can then simply be the current value of this average for the current state-control pair that just occurred. This method was used with success in the earliest version of this controller.

3 Complexity Analysis of Controller Algorithms

To determine the size of the problem, one must consider the number of state space dimensions p and the number of control space dimensions q. Recall that the size of the spaces must be used in deciding the number of fuzzy subsets utilized in the fuzzy discretization of these spaces. As a consequence, the size of the problem in terms of the complexity of the controller algorithms, must also be expressed in terms of the number s of state space terms (i.e., the number of state space nodes N_i) and the number r of control terms.

The state space node map approach permits increasing the dimension of the input state vectors without an exponential increase in controller complexity. We note that other fuzzy discretization schemes generate a complete set of fuzzy subset terms for each dimension of the state, thereby greatly increasing the number of fuzzy rules as the dimension increases. In general, the usual number of rules is $l^P \bullet m^q$ with l and l the number of state terms per dimension and the number of control terms per dimension respectively. In the proposed method, the number of state terms s is the number of map nodes. This is ,in general, selected according to the size of the state space p so that every dimension of the state space can be adequately covered. However, s need not be set in such a way that it grows exponentially with p, since we can elect to cover the state space more sparsely than with a full factorial lattice. Since fuzzification is of the state space vectors rather than merely within each state dimension separately, the generalizing and interpolating characteristics of the fuzzy inference can exploit any smoothness or redundancy in information in all the dimensions simultaneously. Also, the adaptation of the location vectors and spreads tends to move the coverage of the membership function to regions where it is most needed in the state space. The number of rules is determined by the product sr, which can be chosen to grow with p and q more slowly than exponentially. Whether the number of nodes needs to be increased as the dimension increases is a matter of discretion involving a trade-off between the precision of the fuzzification and an increase in the number of nodes. In discussing the size of a problem , we must also consider the number of input data points. For on-line control, there is no fixed number of input data points, rather there is a potentially infinite sequence of data. However, there are some types of algorithms, such as recursive least squares or some types of adaptive clustering algorithms which would require either the complete past history or the history for some period into the past to be stored and used in the computations performed at each time step as a new data point becomes available. The usual clustering algorithms, for example, would require a pass through a set of past data for each additional point, greatly adding to the complexity of the computations as time advanced. Incremental parametric algorithms, including many neural network algorithms and the algorithms proposed here, do not have this drawback. Others such as those recently developedby Esogbue and Liu have attractive features that can be explored in future versions of the controller. Let us briefly address the complexity issues associated with the computation for one time step of the SFAL controller.

We begin with the SFDN algorithms. Computation of a_i requires s sets of calculations, one for each node; each of these includes operations that must be performed on each state dimension, i.e., p times. It has been determined that for one step of the SFDN algorithms, the number of computations is bounded above by $Ms + Nsp + L$ (for some integer l, M, and N), and thus the order of complexity is $O(sp)$.

In the SLFCN algorithm, while in principle, the quantities γ_{ij} require, sr computations, the simplified algorithm utilized by Murrell(1994) with γ_{ij} as an indicator, needs only r computations. On the average, even this can be made smaller. Obtaining all the b_j similarly requires sr, or simply r computations, depending on whether or not γ_{ij} indicates more than one node N_i. The c_{ij} require sr, or actually $s^{'}(k) \bullet r$ computations, where again $s^{'}(k)$ decreases with time down from s. The SLFCN therefore requires $Mr + Nsr + L$ operations $(L, M, N$ some integers, different from above),and as such has complexity $O(sr)$.

The IFCN requires only a few arithmetic operations performed sr times to obtain the g_{ij} from the c_{ij}, and sr computations to obtain $=^T$ leading to a complexity of $O(sr)$. The CAN similarly needs only r sets of simple arithmetic operations, while the PES needs only 1 small set of arithmetic operations per time step.

Thus, one iteration of the entire controller algorithm has a complexity of $O(sp+sr)$. Let us next address the problem of the number of iterations that may be required.

To learn the control law as a fuzzy relation matrix, requires that s pieces of information be learned or estimated. This is accomplished by a stochastic search of r^s combinations which could be of formidable complexity. However, the algorithm does not try each combination one at a time. Each SFDN node is a learning unit which explores only r possibilities, in parallel sequences of learning trials with all the other nodes. If the reinforcement signals were based on performance scores that were absolutely accurate and certain, then in principle, each control choice for each state need be visited exactly once, hence requiring only sr iterations of the controller algorithm. However, since the information is uncertain in a *Statistical* as well as a fuzzy sense, some multiple of sr is required to obtain an adequate sample of information. A conservative analysis shows that , with a probability greater than 0.98, the controller can generate a complete set of performance predictions in approximately $4r$ plant runs.

4 Implementation Issues for the Application of the Controller

In order to utilize this controller in its current configuration, the following conditions must be met:

1. Ability to characterize the process at any given time by its process state.

2. Ability of the control inputs to affect the sequence of input states.
3. Availability of a good performance measure which is related to the system goal
4. There must be some topological nearness measure (e.g., a metric) for both the state space $\mathbf{X}$ and control space $\mathbf{U}$ which additionally satisfies the smoothness assumptions listed below:
   (a) For every $\mathbf{x} \in \mathbf{X}$ there exists a control $\mathbf{u}^\star$ such that $y(\mathbf{x}, \mathbf{u}^\star) = \sup_{\mathbf{u} \in \mathbf{U}} \{y(\mathbf{x}, \mathbf{u}\}$.
   (b) If the optimal control for $\mathbf{x}_i$ is $\mathbf{u}^\star$ then the optimal control for every $\mathbf{x}$ in a neighborhood of $\mathbf{x}_i$ is near $\mathbf{u}^\star$.
5. There must be a direct relationship between the degree to which a control action contributes to the final computed control and the probability of that action being successful when applied purely.
6. The process is either intrinsically recurrent or can be repeatedly restarted at random initial states.

We note that the nearness structure of the spaces stipulated in the foregoing conditions 4 can be imposed or arrived at via some transformation of the original spaces. In particular, it is neither necessary for the smoothness to be analytical differentiability nor the nearness to be in terms of a metric computed by a formula.

Let us now outline the steps to be followed when trying to apply the controller to a system or process which has met the conditions stipulated earlier. These are succintly summarized in the flowchart for the algorithm. 2.

As usual, we begin with the set-up and initialization of the controller for the particular process as outlined below:

1. Set the state space and control space boundaries.
2. Set the state map size and initial location and spread parameters.
3. Set the control terms (control space map).
4. Set initial correlation parameters.
5. Determine the sampling interval for the process.
6. Determine process events to be used to index algorithm steps for each algorithm:
   (a) Map node location update.
   (b) Map node location neighborhood decay.
   (c) Map node location update rate decay.
   (d) Spreads update.

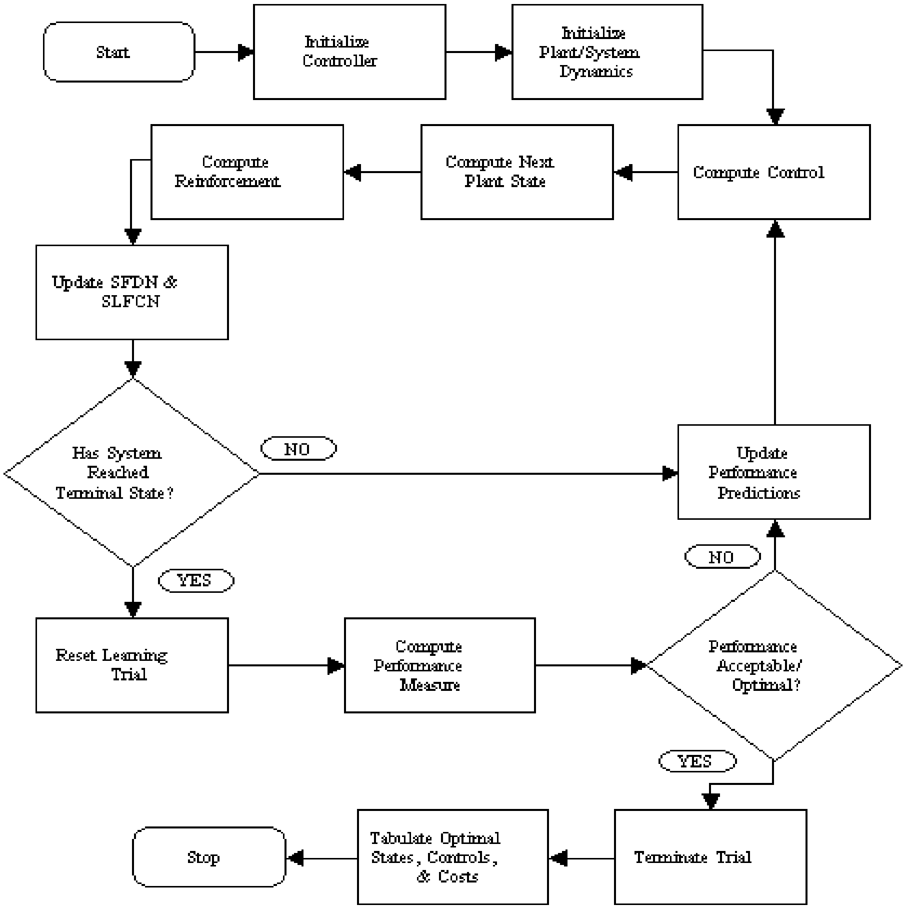

Start
Initialize Controller
Initialize Plant/System Dynamics
Compute Control
Compute Next Plant State
Compute Reinforcement
Update SFDN & SLFCN
Has System Reached Terminal State?
NO
YES
Update Performance Predictions
Reset Learning Trial
Compute Performance Measure
Performance Acceptable/ Optimal?
NO
YES
Terminate Trial
Tabulate Optimal States, Controls, & Costs
Stop

Figure 2: Flowchart for Implementation of Fuzzy-Neuro Controller

   (e) Correlation parameter update.
   (f) Correlation neighborhood decay.
   (g) Performance evaluation update.

7. Set controller learning parameters.

8. Determine a normalized performance measure to be computed at terminal states.

As in most algorithms, it is beneficial to have some basic knowledge or information about the process such as its state and control variables as well their ranges. Using this knowledge, the state and control space limits used by the controller are determined. Determining the number of nodes, the size of the crisp neighborhoods, and the basic level of fuzziness or dispersion for the membership functions requires some reasoning about what level of precision is appropriate for the process. The size of the state space region to be covered and what level of resolution is required to resolve the control switching surfaces must be considered. Ability to partition both the state and control spaces so that they can be fuzzy discretized is essential. This relates to the issues of continuity and nearness discussed above. According to condition 4b the control behavior for a locality in the state space must be generalizable throughout the neighborhood.

When no prior information is available, the map node locations are set at the points of a uniformly spaced grid. If it is impractical to fill all the points of the grid, then design of experiments methods can be used to determine a good subset of the full factorial coverage. The nodes can be placed with greater density in regions where more coverage is needed if there is any prior information. Similarly, if there is no prior information about the control relation, then one can set the correlation parameter vectors to be uniform probability distributions. Any prior information can be encoded by biasing the distribution vectors towards favored control actions.

The operation then consists of repeated cycles of computing controls to input to the plant, plant state transitions, computing reinforcements, and performing learning updates. Whenever a terminal state is reached, such as a goal state or set-point of the problem, then a performance measure is computed and provided to the controller Performance Evaluation System in order to update its performance prediction algorithm. The system is reset—i.e., a new initial state is determined and any algorithm counters, etc., are reset. The operation terminates if it is determined that the attained performance is acceptable or near optimal.

In general, the most ideal problem situations for applying the controller are those in which there exist a high tolerance for mistakes to be made for a brief period early in the learning phase or which allow for initial learning with a simulated on-line system.

5 Controller Application and Performance on Sample Problems

5.1 Application to Inverted Pendulum Problem

A classic testbed problem for nonlinear controllers is the inverted pendulum problem. One of the simplest inherently unstable systems known, yet it has a broad base for comparison throughout the literature. The system is shown in the following Figure 3.

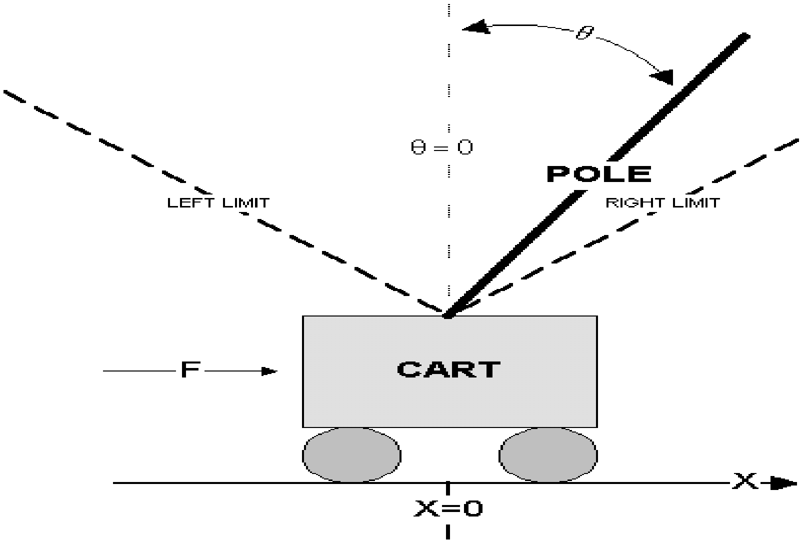


Figure 3: The Pole Cart System

5.1.1 Model of Process

Mathematically, the system is described as follows: **Inputs:** State vector—$\mathbf{x} = [\theta, \Delta\theta]$.

Outputs: Force applied (in Newtons)—$\mathbf{u} = [F]$ where $F > 0$ denotes a force applied in the positive x direction.

Equations of Motion: These equations serve only to simulate the system and are not used in the derivation of the control law:

$$\ddot{\theta} = \frac{g\sin\theta + \cos\theta\left(\frac{-F - m_p l\dot{\theta}^2\sin\theta + \mu_p \mathrm{sgn}(\dot{x})}{m_p+m_c}\right) - \frac{\mu_p\dot{\theta}}{m_p l}}{l\left(\frac{4}{3} - \frac{m_p\cos^2\theta}{m_p+m_c}\right)} \tag{12}$$

$$\ddot{x} = \frac{F + m_p l\left(\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right) - \mu_c \mathrm{sgn}\left(\dot{x}\right)}{m_p + m_c} \tag{13}$$

where the variables are defined as g: acceleration due to gravity, m_c: mass of the cart (kg), m_p: mass of the pole (kg), l: half-length of pole (m), μ_c: coefficient of friction for cart (N), and μ_p is the coefficient of friction for pole (N) with values shown in Table 1.

$g = 9.8\text{m/s}^2$	$m_c = 1.0\text{kg}$
$m_p = 0.1\text{kg}$	$l = 0.5\text{m}$
$\mu_c = 0.0\text{N}$	$\mu_p = 0.0\text{N}$

Table 1: Parameters of Inverted Pendulum System Simulation.

Success and Failure: The setpoint for the inverted pendulum problem is $[0, 0]$. Failure occurs at either of the following conditions:

$$\begin{aligned} \mid \theta \mid &> 12^\circ \\ \mid \Delta\theta \mid &> 25^\circ/\text{s} \end{aligned}$$

In a dynamical system problem such as the inverted pendulum under investigation, the system is always reset upon failure (exceeding state space boundaries), and may be reset upon reaching the goal state; in all the experiments reported in this study for this type of system, reset is not performed upon reaching a goal state. Also, new initial states of the system are selected randomly. The results of a representative run of the simulations are depicted in the following figures of the trajectory 4 and the control surface 5.

5.2 Application to Communication Networks

The controller's performance was further tested on the communications network problem using first a three node problem and then a larger ten node problem. The configuration for the later is shown in figure 6. Here, reset is upon arrival of the current message to its destination, and determining a new initial state consists of observing where the next message arrives in the network. No failure state is defined for the routing problem since every message eventually reaches its destination (network protocols prevent endless cycling).

5.2.1 Comparison with Other Network Methods

The performance of the controller in the sample networks was compared to several other message routing approaches known in the literature. A rigorous statistical analyis of the results indicate acceptable performance which is equal to and superior in instances to the approaches in question. These results are reported in detail in Murrell [31]. A graphical representation of these results where the 89 percent and 95 percent confidence intervals show that both the controller and the dynamic shortest path algorithms have statistically equal performance which is better than that attained by both the SLA and random controllers. This is depicted in figure 7

5.3 Application to Power System Stabilization Problems

One of the most intriguing and frequently investigated problem areas for deploying potent and novel tools of control engineering is the power system stabilization problem. Many different control strategies as well as controllers have been tested on this problem. Part of

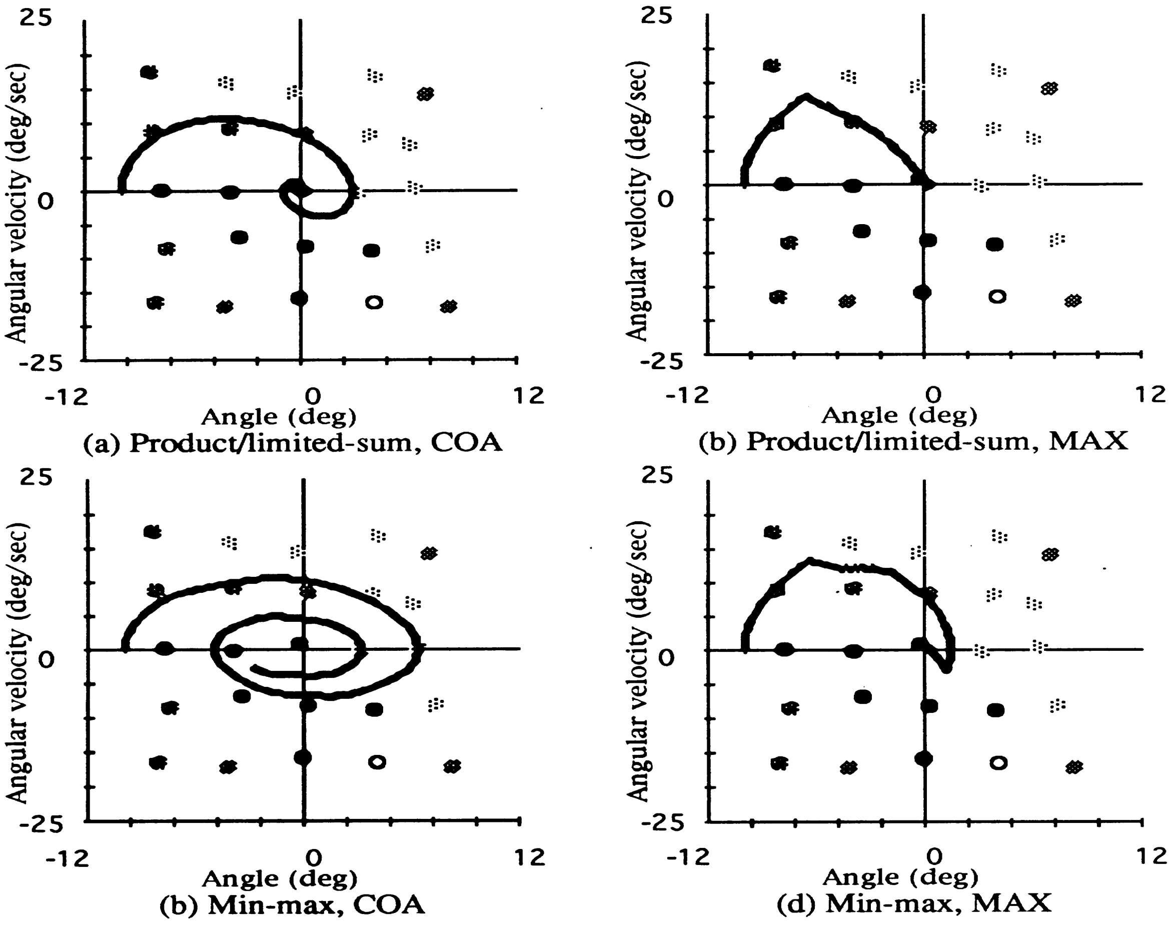


Figure 4: Control Trajectory for the Pole Cart System

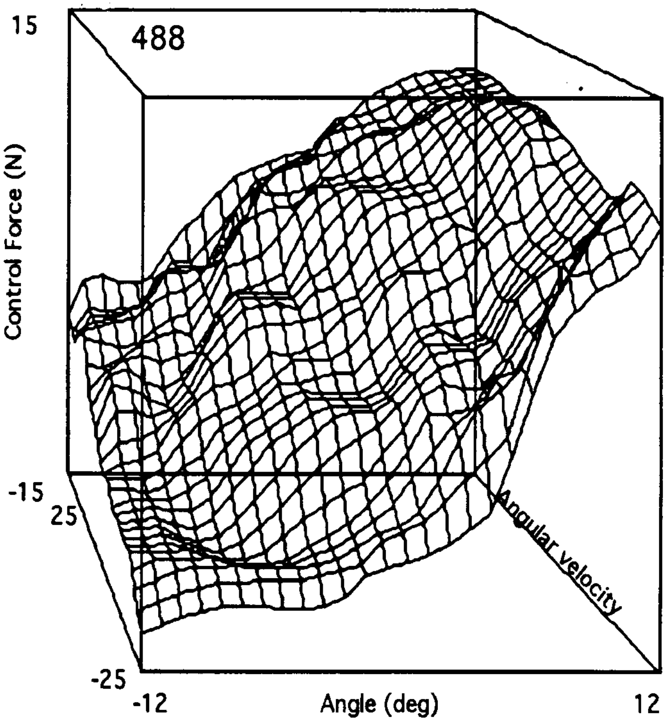


Figure 5: The Control Surface for the Pole Cart System

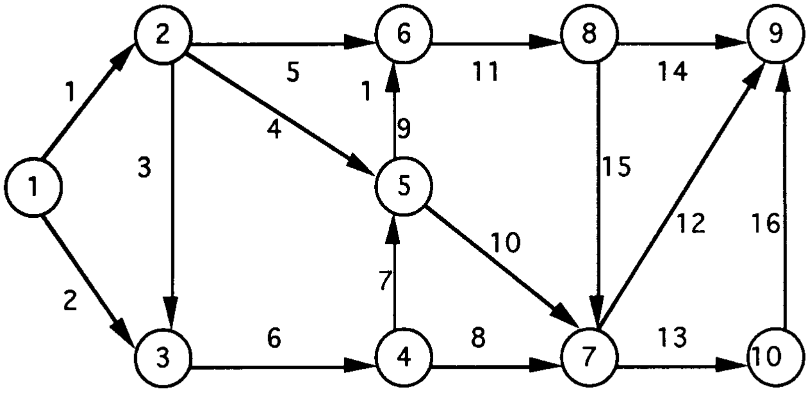


Figure 6: A Ten Node Communication Network Problem

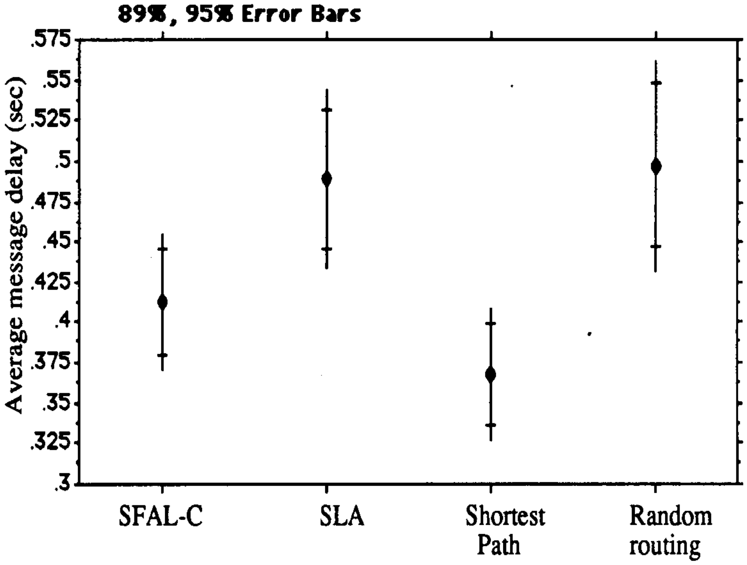


Figure 7: Average Delay Statistics of 4 Controllers for the Network Problem

this interest is engendered by both the challenge and intractability of power systems which are characterized by the existence of power inherently complex, nonlinear, time-varying and indeterminable elements, simple controllers which work well in one situation may not perform equally well in another. As part of the experimental ivestigations with the SFAL controller, we explored its ability to learn a robust control law to stabilize the power system under various operating conditions.

The earliest stabilizers consisted of a lead-lag analog circuit with the speed as the input. Such a simple controller cannot satisfy the high standards of the power system. PID controllers [20] perform better than the lead-lag circuit. Yet, unless their parameters are tuned automatically as operating conditions change, PID controllers in general do not work satisfactorily within a wide range of conditions. The self-tuning controller [8, 16, 26, 25] and the adaptive controller [6, 7, 22] are designed for this purpose. By continuously identifying the model of the plant, the self-tuning controller adjusts its parameters to achieve optimal performance, while the adaptive controller adjusts its parameters based on the knowledge of the plant model. These two controller paradigms are time-consuming in design but can perform well under different operating conditions. Most recently, fuzzy logic controllers [18, 19, 21, 25, 36] have been successfully applied to stabilize power systems. It has been found that the fuzzy logic controller performs as well as the self-tuning controller in power system stabilization [25] and shows great potential for application in power systems. When the system is of large scale and of high complexity, however, it is not easy to extract the control rules from human expert(s) [23] and, even if this can be done, the expert's experience is still limited. Therefore, it is necessary to design a controller that can "learn" the control law via its own experience.

6 Mathematical Models of the Power System

The system considered here is composed of a synchronous machine with an exciter and a stabilizer connected to an infinite bus. The dynamics of the synchronous machine can be expressed as follows using the linearized incremental model [20]. These equations serve only to simulate the system and are not used in the derivation of the control law:

$$
\begin{aligned}
\Delta\dot{\omega} &= \frac{1}{M}(\Delta T_m - \Delta T_e - \\
&\quad \Delta T_L - D\Delta\omega) && (14)\\
\Delta\dot{\delta} &= 377\Delta\omega && (15)\\
\Delta T_e &= K_e\Delta\delta + K_2\Delta e_q && (16)\\
\Delta\dot{e}_q &= \frac{1}{K_3 T_{de}}(K_3\Delta e_{fd} - \\
&\quad K_3K_4\Delta\delta - \Delta e_q) && (17)\\
\Delta V_t &= K_5\Delta\delta + K_6\Delta e_q && (18)\\
\Delta\dot{V}_f &= \frac{1}{T_F}\left(K_f\Delta\dot{e}_{fd} - \Delta V_F\right) && (19)\\
\Delta\dot{e}_{fd} &= \frac{1}{T_E}\left(\Delta V_A - K_E\Delta e_{fd}\right) && (20)\\
\Delta\dot{V}_A &= \frac{1}{T_A}(K_A\Delta V_{ref} - K_A\Delta V_F + K_A u - \\
&\quad K_AK_6\Delta e_q - K_AK_5\Delta\delta - \Delta V_A) && (21)\\
\mid u \mid &\leq u_{\max} && (22)
\end{aligned}
$$

where

V_{ref}	constant reference input voltage
ΔV_t	terminal voltage change,
ΔV_o	infinite bus voltage change
Δe_{fd}	equivalent excitation voltage change
Δe_q	q-axis component voltage behind transient reactance change
ΔV_F	stabilizing transformer voltage change
u	stabilizer output
ΔT_m	mechanical input change
ΔT_e	energy conversion torque change
ΔT_L	load demand change
$\Delta\delta$	torque angle deviation,
$\Delta\omega$	angular velocity deviation
K_A, K_E	voltage regulator gains
T_A, T_E	voltage regulator time constants

K_F	stabilizing transformer gain
T_F	stabilizing transformer time constant
$K_1, \ldots, K_6$	constants of the linearized model of synchronous machine
T_{do}	d-axis transient open circuit time constant
M	inertia coefficient
D	damping coefficient
T_s	sampling period

The objective of the controller is to drive the state of the system $\vec{x}$ to $[0, 0]$ via the stabilizer output u. The values for the above parameters are given in Table 2 below.

$K_1 = 1.4479$	$K_2 = 1.3174$	$K_3 = 0.3072$
$K_4 = 1.8050$	$K_5 = 0.0294$	$K_6 = 0.5257$
$K_A = 400$	$T_F = 1.0$	$T_A = 0.05$
$D = 0$	$T_{do} = 5.9$	$K_E = -0.17$
$M = 4.74$	$T_E = 0.95$	$K_F = 0.025$
$\Delta T_m = 0$	$\Delta V_{ref} = 0$	$T_s = 0.01$

Table 2: Parameters of Simulation.

7 Simulation Results

The experiments run on the power system stabilization problem consisted of multiple replications of the learning phase of the controller on a simulated power system written in C. The inputs to the controller are $\Delta\omega$ and $\Delta\dot{\omega}$. Thus, the controller in effect mimics a PD-like controller with unknown structure. The state space is defined as $\Delta\omega \in [-0.012, 0.012]$ and $\Delta\dot{\omega} \in [-0.025, 0.025]$. The number of nodes for the SFDN is set at $N = 25$ and there are 5 reference control fuzzy sets defined for $u \in [-0.12, 0.12]$. Once the controller has completed the learning phase, it is used as a stabilizer in the system.

Several experiments were run and an example of the resulting controller is shown in the figures below. Figure 8 shows the transient process of $\Delta\omega$ when the load increases 0.05 pu and 0.3 pu, respectively. It takes about 2 seconds for the speed deviation $\Delta\omega$ to vanish for the 0.05 pu load change and about 3 seconds for the 0.3 pu load change. Figure 9 shows the learned control surface using product–limited sum inference and center-of-area defuzzification.

7.1 Comparison to Existing Controllers

The simulation results clearly showed that the controller can learn an effective control law to stabilize the system under varying load conditions. However, the results, are not optimal with

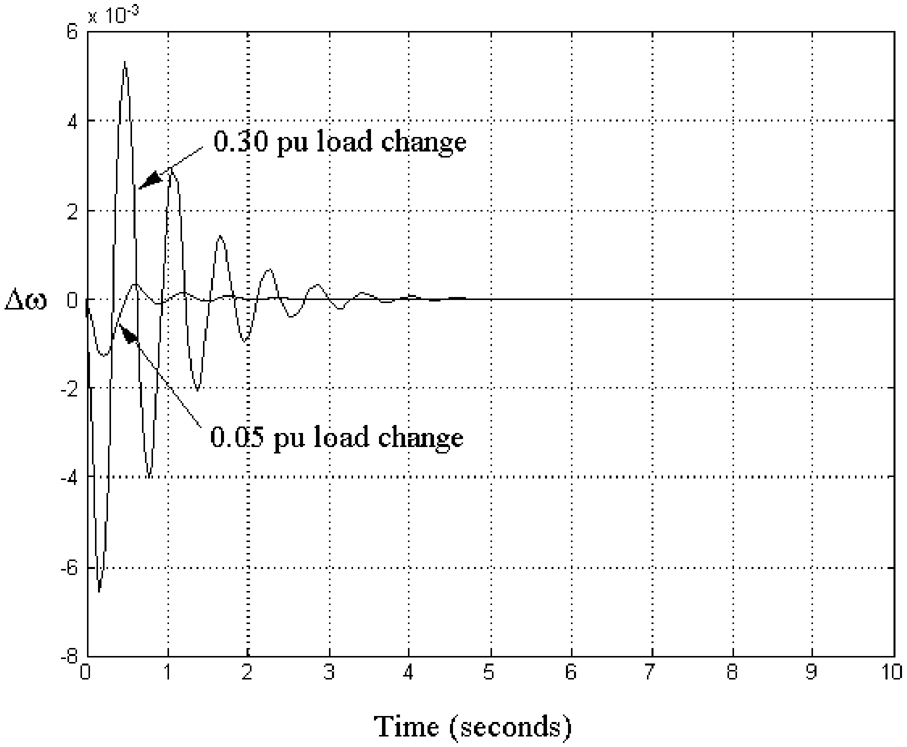


Figure 8: Transient Process for Selected Load Changes.

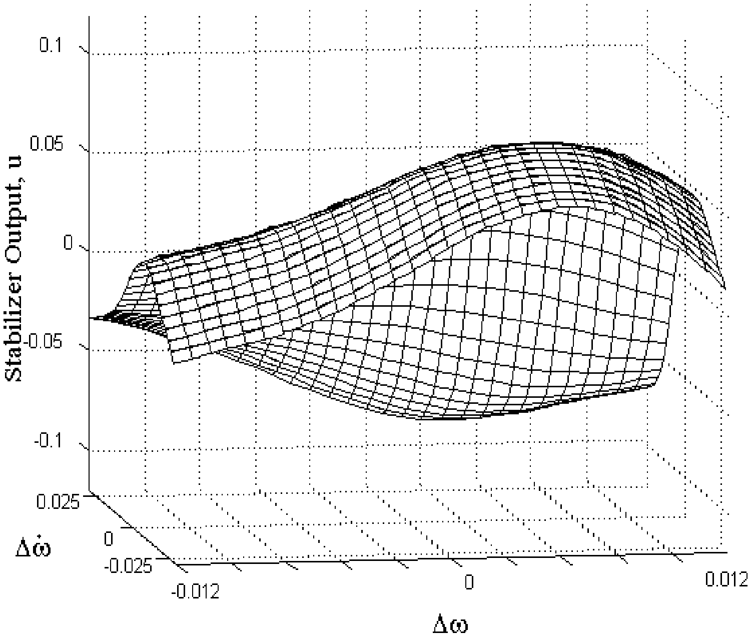


Figure 9: Learned Control Surface for Stabilization.

regard to settling time. The settling time obtained with our controller was slightly longer than the results obtained using an existing PID controller [20] and a fuzzy controller [21], but comparable to or shorter than the settling time for other fuzzy controllers [18, 19, 36] reported in the literature. The optimality issue (see Section 7.2) is currently under investigation, but the relative ease of developing an "efficient" controller via the self-learning controller with respect to the existing methods illustrates the potential of this approach.

Despite the foregoing, the advantages of our controller over other controllers that have been applied to the power systems stabilization problem are very significant and should be noted:

- The controller successfully learned the control law via its own experience. It did not require the analytic solution of a dynamical model, the tuning of parameters as in PID control, and it did not rely on existing expert knowledge about the control of the process.
- The learning phase of the controller took less than 5 minutes to complete. Thus, there is a huge time savings in development over the existing controllers.
- The internal controller parameters (that control the learning and other attributes) are very robust—the parameters used for the inverted pendulum problem were used for the power system stabilization problem. No tuning of these parameters was performed, although doing so may improve the resulting control.
- The resulting control is robust—the controller can handle a more extreme range of load changes than the PID controller [20].

7.2 Reinforcement Learning and Optimal Control

The lack of optimality with respect to settling time is not unexpected since the controller, which learns via reinforcements, is only given one goal: *Drive the power system to its set point* $\vec{x} = [0, 0]$. The only reinforcements that are given are upon *success* or *failure* of the plant and a desired trajectory through the state space is not defined. These external reinforcements are used to update an internal prediction function for each maximally activated node n_t at time t via $P_{t+1}(n_t) = P_t(n_t) + \alpha(P_t(n_{t+1}) - P_t(n_t))$, similar to Sutton's method of temporal differences. Thus, only by trial and error does the controller learn to drive the system to the set point, and it does so without requiring (and therefore without necessarily satisfying) a performance objective function.

The two controllers that performed better [21, 20] both utilized external information about the plant or the behavior of the controller. This additional information about the plant dynamics and desired control behavior could possibly be used by our controller to develop near-optimal control automatically. For example, knowing the desired trajectory of the power system through the state space allows the controller to give itself *additional* external reinforcement about its behavior, thereby changing $P_t(n_t)$ and the learned control law.

8 Unique Features

This controller has several unique features mentioned earlier which we summarize here. It is adaptive and well suited to the control of complex processes. In particular, it is capable of learning effective control using process data and improving its control through on-line adaption. The controller performs a fuzzy discretization of the state and control spaces and learns the fuzzy relations for these fuzzy subsets using a variation of the TD method with its dynamic programming inspirations. While it adapts both the membership functions and the control rule state-control association, the controller primarily learns the control rule associations, unlike many other methods which fix the rules and adjust the membership functions. Most important, it does so for the entire state space. Additionally, no training data sets nor any error signal derived from knowledge of the desired plant trajectory are needed. This self-learning controller has been successfully applied to the inverted pendulum problem, the DC servomotor position control problem, the switching problem in a distributed communications network, and the power system stabilization problem.

EDIT & INTEGRATE

To repeat the summarization presented in Murrell(1994), one of the principal developments of this research includes a feasible and practical tool for the application of fuzzy generalization to the discrete rules or associations of action -response utilized in reinforcement learning methods. Another important and novel characteristic of the controller is the manner in which it integrates the information derived from the fuzzy associative reinforcement learning process into the adaption mechanism for the fuzzy discretization pre-processing transformation. The resultant effect is that the location of the fuzzy clusters is both a function of the distribution of the states as well as the regions in the state space which are most significant in determining the appropriate controls. Clearly, the importance of the location of these fuzzy clusters in the optimal development of the controller can not be over-emphasized and hence the role of optimal fuzzy clustering techniques in future enhancements of the performance of the controller. For work in this arena which is beyond the scope of this project, see the Appendix where the developments of Esogbue and Liu(1996) can be found.

9 Conclusions

We have reported the development an intelligent controller with many unique features and successfully applied it, with various modifications, to an array of problems. Of note are: Esogbue and Murrell, 1993a; Esogbue and Murrell, 1993b; Murrell, 1993; Esogbue and Murrell, 1994; Esogbue and Hearnes, 1995; Esogbue, Hearnes and Song, 1995). Further investigations and extensions are underway. These include the use of more intelligent reinforcement strategies especially those that are DP-based algorithms useful in learning real-time control strategies. Of interest are Watkin's Q-learning algorithm and its variations and hybridization of the controller involving dynamic switching between the reinforcement controller and a stabilizing controller.

10 Acknowledgements

The contributions of several graduate and undergraduate students including James Murrell, Warren Hearnes, and Qiang Song who worked on various aspects of this project that are reported here, under funding provided principally by the National science Foundation under Grant No. ECS 91216004 and the Electic Power Research Institute under Grant No RP 8030-16 are gratefully acknowledged.

References

[1] Barto, A.G., Bradtke, S.J., and Singh, S.P. Learning to act using real-time dynamic programming. *Artificial Intelligence*, **72:1-2**, 1995, 81-138.

[2] Berenji, H.R. A reinforcement learning-based architecture for fuzzy logic control. *International Journal of Approximate Reasoning*, **6:2**, 1992, 267-292.

[3] Berenji, H.R. Fuzzy Q-learning: a new approach for fuzzy dynamic programming. *Proceedings of the Third IEEE Conference on Fuzzy Systems*, Orlando, FL, June 26-29, 1994, 486-491.

[4] Berenji, H.R. and Khedkar, P. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on Neural Networks*, 1992, **3:5**, 724-740.

[5] Berenji, H.R. and Ralescu, A.L. Fuzzy reinforcement learning and dynamic programming. *Proceedings of the Second Workshop on Fuzzy Logic in Artificial Intelligence*, Chamberry, France, August 28, 1993, 1-9.

[6] Cheng, C.-H. and Hsu, Y.-H. Damping of generator oscillations using an adaptive static VAR compensator. *IEEE Transactions on Power Systems*, **7:2**, 1992, 718-725.

[7] Cheng, S.-J., Chow, Y.S., Malik, O.P and Hope, G.S. An adaptive synchronous machine stabilizer. *IEEE Transactions on Power Systems*, **PWRS-1:3**, 1986, 101-107.

[8] Cheng, S.-J., Malik, O.P and Hope, G.S. Self-tuning stabilizer for a multimachine power system. *IEE Proceedings*, **133-C:4**, 1986, 176-185.

[9] Esogbue, A.O. Optimal clustering of fuzzy data via fuzzy dynamic programming. *Fuzzy Sets and Systems*, **18**, 1986, 283-298.

[10] Esogbue, A.O. and Hearnes, W.E. Constructive experiments with a new fuzzy adaptive controller. *NAFIPS/IFIS/NASA '94. Proceedings of the First International Joint Conference of the North American Fuzzy Information Processing Society Biannual Conference. The Industrial Fuzzy Control and Intelligent Systems Conference, and the NASA Joint Technology Workshop on Neural Networks and Fuzzy Logic*, San Antonio, TX, December 18-21, 1994, 377-380.

[11] Esogbue, A.O., Hearnes, W.E.and Song, Q. A reinforcement learning fuzzy controller for general set-point regulator problems. *IEEE Transactions on SMC,* Submitted 1995.

[12] Esogbue, A.O. and Murrell, J.A. A fuzzy adaptive controller using reinforcement learning neural networks. *Proceedings of Second IEEE International Conference on Fuzzy Systems*, March 28–April 1, 1993, 178-183.

[13] Esogbue, A.O. and Murrell, J.A. Advances in fuzzy adaptive control. *Computers & Mathematics with Applications*, **27:9-10**, 1994, 29-35.

[14] Esogbue, A.O. and Song, Q. Optimal defuzzification and applications. *Proceedings of the International Joint Conference on Information Science, Fourth Annual Conference on Fuzzy Theory & Technology*, Wrightsville Beach, NC, September 28-October 1, 1995.

[15] Esogbue, A.O., Song, Q. and Hearnes, W. E. Application of a self-learning controller to the power system stabilization problem. *Proceedings of the 1995 World Conference on Neural Networks* Washington, D.C. **II** 699-700.

[16] Ghandakly, A.A. and Farhoud, A.M. A parametrically optimized self-tuning regulator for power system stabilizers. *IEEE Transactions on Power Systems*, **7:3**, 1992, 1245-1250.

[17] Glorennec, P.Y. Fuzzy Q-learning and dynamical fuzzy Q-learning. *Proceedings of the Third IEEE Conference on Fuzzy Systems*, Orlando, FL, June 26-29, 1994, 474-479.

[18] Hassan, M.A.M., Malik, O.P and Hope, G.S. A fuzzy logic based stabilizer for a synchronous machine. *IEEE Transactions on Energy Conversion*, **6:3**, 1991, 407-413.

[19] Hiyama, T. and Sameshima, T. Fuzzy logic control scheme for on-line stabilization of multi-machine power system. *Fuzzy Sets and Systems*, **39**, 1991, 181-194.

[20] Hsu, Y.-Y. and Hsu, C.-Y. Design of a proportional-integral power system stabilizer. *IEEE Transactions on Power Systems*, **PWRS-1:2**, 1986, 46-53.

[21] Hsu, Y.-Y. and Cheng, C.-H. A fuzzy controller for generator excitation control. *IEEE Transactions on Systems, Man and Cybernetics*, **23:2**, 1993, 532-539.

[22] Irving, E., Barret, J.P., Charcossey, C. and Monville, J.P. Improving power network stability and unit stress with adaptive generator control. *Automatica*, **15**, 1979, 31-46.

[23] Jang, J.-S.R. ANFIS: Adaptive-network-Based fuzzy inference system. *IEEE Transactions on Systems, Man and Cybernetics*, **23:3**, 1993.

[24] Kaymak, U. and Babuska, R. Compatible cluster merging for fuzzy modelling. *Proceedings of 1995 IEEE International Conference on Fuzzy Systems. The International Joint Conference of the Fourth IEEE International Conference on Fuzzy Systems and The Second International Fuzzy Engineering Symposium*, Yokohama, Japan, March 20-24, 1995, 897-904.

[25] Lim, C.M. and Hiyama, T. Comparison study between a fuzzy logic stabilizer and a self-tuning stabilizer. *Computers in Industry*, **21**, 1993, 199-215.

[26] Lim, C.M. and Hiyama, T. Self-tuning control scheme for stability enhancement of multimachine power systems. *IEE Proceedings*, **137-C:4**, 1990, 269-275.

[27] Lin, C.-T. and Lee, C.S.G. Reinforcement structure/parameter learning for neural-network-based fuzzy logic control systems. *IEEE Transactions on Fuzzy Systems*, **2:1**, 1994, 46-63.

[28] Liu, B. and Esogbue, A.O. Optimal fuzzy criterion clustering based on fuzzy prototypes. *Proc. 1995 IEEE International Conference on Systems, Man, and Cybernetics*, Vancouver, Canada, 1995, 4702-4705.

[29] Routing,flow control and learning algorithms. em First IEEE National Conference on UK Telecommunication Networks - Present and Future London, England, 1987, 78-83.

[30] Maeda, M., Sato, T., and Murakami, S. Design of the self-tuning fuzzy controller. *Proceedings of the International Conference on Fuzzy Logic and Neural Networks*, Iizuka, Japan, 1990, 393-396.

[31] Murrell, J. A. *A statistical fuzzy associative learning approach to intelligent control*. Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, December 1993.

[32] Nakanishi, S., Takagi, T., Unehara, K., and Gotoh, Y. Self-organizing fuzzy controllers by neural networks. *Proceedings of the International Conference on Fuzzy Logic and Neural Networks*, Iizuka, Japan, 1990, 187-191.

[33] Nedzelnitsky, O. V., and Narenda, K. S. Nonstationary models of learning automata routing in data communication networks. em IEEE Trans. Syst. Man Cybern. **17:6**, 1987, 1004-1015.

[34] Patrikar, A. and Provence, J. A self-organizing controller for dynamic processes using neural networks. *Proceedings of the International Joint Conference on Neural Networks*, **3**, 1990, 359-364.

[35] Procyk, T.J. and Mamdani, E.H. A linguistic self-organizing process controller. *Automatica*, **15**, 1979, 15-30.

[36] Shi, J., Herron, L.H. and Kalam, A. A fuzzy logic controller applied to power system stabilizer for a synchronous machine power system. *Proceedings of IEEE Region 10 Conference, Tencon 92*, Melbourne, Australia, November 11-13, 1992.

[37] Sutton, R.S. Learning to predict by the method of temporal differences. *Machine Learning*, **3**, 1988, 9-44.

[38] Watkins, C.J.C.H. *Learning from delayed rewards*. Ph.D. Thesis, Cambridge University, Cambridge, England, 1989.

[39] Watkins, C.J.C.H. and Dayan, P. Q-Learning. *Machine Learning*, **8**, 1992, 279-292.

[40] Werbos, P.J. An overview of neural networks for control. *IEEE Control Systems Magazine*, **11:1**, 1991, 40-41.

[41] Whitehead, S.D. and Lin, L.-J. Reinforcement learning of non-Markov decision processes. *Artificial Intelligence*, **73:1-2**, 1995, 271-306.

[42] Yamaoka, M. and Mukaidono, M. A learning method of fuzzy inference rules with a neural network. *Proceedings of the International Fuzzy Systems Association, Fourth World Congress*, July, 1991.

[43] Zadeh, L.A. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-1**, 1973, 28-44.

[44] Zeng, S. and He, Y. Learning and tuning fuzzy logic controllers through genetic algorithm. *Proceedings of the 1994 IEEE International Conference on Neural Networks*, Orlando, FL, June 27-July 2, 1994, 1632-1637.

[45] Zomaya, A.Y. Reinforcement learning for the adaptive control of nonlinear systems. *IEEE Transactions on Systems, Man and Cybernetics*, **24:2**, 1994, 357-363.

A Learning Algorithm for the Control of Continuous Action Set-Point Regulator Systems

Augustine O. Esogbue and Warren E. Hearnes II
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0205
aesogbue@isye.gatech.edu
wp46268@west-point.org

September 16, 1998

Abstract

The convergence properties for reinforcement learning approaches such as temporal differences and Q-learning have been established under moderate assumptions for discrete state and action spaces. In practice, however, many systems have either continuous action spaces or a large number of discrete elements. This paper presents an approximate dynamic programming approach to reinforcement learning for continuous action set-point regulator problems which learns near-optimal control policies based on scalar performance measures. The Continuous Action Space (CAS) algorithm uses derivative-free line search methods to obtain the optimal action in the continuous space. The theoretical convergence properties of the algorithm are presented. Several heuristic stopping criteria are investigated and practical application is illustrated on two example problems–the inverted pendulum balancing problem and the power system stabilization problem.

1 Introduction

As control problems in real-world applications become more complex, the use of traditional analytical and statistical control techniques requiring mathematical models of the plant becomes less appealing and appropriate. In many cases, the assumption of certainty in the resultant models is made not so much for validity but the need to obtain simpler and more readily solvable formulations.

If a model of the system is known, then traditional well-developed theories of optimal control or adaptive control may be used. Without a reliable analytic model, however, these methods may not be adequate and a *model-free approach* is required. Model-free techniques learn the control law through either supervised or unsupervised means. Supervised learning requires some sort of a teacher or critic to provide the desired response at each time period. In some cases, however, an expert or quantitative input-output training data may be unavailable. Consequently, model-free control methods that can learn a control policy for a complex system through online experience have recently been proposed [1, 2, 3, 9, 10, 14, 17, 18]. These model-free reinforcement learning methods are increasingly being used as capable and potent learning algorithms in intelligent autonomous systems.

Reinforcement learning systems form effective control policies online through a systematic search of the action space in an environment which is possibly dynamic. Two major approaches to model-free reinforcement learning–specifically Sutton's method of temporal differences (TD) [17] and Watkin's Q-learning algorithm [18]–are online versions of classical dynamic programming approximation methods. Convergence properties for these algorithms have been derived for discrete state and action spaces [19, 4, 1]. In practice, however, many processes to be controlled have either continuous action spaces or a large number of discrete elements. Examples include stabilizing a power system under a load, controlling a multi-degree of freedom robot arm manipulator, and retrieving a tethered satellite into a spacecraft.

This paper presents an approximate dynamic programming approach to reinforcement learning for continuous action set-point regulator problems which learns near-optimal control policies based on scalar performance measures. The set-point regulation problem is reviewed in Section 2. The continuous action space (CAS) algorithm, developed in Section 3.1, uses derivative-free line search methods to obtain the optimal action in the continuous space using a dynamic discrete subset of the state space. Theoretical convergence properties and computational aspects of the algorithms are investigated in Section 3.2. The approach is then illustrated in Section 4 on two example set-point regulator systems–the inverted pendulum balancing problem and the power system stabilization problem.

2 The Set-Point Regulator Problem

We restrict our attention to the general class of set-point regulator problems. In these problems, a goal state, or set-point $\mathbf{s}^\star$, is defined. The objective is to drive the system from any initial state $\mathbf{s} \in \mathbf{S}$ to the set-point $\mathbf{s}^\star$ in an optimal manner with respect to some scalar return function.

2.1 Formulation

Consider a possibly stochastic dynamical system with scalar returns. The set of all possible states is represented by $\mathbf{S}$. In order to control the system P, some set of possible decisions or actions must also be available. The set of all possible actions is represented by $\mathbf{A}$. The state of the system at time step k is denoted generally by the m-vector $\mathbf{s}(k) \in \mathbf{S}$. The action taken at time step k is the n-vector $\mathbf{a}(k) \in \mathbf{A}$. The output state $\mathbf{s}(k+1)$ is defined by the transition function

$$\mathbf{s}(k+1) = \tau\left(\mathbf{s}(k), \mathbf{a}(k), \omega(k)\right) \tag{1}$$

where $k = 0, 1, \ldots$ and $\omega(k)$ is some random disturbance. Let the probability that $\mathbf{s}(k+1) = j$ when $\mathbf{s}(k) = i$ and $\mathbf{a}(k) = a$ be $p_{ij}(a)$.

We restrict our investigation to policies that are time-invariant. Therefore, for any given control policy $\pi : \mathbf{S} \to \mathbf{A}$, define the control objective function for an infinite horizon problem as

$$V_\pi(\mathbf{s}) = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R(\mathbf{s}(k), \mathbf{a}(k)) | \mathbf{s}(0) = \mathbf{s}\right] \quad \forall \mathbf{s} \in \mathbf{S} \tag{2}$$

and let

$$V(\mathbf{s}) = \mathrm{inf}_\pi V_\pi(\mathbf{s}) \quad \forall \mathbf{s} \in \mathbf{S} \tag{3}$$

where $\gamma \in [0, 1)$ is a discount factor and E_π is the conditional expectation using policy π. $V_\pi(i)$ represents the expected discounted total return using policy π and starting in state $\mathbf{s}$. A policy $\pi^\star$ is γ-optimal if

$$V_\pi(\mathbf{s}) = V(\mathbf{s}) \quad \forall \mathbf{s} \in \mathbf{S}. \tag{4}$$

Formulating the optimal control problem as a dynamic program, the functional equation becomes

$$V(\mathbf{s}) = \min_{\mathbf{a} \in \mathbf{A}}\left[R(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}' \in \mathbf{S}} p_{\mathbf{s}\mathbf{s}'}(\mathbf{a}) V(\mathbf{s}')\right] \quad \forall \mathbf{s} \in \mathbf{S} \tag{5}$$

which represents the minimum expected discounted return when starting in state $\mathbf{s}$ and always following an optimal policy.

The existence of an optimal stationary policy is guaranteed for the discounted case if the return function $R(\mathbf{s}, \mathbf{a})$ is bounded below by 0 and above by some number B for all $\mathbf{s}, \mathbf{a}$ [15]. If π is the stationary policy that chooses in state i the action minimizing

$$R(i, a) + \gamma \sum_{j=0}^{\infty} p_{ij}(a) V(j) \tag{6}$$

then π is γ-optimal [7].

2.2 Assumptions

The above formulation encompasses a variety of performance criteria including the popular performance minimization measures, such as sum of squared error (SSE), for reinforcement learning problems. For the application of the proposed algorithms, the following assumptions are made:

Assumption 2.1 *The return, $R(i, a)$, for action a taken in state i is determined immediately or in some fixed time period. Further, $R(i, a)$ is bounded below by 0 and above by some finite number B.*

Assumption 2.2 *The system is controllable. For every state* $\mathbf{s}$*, there exists a sequence of actions such that the system reaches* $\mathbf{s}^\star$ *in finite time with probability 1.*

These two assumptions ensure that there exists a policy π such that $V_\pi(\mathbf{s}) < \infty$ for all $\mathbf{s} \in \mathbf{S}$.

Assumption 2.3 *Multiple trials can be run on the system with both success and failure.*

Approximate dynamic programming methods rely on repeated experimental runs. Systems must be allowed to experience both success and failure without damaging the plant.

Assumption 2.4 *The transition function* $\tau(\mathbf{s}, \mathbf{a}, \omega)$ *is not known. Subsequently, the transition probability,* $p_{ij}(a)$*, from state* i *to state* j *when action* a *is taken is not known.*

The set-point regulation problem is a terminal control problem where the number of stages until the set-point is reached is not fixed. Rather, it is dependent upon the policy π. Therefore, the objective function is defined as in (5) over an infinite horizon. With an appropriate boundary condition of $V(\mathbf{s}^\star) = 0$, the problem is appropriately formulated.

3 Learning Optimal Continuous Actions

Dynamic programming determines the optimal solution to a variety of multi-stage decision problems by taking advantage of recurring imbedded subproblems, commonly referred to as Bellman's principle of optimality. In doing so, dynamic programming solves two related fundamental problems:

1. Determination of the optimal functional value $V(i)$ for each state i.

2. Determination of a policy $\pi^\star$ that achieves that value.

In many control problems, the optimal policy is the more important, and oftentimes the more readily available, part of the solution. The value of the functional equation when using an optimal policy may not necessarily be required. Unlike classical dynamic programming, in approximate dynamic programming the optimal policy and the optimal functional values are not necessarily determined in an identical computational effort. This key property is exploited in the Continuous Action Space (CAS) algorithm below.

Temporal differences and Q-learning, which have proved effective in online control problems, are based on finite state and action spaces. However, when either space is infinite their convergence is not necessarily guaranteed. The controller which we have proposed in [8] uses Q-learning as the evaluator for a discrete subset of actions within the continuous action space. We note that Watkin's Q-learning algorithm is an online version of successive approximations [18] which learns the particular value, termed the Q-value, of taking a specific action in a particular state.

The Q-values in the Q-learning algorithm may be defined so that they either learn the functional values for the optimal policy $\pi^\star$ with

$$Q(i,a) = R(i,a) + \gamma \sum_{j=1}^{S} p_{ij}(a)V(j) \tag{7}$$

and

$$V(i) = \min_j Q(i, \mathbf{a}_j) \tag{8}$$

or for a fixed policy π with

$$Q(i,a) = R(i,a) + \gamma \sum_{j\in\mathbf{S}} p_{ij}(a) \sum_h \xi_j^\pi(h) Q(j,h) \tag{9}$$

where $\xi_j^\pi(h)$ is the probability of choosing action h in state j using policy π. Under the optimal stationary policy $\pi^\star$, (9) reduces to (7). In Assumption 2.4 the transition probabilities $p_{ij}(a)$ are not known. The Q-values for each state-action pair are estimated by $Q_n(i,a)$ with the update equation

$$Q_{n+1}(i,a) = \alpha Q_n(i,a) + (1-\alpha)\left[R(i,a) + \gamma \min_h \{Q_n(j,h)\}\right] \tag{10}$$

for (7) and

$$Q_{n+1}(i,a) = \alpha Q_n(i,a) + (1-\alpha)\left[R(i,a) + \gamma \sum_h \xi_j^\pi(h) Q_n(j,h)\right] \tag{11}$$

for (9). The Q-learning algorithm converges with probability 1 to within ϵ of the optimal Q-values and, subsequently, the optimal functional values V if the system is controllable and there exists an absorbing state [18]. A number of other convergence proofs for Q-learning exist in the literature. Notable examples include [19, 4, 1]. In the sequel, we present an algorithm which exploits the convergence properties of Q-learning on discrete sets and nonlinear optimization methods to search for the optimal control in a continuous space.

3.1 CAS Algorithm

The Continuous Action Space (CAS) algorithm begins with a representative subset $\mathbf{a}_j, j = 1, \ldots, A$, of the action space $\mathbf{A}$ for each state i. This subset spans some interval of uncertainty (IoU) regarding the location of the optimal control action in the continuous action space. The Q-learning algorithm determines the optimal control policy given this action subset. Based on this policy, the interval of uncertainty is reduced for selected states, thereby adjusting the locations of the reference actions. As the CAS algorithm continues, the intervals of uncertainty for each state are reduced toward 0, centering on the optimal action in the continuous action space if certain assumptions are maintained.

The general CAS algorithm is as follows:

Algorithm 3.1

```
Set boundary condition: $V(\mathbf{s}^\star) = 0$.
Initialize $\mathbf{a}_j$ for all states.
Set $Q_0(i, \mathbf{a}_j) = M \gg 0 \quad \forall i, j$.
$n(i) \leftarrow 0 \ \forall i$
while $n(i) < N \ \forall i$ do
        Perform an iteration of $Q$-learning.
        if Policy doesn't change for state $i$
            $n(i) \leftarrow n(i) + 1$
    else
            $n(i) \leftarrow 0$
        fi
        if Reduction criteria is met for state $i$
            Reduce IoU by $\beta < 1$ around $\mathbf{a}_{j^\star}, j^\star = \text{argmin}_j Q_n(i, \mathbf{a}_j) \quad \forall i$ od
```

The choice of the reduction parameter β, learning rate α, threshold N, and initial Q-value M affects the rate of convergence. The properties of the CAS algorithm are examined in the next section.

3.2 Properties of the CAS Algorithm

The key to the efficiency of the CAS algorithm is that the optimal policy can, in many cases, be determined before the Q-learning algorithm converges to the optimal functional values. Basing the policy improvement procedure on this information is equivalent to waiting for the Q-learning algorithm to converge.

The Q-values from (7) are equivalent to a positive stochastic dynamic program and, therefore, an optimal stationary policy exists [7]. A stationary policy is one that is nonrandomized and is time-invariant. By Assumption 2.2, the set-point regulation problem is controllable. Defining the set-point $\mathbf{s}^\star$ as an absorbing state, the estimates $Q_n(i, \mathbf{a}_j)$ are guaranteed to converge to $Q(i, \mathbf{a}_j)$ as defined in (10) with probability 1 [18].

Theorem 3.1 *Given a Markov system with an absorbing state $\mathbf{s}^\star$ and a unique optimal stationary policy $\pi^\star$, there exists an $\epsilon > 0$ sufficiently small such that if*

$$|Q_n(i, a) - Q(i, a)| < \epsilon \quad \forall i, \forall a \tag{12}$$

for all $n \geq k$, there exists a $k' \leq k$ such that

$$\pi_n(i) \equiv \min_a Q_n(i, a) = \pi^\star(i) \quad \forall i \tag{13}$$

for all $n \geq k'$.

Proof: The Q-learning algorithm converges for this system. Therefore, (12) is satisfied for each $\epsilon > 0$ and a corresponding $k \in \mathbb{N}$. A unique optimal policy implies

$$m(i) = \min_{a \neq \pi^\star(i)} |Q(i,a) - Q(i,\pi^\star(i))| > 0 \quad \forall i. \tag{14}$$

Choose ϵ such that

$$0 < \epsilon < \min_i \frac{1}{2} m(i). \tag{15}$$

It follows from convergence that for $n \geq k$

$$|Q_n(i,a) - Q(i,a)| < \epsilon \quad \forall i, a. \tag{16}$$

The choice of $m(i)$ ensures that

$$Q(i,\pi^\star(i)) + \epsilon < Q(i,\pi^\star(i)) + \frac{1}{2}m(i) < Q(i,a) - \frac{1}{2}m(i) < Q(i,a) - \epsilon \quad \forall i, a \neq \pi^\star(i). \tag{17}$$

Therefore, for any estimate $Q_n(i,a)$ for $n \geq k$,

$$Q_n(i,\pi^\star(i)) < Q_n(i,a) \quad \forall a \neq \pi^\star(i), \forall i \tag{18}$$

and

$$\pi_n(i) = \pi^\star(i) \quad \forall i, \tag{19}$$

The optimal policy $\pi^\star$ is found in no more than k iterations. □

Theorem 3.1 provides a weak theoretical upper bound on the number of iterations until the ϵ-optimal policy is found. In practice, the ϵ-optimal policy may be found in *significantly* fewer iterations. Consider a discrete approximation to the inverted pendulum balancing problem with 625 discrete states. Using a full backup for each iteration, 479 iterations are necessary before the Q-values converges to within $\epsilon = 0.001$ of the optimal functional values. Yet, the optimal policy is actually determined after only 108 iterations, as shown in Figure 1–a 77.45% reduction in computational effort. Figure 2 illustrates the convergence of the maximum change in the $Q_k(i,a)$ approximation toward 0 as Q-learning progresses. The abrupt change in the function at iteration 108 occurs as the last policy change takes place. From that point onward, the standard Q-learning algorithm maintains a constant optimal policy but the approximation to the optimal value function is converging to the true values.

Each state i has an interval of uncertainty (IoU) in the continuous action space $\mathbf{A}$ which contains the true optimal action $\pi^\star(i)$. The estimates of the $Q(i,\mathbf{a}_j)$ values for each state i serve as a guide for reducing the interval of uncertainty. Initially, this interval is the entire action space $\mathbf{A}$. Each reduction is by a factor of $0 < \beta < 1$. Therefore, the interval can be made arbitrarily small using successive reductions. Let the reference action subset for state i be defined

$$\mathbf{A}_A(i) = \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_A\} \subset \mathbf{A}. \tag{20}$$

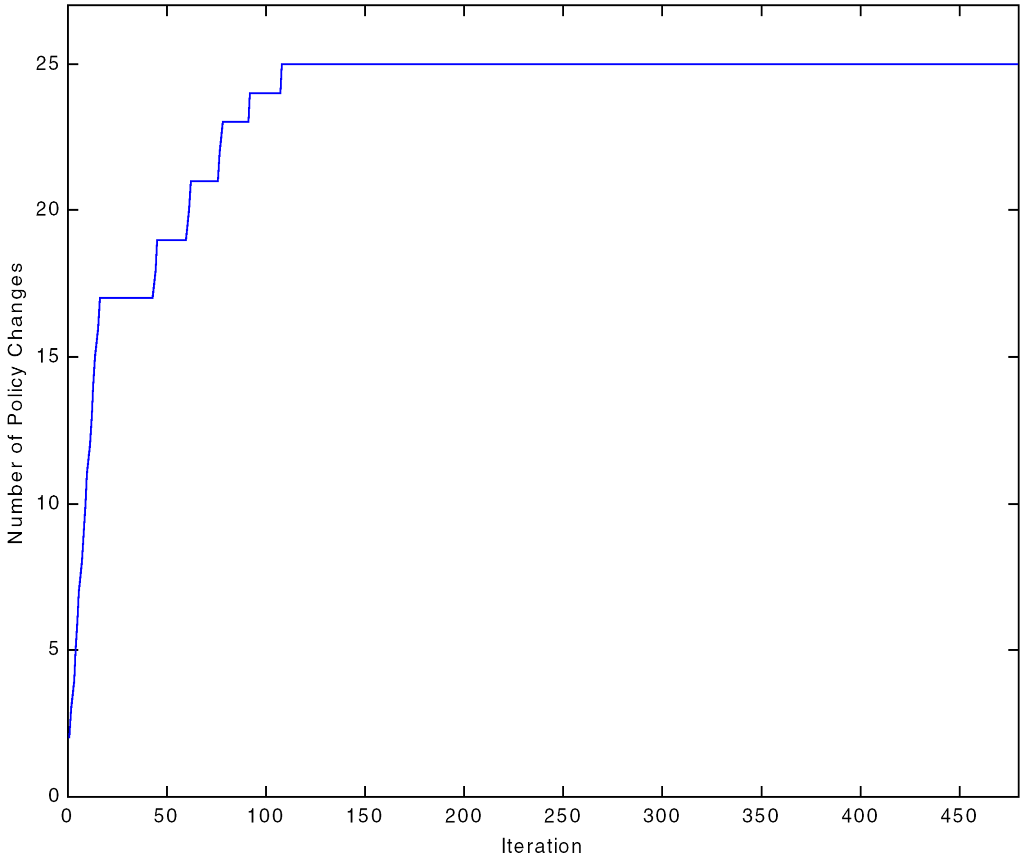


Figure 1: Optimal policy found in considerably fewer iterations than the theoretical optimal functional values.

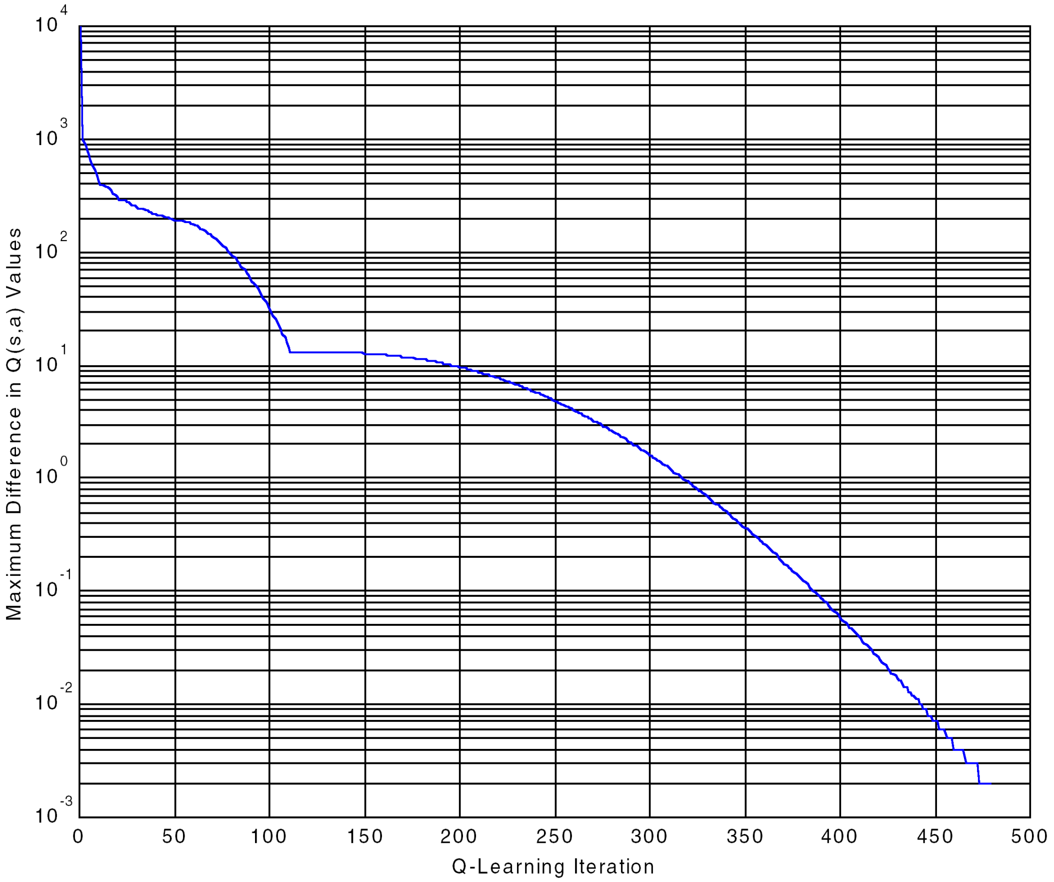


Figure 2: Maximum change in the estimated Q-values during each iteration, plotted on semilog paper.

Define the transformation

$$B(\beta, \mathbf{A}_A(i), k) = \{\beta[\mathbf{a}_1 - \mathbf{a}_{j^\star}] + \mathbf{a}_{j^\star}, \ldots, \beta[\mathbf{a}_A - \mathbf{a}_{j^\star}] + \mathbf{a}_{j^\star}\} \tag{21}$$

where $j^\star = \text{argmin}_j Q_k(i, \mathbf{a}_j)$ as the Successive Reduction procedure. Let $\mathbf{A}_A^0(i)$ be the initial set of reference actions. Each successive set of reference actions is defined by the transformation in (21) giving

$$\mathbf{A}_A^{l+1}(i) = B(\beta, \mathbf{A}_A^l(i), k). \tag{22}$$

This reduces the interval of uncertainty for each state i as follows:

Algorithm 3.2

```
1 Determine $\epsilon$-optimal policy for $\mathbf{A}_A^l(i)$ after $k$ iterations.
2 $\mathbf{A}_A^{l+1}(i) \leftarrow B(\beta, \mathbf{A}_A^l(i), k)$
```

If we denote the initial interval of uncertainty, IoU_0, as

$$IoU_0 = \sup_a\{a \in \mathbf{A}\} - \inf_a\{a \in \mathbf{A}\}, \tag{23}$$

then it requires at least

$$\frac{\ln \epsilon - \ln IoU_0}{\ln \beta} \tag{24}$$

successive reductions of the interval of uncertainty to reach any specified accuracy $\epsilon > 0$.

Theorem 3.2 *The Q-learning algorithm, when used to estimate the $Q(i, \mathbf{a}_j)$ values, generates two possibly distinct times k' and k with*

$$B(\beta, \mathbf{A}_A(i), k') = B(\beta, \mathbf{A}_A(i), k) \quad \forall i. \tag{25}$$

Proof: The Successive Reduction procedure is based on $j_{k'}^\star = \text{argmin}_j Q_{k'}(i, \mathbf{a}_j)$ and $j_k^\star = \text{argmin}_j Q_k(i, \mathbf{a}_j)$, respectively, $\forall i$. From Theorem 3.1, it follows that $j_{k'}^\star = j_k^\star \quad \forall i$. Therefore,

$$\beta[\mathbf{a}_1 - \mathbf{a}_{j_{k'}^\star}] + \mathbf{a}_{j_{k'}^\star} = \beta[\mathbf{a}_1 - \mathbf{a}_{j_k^\star}] + \mathbf{a}_{j_k^\star} \quad \forall i, \tag{26}$$

and (25) is proved. □

Theorem 3.2 allows for the application of the Successive Reduction procedure when the optimal policy is found rather than waiting for the optimal functional values to also converge.

Theorem 3.3 *If $\pi^\star$ represents the ϵ-optimal policy at time k for the system on the reference action subset $\mathbf{A}_A$, then $\pi^\star$ is an allowable policy on the new reference action subset*

$$\mathbf{A}_A'(i) = B(\beta, \mathbf{A}_A(i), k). \tag{27}$$

Proof: The optimal policy $\pi^\star$ is defined as $\pi^\star(i) = \mathbf{a}_{j^\star}$, where $j^\star = \text{argmin}_j Q_k(i, \mathbf{a}_j) \quad \forall i$. Therefore, under the successive reduction transformation defined in (21), the element corresponding to $\mathbf{a}_{j^\star}$ becomes

$$\beta[\mathbf{a}_{j^\star} - \mathbf{a}_{j^\star}] + \mathbf{a}_{j^\star} = \mathbf{a}_{j^\star} \tag{28}$$

and $\mathbf{a}_{j^\star} \in \mathbf{A}'_A$, for all i. □

Since the current optimal policy over the reference subset is an allowable policy in the revised reference subset, the following Corollary holds.

Corollary 3.1 *If the l-th error bound is defined as*

$$\varepsilon_V^l \equiv \max_i |V_n^l(i) - V(i)| \tag{29}$$

where

$$V_n^l(i) = \min_j Q_n(i, \mathbf{a}_j) \quad \forall \mathbf{a}_j \in \mathbf{A}_A^l(i), \tag{30}$$

then $\{\varepsilon_V^0, \varepsilon_V^1, \ldots\}$ is a non-increasing sequence.

Proof: Determine ε_V^0 from the Q-learning algorithm on the initial reference action subset $\mathbf{A}_A^0$. Apply the Successive Reduction procedure:

$$\mathbf{A}_A^1 = B(\beta, \mathbf{A}_A^0, n). \tag{31}$$

From Theorem 3.3, the optimal action at each state is an element of $\mathbf{A}_A^1$. Therefore,

$$V_n^{k+1}(i) = \min_j Q_n(i, \mathbf{a}_j) \leq \min_j Q_n(i, \mathbf{a}'_j) = V_{n'}^k(i) \quad \forall i \tag{32}$$

where $\mathbf{a}_j \in \mathbf{A}_A^0$ and $\mathbf{a}'_j \in \mathbf{A}_A^1$. Thus,

$$\varepsilon_V^1 \leq \varepsilon_V^0. \tag{33}$$

By induction, the result is proven. □

The error bounds are non-increasing, thus the sequence converges to some real number in $[0, \varepsilon_V^0]$. Corollary 3.1 provides that each application of the Successive Reduction procedure produces a policy that is at least as good as the previous policy.

We now examine conditions that ensure convergence to the optimal policy $\pi^\star$. Assume the state of the system is i, and the optimal policy over the continuous action space $\mathbf{A}$ is $\pi^\star = \{\mathbf{a}(0),$ ..., $\mathbf{a}(k)\}$, where $k+1$ is the iteration when the system enters the absorbing state $\mathbf{s}^\star$. If the transition function $\tau(\mathbf{s}, \mathbf{a}, \omega)$ is not random, then the state space $\mathbf{S}$ can be divided into subsets

$$\mathbf{S} = \mathbf{s}^\star \cup \mathbf{S}^1 \cup \cdots \cup \mathbf{S}^N \tag{34}$$

where $\mathbf{S}^j$ is the set of all states that are j transitions away from the set-point $\mathbf{s}^\star$ under the optimal policy $\pi^\star$. Under Assumption 2.2, $N < \infty$.

Initially, we look at the one-stage case where $k = 0$, or, equivalently, some state $i \in \mathbf{S}^1$. The Successive Reduction procedure guarantees policy improvement when the piecewise-linear approximation defined by the $Q(i, \mathbf{a}_j)$ values for state i is strictly quasi-convex in j. See Figure 3. This ensures that if action $a^\star$ is optimal, then actions closer to that optimal action are better than actions further away.

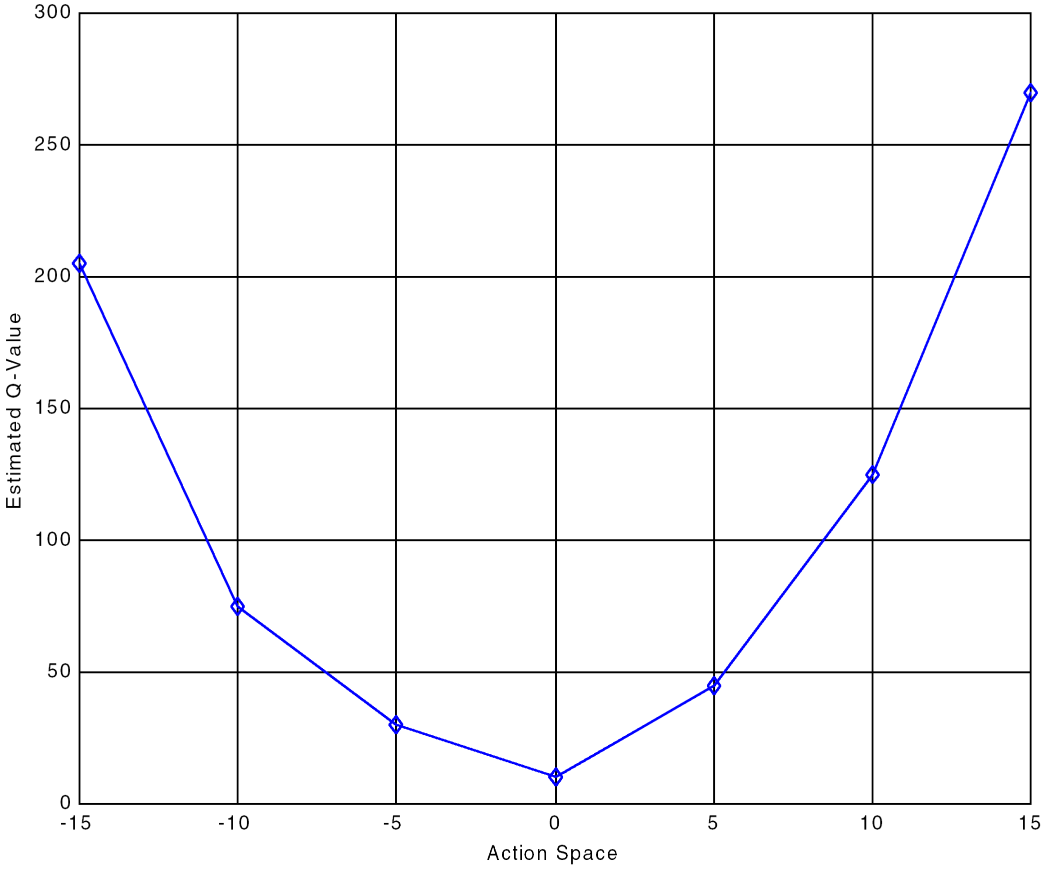


Figure 3: Piecewise-linear approximation of $Q(i, a)$ using reference subset $\mathbf{A}_7 = \{-15, -10, -5, 0, 5, 10, 15\}$.

Theorem 3.4 *If the $Q(i, \mathbf{a}_j)$ function is strictly quasi-convex in j for state $i \in \mathbf{S}^1$ using reference subset $\mathbf{A}_A$, then for*

$$(\sup_a\{a \in \mathbf{A}\} - \inf_a\{a \in \mathbf{A}\})(A-1)^{-1} < \beta < 1$$

the reference subset

$$\mathbf{A}'_A(i) = B(\beta, \mathbf{A}_A(i), k) \quad \forall i \tag{35}$$

is such that

$$\min_j Q(i, \mathbf{a}'_j) \leq \min_j Q(i, \mathbf{a}_j). \tag{36}$$

Further, for any $\epsilon > 0$ there exists an $N \in \mathbb{N}$ such that N applications of the Successive Reduction procedure satisfies

$$\max_j \|\mathbf{a}_j - \pi^\star(i)\| < \epsilon \tag{37}$$

Proof: Upon convergence of the Q-values, the optimal action in $\mathbf{A}_A$ for state i is

$$j^\star_k = \text{argmin}_j Q_k(i, \mathbf{a}_j). \tag{38}$$

Therefore,

$$\beta[\mathbf{a}_{j^\star_k} - \mathbf{a}_{j^\star_k}] + \mathbf{a}_{j^\star_k} = \mathbf{a}_{j^\star_k}, \tag{39}$$

and $\mathbf{a}_{j^\star_k} \in \mathbf{A}'_A$. Thus,

$$\min_j Q(i, \mathbf{a}'_j) \leq Q(i, \mathbf{a}'_{j^\star_k}) = \min_j Q(i, \mathbf{a}_j). \tag{40}$$

Therefore, (36) is satisfied. As a result of the strict quasi-convexity, the optimal action

$$\mathbf{a}_{j^\star_k - 1} < a^\star < \mathbf{a}_{j^\star_k + 1}. \tag{41}$$

The choice of $\beta > (\sup_a\{a \in \mathbf{A}\} - \inf_a\{a \in \mathbf{A}\})(A-1)^{-1}$ ensures that both

$$\mathbf{a}'_1 \leq \mathbf{a}_{j^\star_k - 1}$$

and

$$\mathbf{a}_{j^\star_k - 1} \leq \mathbf{a}'_A.$$

Therefore,

$$\mathbf{a}'_1 < a^\star < \mathbf{a}'_A. \tag{42}$$

□

The strict quasi-convexity assumption does not always hold, but even then an adroit choice of β and A can achieve the desired result in practice. This is evident in the example applications that follow.

Theorem 3.4 ensures convergence to the optimal action for the one-stage problem. By induction, the CAS algorithm converges to the optimal policy $\pi^\star$ under the strictly quasi-convex assumption if the successive reductions are applied to a particular sequence of states.

Theorem 3.5 *The Successive Reduction procedure determines the ϵ-optimal policy $\pi^\star$ for all states, for a system with the $Q(i, \mathbf{a}_j)$ function strictly quasi-convex in j for all i.*

Proof: Consider all states $i \in \mathbf{S}^1$. By Theorem 3.4, there exists a K such that after K successive reductions on each state i,

$$\max_j \|\mathbf{a}_j - \pi^\star(i)\| < \epsilon. \tag{43}$$

Assume this is true for subsequent applications to sets $\mathbf{S}^2, \ldots, \mathbf{S}^{N-1}$. Therefore, by induction and Bellman's principle of optimality, for all states $i \in \mathbf{S}^N$, the problem is equivalent to the one-stage case with a terminal cost of $\min_{j'} Q(i', \mathbf{a}'_j)$ where $i' \in \mathbf{S}^{N-1}$ is the successor state to the action j taken in state i. Thus, the optimal action is found for all states i. □

Theoretical convergence is guaranteed by the appropriate choice of which states to apply the Successive Reduction procedure to first. The constructed sequence, $\mathbf{S}^1, \ldots, \mathbf{S}^N$, accomplishes this but, under Assumption 2.4 this classification of states cannot be made explicitly. As such, heuristics are employed that attempt to approximate these subsets of $\mathbf{S}$.

3.3 Heuristic Stopping Criterion

Recall from Figure 1 that the potential for significant computational savings exists. In practice, to ensure that the ϵ-optimal policy $\pi^\star$ is found, the ϵ-optimal Q-values must also be determined. This negates any computational savings. This section illustrates that, in practice, near-optimal policies π^* may be found in significantly less computational effort. A stopping criterion is used to halt the Q-learning algorithm. The stopping criterion investigated is the *$Z\%$ K-stationary stopping criterion.*

Definition 3.1 *The $Z\%$ K-*stationary stopping criterion *halts the Q-learning algorithm at iteration n if the current policy*

$$\pi^*(i) \equiv \min_j Q_n(i, \mathbf{a}_j), \ \ \mathbf{a}_j \in \mathbf{A}_A(i) \quad \forall i \tag{44}$$

has remained constant for the last K visits to each state i for at least $Z\%$ of the reference state subset.

This prevents states that are infrequently visited during the learning phase from preventing the CAS algorithm to continue with its policy improvement steps.

3.4 Order of Successive Reduction Procedure

The theoretical convergence properties of the CAS algorithm rest upon the assumption that the Successive Reduction procedure is performed on the proper sequence of states. In Theorem 3.5, the constructed sequence, $\mathbf{S}^1, \ldots, \mathbf{S}^N$, accomplishes this but, under Assumption 2.4, this classification of states cannot be made explicitly. We propose a heuristic based on temporal differences policy evaluation that approximates these subsets of $\mathbf{S}$.

Determining the number of expected transitions from any state i to the set-point $\mathbf{s}^\star$ is, itself, an approximate dynamic programming problem. The current ϵ-optimal policy π^* on reference action subset $\mathbf{A}_A(i)$ was determined via Q-learning using a uniform probability exploration strategy. A proper sequence of updates must be determined or the CAS algorithm converges to a suboptimal policy.

The learned Q-values determine the optimal action but these values do not necessarily correlate with a measure of the number of expected transitions until the set-point is reached. The online algorithm is used for approximating these Q-values may be extended to estimating this new value function:

$$\begin{aligned} V_{\pi^*}(i) \quad = \quad & \text{Expected number of transitions under policy } \pi^* \text{ from state } i \\ & \text{to the set-point } \mathbf{s}^\star. \end{aligned} \tag{45}$$

The functional equation is

$$V_{\pi^*}(i) = 1 + \sum_{j=1}^{S} p_{ij}(\pi^*(i))V_{\pi^*}(j), \tag{46}$$

with boundary condition $V_{\pi^*}(\mathbf{s}^\star) = 0$. This can easily be determined via approximate dynamic programming with the update equation:

$$V_{\pi^*}^{n+1}(i) = \alpha V_{\pi^*}^{n}(i) + (1-\alpha)(1 + V_{\pi^*}^{n}(j)). \tag{47}$$

The CAS algorithm now becomes a sequence of optimal policy determinations and successive reductions of the interval of uncertainty, IoU. The flowchart is given in Figure 4.

3.5 Computational Complexity

Both Q-learning and the CAS algorithm possess the same complexity in terms of data storage ($O(SA)$) and computations, but the potential for savings comes through the total number of iterations for each. The cyclical policy determination/policy improvement procedure of the CAS algorithm terminates in a finite number of cycles. Let the desired number of visits for each state-action pair be D. In full backups, D iterations accomplishes this goal. In sample backups, if each state is equally probable and each action is chosen according to a uniform distribution, DSA iterations are required for each state-action pair to have D expected visits. If an accuracy of ϵ is desired, standard Q-learning with sample backups requires

$$A > \frac{(\sup_a\{a \in \mathbf{A}\} - \inf_a\{a \in \mathbf{A}\})}{2\epsilon} = \frac{IoU_0}{2\epsilon} \tag{48}$$

reference actions. However, the CAS algorithm requires fewer reference actions for the same ϵ since the interval of uncertainty is reduced during each policy determination/policy improvement cycle. Under the strict quasi-convexity assumption,

$$\beta = \frac{2}{A-1}, \tag{49}$$

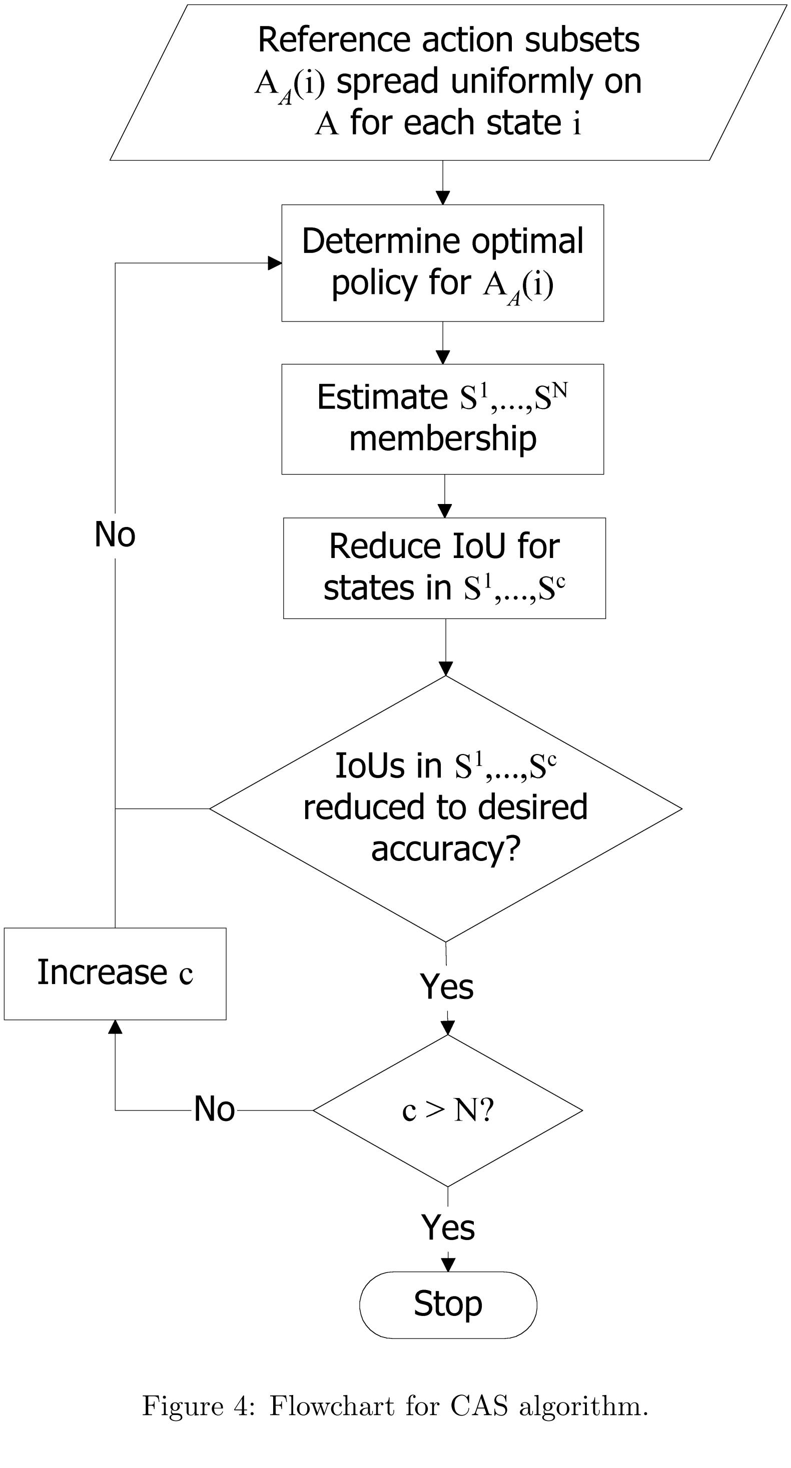

Reference action subsets $A_A(i)$ spread uniformly on A for each state i
Determine optimal policy for $A_A(i)$
Estimate $S^1,\dots,S^N$ membership
Reduce IoU for states in $S^1,\dots,S^c$
IoUs in $S^1,\dots,S^c$ reduced to desired accuracy?
No
Yes
Increase c
c > N?
No
Yes
Stop

Figure 4: Flowchart for CAS algorithm.

insures that the IoU is reduced to the interval $[\mathbf{a}_{j^\star-1}, \mathbf{a}_{j^\star+1}]$ after the Successive Reduction procedure is applied. Therefore, if a limit of M successive reductions for each state is desired,

$$A > \exp\left[\frac{M \ln 2 + \ln(\sup_a\{a \in \mathbf{A}\} - \inf_a\{a \in \mathbf{A}\}) - \ln \epsilon}{M}\right] + 1 \tag{50}$$

reference actions achieves ϵ-accuracy. For example, if $\epsilon = 0.001$, then for standard Q-learning with sample backups for the inverted pendulum requires 15,000 reference actions. With the CAS algorithm, however, the same accuracy can be achieved with 347 actions and 2 successive reductions at each state. Thus, accuracy is linear in A for standard Q-learning, but exponential in A for the CAS algorithm.

In the worst case, only 1 state belongs to each $\mathbf{S}^1, \ldots, \mathbf{S}^N$ and, therefore, $(MN)DSA = MDS^2A$ iterations are required. While this is a considerable amount, in practice $N \ll S$. For the average-case in the example problems, the combination of the heuristic stopping criterion coupled with the large reduction in the number of reference actions required generates a near-optimal control policy in 20-40% fewer iterations than standard Q-learning.

4 Example Applications

The CAS algorithm and the $Z\%$ K-stationary stopping criterion are investigated on two set-point regulation problems. The first is the benchmark nonlinear control problem of balancing an inverted pendulum on a cart [13]. This common problem illustrates some of the advantages of the proposed algorithm. The second problem of interest is the more complex task of stabilizing a power system under a load [10, 11, 5, 6, 16]. This exhibits the application of the proposed reinforcement learning algorithm on a practical problem.

4.1 Inverted Pendulum Balancing

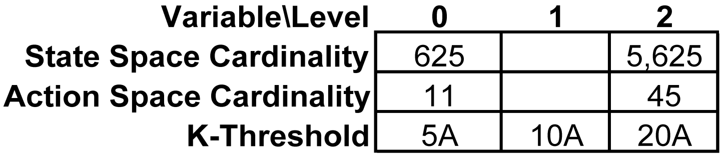
The inverted pendulum balancing problem is an example of an inherently unstable system [13] and has a broad base for comparison throughout the literature. There are four state variables, $\mathbf{s} = \langle \theta, \Delta\theta, x, \Delta x \rangle$, and one action variable, $\mathbf{a} = F$. In this experiment, the immediate return function for taking action $\mathbf{a}$ while in state $\mathbf{s}$ is defined

$$R(\mathbf{s}, \mathbf{a}) \equiv \left[\theta^2 + x^2\right] \Delta t \tag{51}$$

where Δt represents the time interval between samplings of the state vector $\mathbf{s}$. Therefore,

$$\begin{aligned} V(i) \;=\;& \text{Expected discounted sum of squared error (SSE) when} \\ & \text{starting in state } i \text{ and following an optimal policy } \pi^\star \text{ thereafter.} \end{aligned} \tag{52}$$

Initially, we examine the effectiveness of our stopping criterion. Specifically, we investigate the percentage of states meeting a particular K-stationary criterion after a fixed number of iterations. Three factors are used–the state space cardinality S, the action space cardinality A, and the particular K-stationary level. When using sample backups, we are not assured that each state-action pair is tried during the learning phase. Therefore, the K-stationary criterion is made dependent upon the number of actions A. For any level, xA, this implies an expected number of visits to each reference action at a particular state is x. Table 1 shows the various levels for the three factors in the experiment. Figure 5 plots the main effects of these factors on the percentage of states meeting the K-stationary stopping criterion after 1,000,000 iterations.



Variable\Level	0	1	2
State Space Cardinality	625		5,625
Action Space Cardinality	11		45
K-Threshold	5A	10A	20A

Table 1: Levels of variables in K-stationary sample backup inverted pendulum balancing experiment.

Four replications for each factor level combination were run. The mean percentage of states meeting the K-stationary stopping criterion is $28.5\% \pm 0.114\%$. The standard error is estimated using a pooled estimate from the replicated runs:

$$s^2 = \frac{\nu_1 s_1^2 + \cdots + \nu_8 s_8^2}{\nu_1 + \cdots + \nu_8} = 0.004\% \tag{53}$$

with 24 degrees of freedom. From the percentage of states that have a stationary policy for K iterations, the following significant inferences can be drawn:

1. All three factors seem to have a significant negative effect on the percentage of states that have a stationary policy for K iterations. The variable K has an effect for obvious reasons. The negative effects of S and A are due to the fixed stopping iteration. Naturally, with more state-action pairs, a fixed stopping point implies fewer expected visits to each pair.

2. The $S \times A \times K$ interaction was significant and, therefore, the separate effects of S and A are difficult to interpret.

3. Failure to balance the pendulum occurred for 3 of the 48 replicate runs. All three of these were at the $K = 5A$ level. The next experiment uses only the $K = 10A$ level for this reason.

We now examine the computational savings based on the iteration that particular $Z\%$ K-stationary stopping criterion are met. The state space is kept at a constant size of 625. The two factors are A and $Z\%$. The levels for these two factors are given in Table 2.

The CAS algorithm cycles through a number of policy determination and policy improvement steps until a sufficient approximation of the optimal continuous policy is found. The main

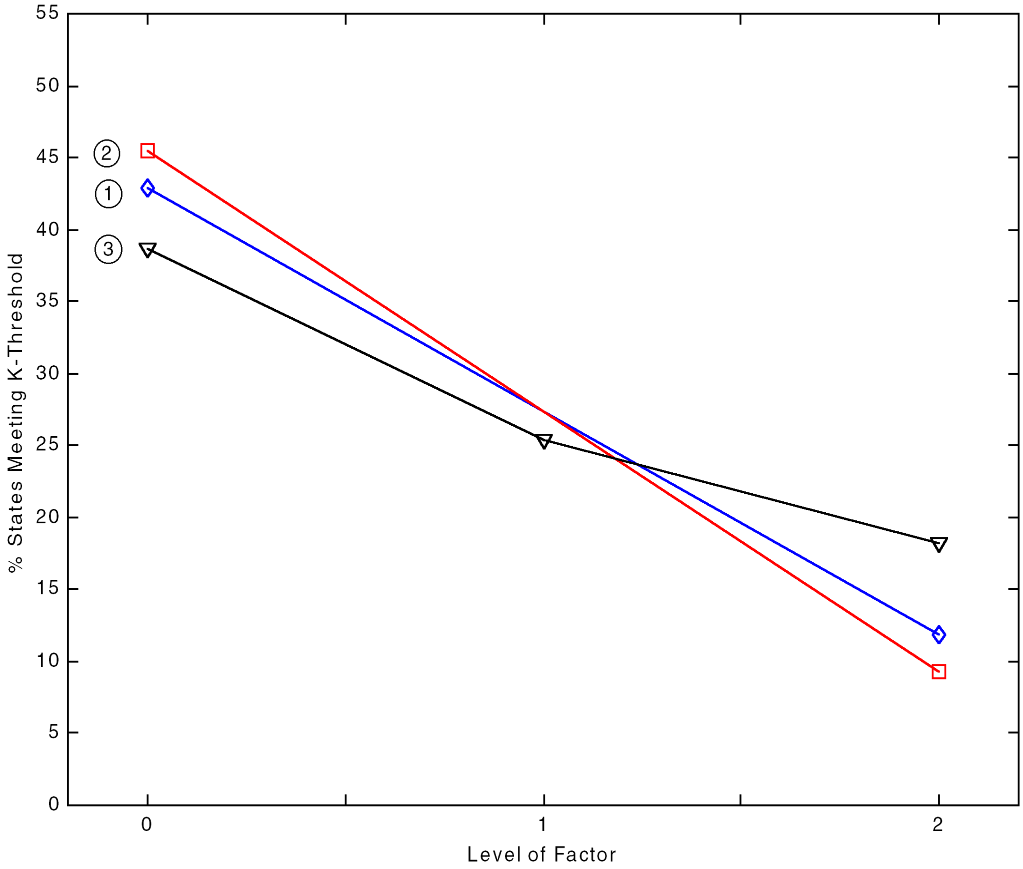


Figure 5: Main effects of factors on the percentage of states meeting K-stationary stopping criterion: (1) State space cardinality, (2) Action space cardinality, (3) K.

Variable\Level	**0**	**1**	**2**
Action Space Cardinality	7	11	45
Z%	25%	50%	75%

Table 2: Levels of variables in $Z\%$ K-stationary sample backup inverted pendulum balancing experiment.

effects on computational savings from a particular $Z\%$ K-stationary stopping rule is shown in Figure 6. With a small number of reference actions, A, the computational savings can be quite large. Specifically, for this problem the average computational savings of the CAS algorithm over standard Q-learning is 82.1%, 71.4%, and 49.6% respectively for $A = 7$, $A = 11$, and $A = 45$. Furthermore, there was no statistically significant difference in the optimal value function between the CAS algorithm and Q-learning for various initial state vectors.

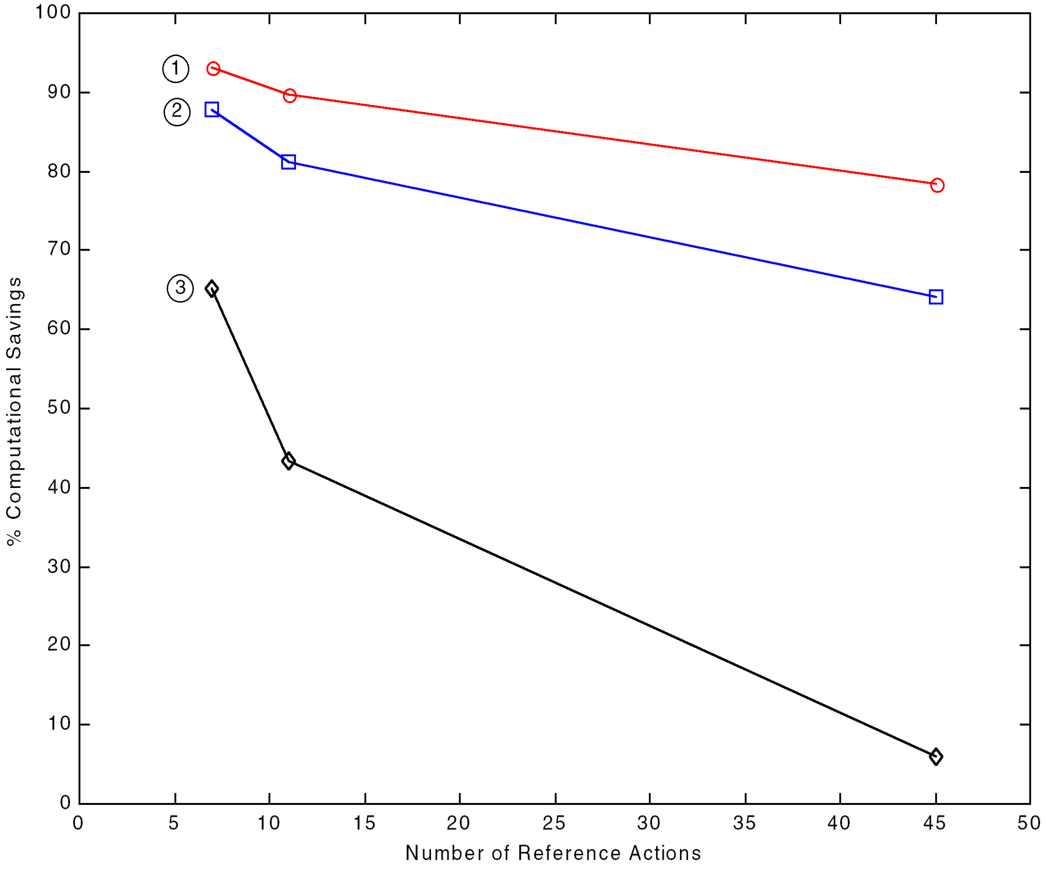


Figure 6: Main effects of factors on the number of iterations required before meeting various $Z\%$ K-stationary stopping criterion: (1) $Z\%$ set to 25%, (2) $Z\%$ set to 50%, (3) $Z\%$ set to 75%.

For the inverted pendulum balancing problem, Figure 7 plots the trajectory of the learned optimal control for a particular initial point compared with a benchmark optimal control. The first trajectory is the benchmark optimal trajectory assuming the model is known. The second is a benchmark trajectory for an unknown model using sample backups. The third trajectory is the learned trajectory for a discrete action space approximation of $A = 7$. Finally, the fourth trajectory is the result of the CAS algorithm using $A = 7$. Corollary 3.1 ensures that the result of the CAS algorithm can be no worse than this third trajectory. The figure highlights the advantage of searching a continuous action space, as the CAS algorithm does, over searching over a fixed discrete subset of the action space like standard Q-learning. In this case, the optimal control is *not* part of this discrete subset and the learned control law for Q-learning, while optimal over the discrete subset of reference actions, is sub-optimal for the continuous space.

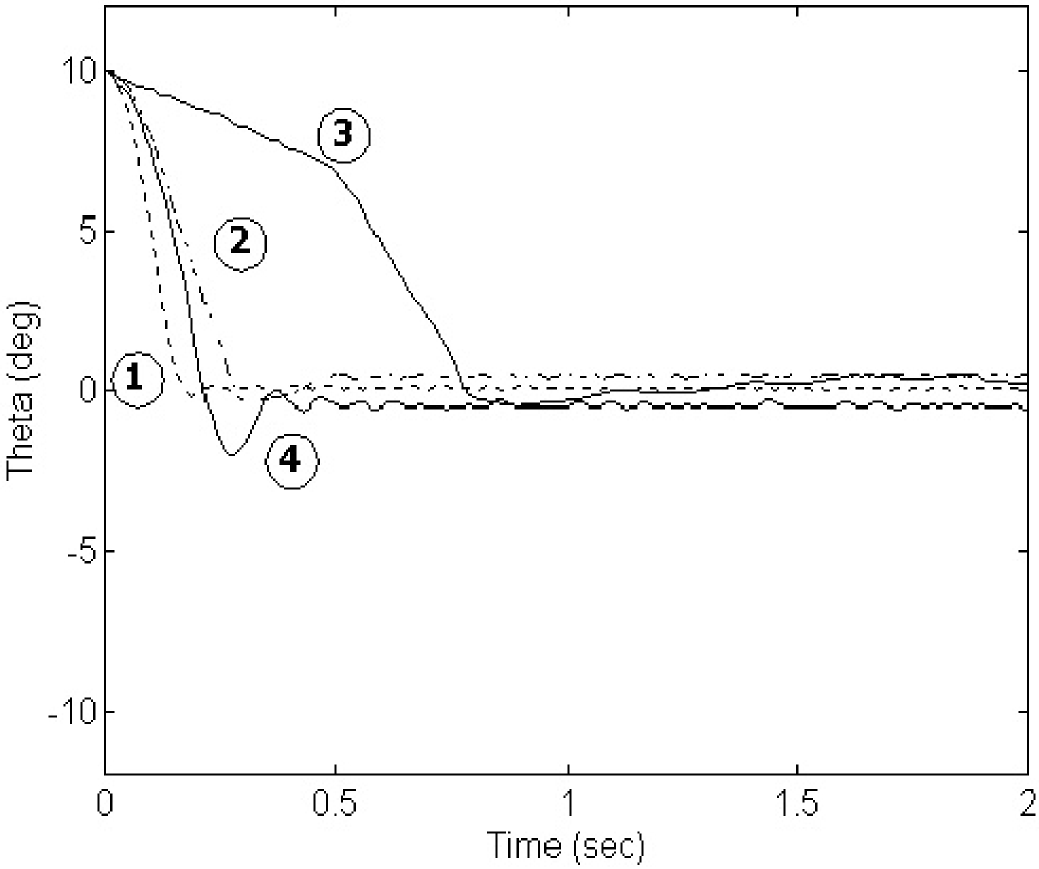


Figure 7: Learned control trajectories for various algorithms: (1) Benchmark with full backup, (2) Benchmark with sample backup, (3) Q-learning, (4) CAS algorithm.

4.2 Power System Stabilization

The power system stabilization problem represents an underdetermined system that can be approached as either a multiple-input/single-output (MISO) or multiple-input/multiple-output (MIMO) control problem [10]. For illustrative purposes, the MISO approach is taken here. The system considered is composed of a synchronous machine with an exciter and a stabilizer connected to an infinite bus. The dynamics of the synchronous machine can be expressed using the linearized incremental model with two measurable inputs, $\mathbf{s} = [\omega, \Delta\omega]$, and one output, $\mathbf{a} = [u]$ [12]. In contrast to the inverted pendulum system where the dynamics are assumed unknown but the complete state of the system can be measured, the power system stabilization problem has only two state variables, $[\omega, \Delta\omega]$, that can readily be measured. Therefore, the unknown values for the remaining state variables create a stochastic process with unknown transition probabilities which the algorithms will implicitly learn.

Due to the additional uncertainties in the dynamics of the system, more learning iterations are needed in both Q-learning and the CAS algorithm. Not all experiments ended in success, though the majority did. Four replications of each experiment were run. For each, the desired accuracy, ϵ, is set to 0.001. For standard Q-learning, this implied that

$$A > \frac{+0.12 - (-0.12)}{2 \cdot 0.001} = 120. \tag{54}$$

For the CAS algorithm, $A = 11$ was chosen. Under the strict quasi-convexity assumption, $\beta = 0.2$ and three iterations of the Successive Reduction procedure will obtain the ϵ-accuracy. The number of discrete states was set to 625 for both algorithms.

Our experiments on this practical problem of interest highlight some of the advantages and disadvantages of using the CAS algorithm on an underdetermined system. Of the twenty random experiments run for both Q-learning and the CAS algorithms, both learned to keep the system from failing in 19 of 20 trials. By varying the number of learning iterations to insure that convergence to the optimal value function is obtained in Q-learning, it was determined that in some instances this system cannot be controlled simply by measuring ω and $\Delta\omega$ and varying u. In the 19 trials where success was obtained, there were varying degrees of “goodness” of the learned control law. For the Q-learning algorithm, the range of settling times varied from 4.21 seconds to 11.38 seconds. Similarly, the range of settling times for the CAS algorithm varied from 4.53 seconds to 9.76 seconds. There was no statistically significant difference in the paired settling times for the same random trial for each algorithm. As an example of a learned control trajectory, see Figure 8. The CAS controller successfully learned to stabilize the plant in approximately 4.5 seconds after a load is applied.

The advantage of using the CAS algorithm over standard Q-learning is in the computational savings. For the power system stabilization problem, the average computational savings is 23.4% for the 20 trials. This is due to the ability of the CAS algorithm to concentrate its learning effort on a particular region of the action space by refining the interval of uncertainty (IoU) through the Successive Reduction procedure.

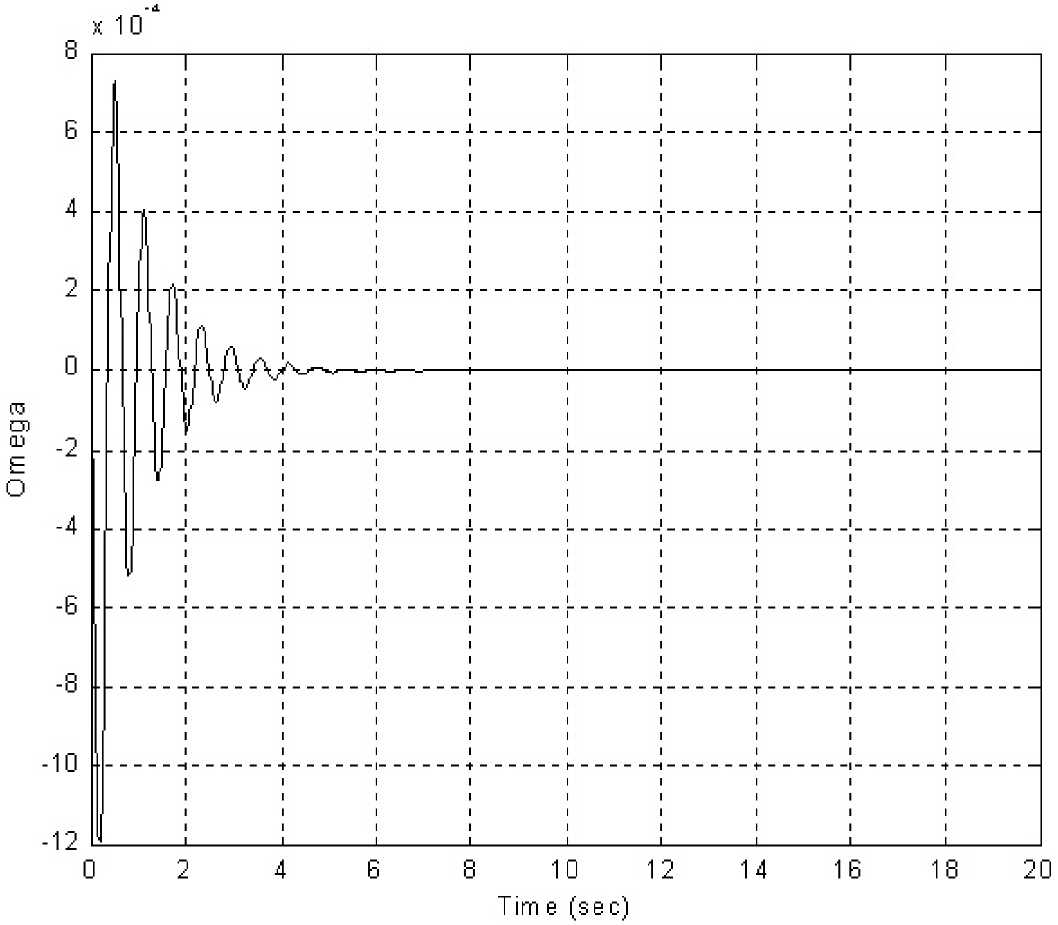


Figure 8: Learned control trajectory for the power system stabilization problem using the CAS algorithm.

5 Summary

This paper proposes a reinforcement learning algorithm for set-point regulation problems that have continuous action spaces. It is based on Watkin's Q-learning algorithm and derivative-free line search methods in optimization. The computational savings over traditional Q-learning have been illustrated. The CAS algorithm has also been shown to efficiently learn an ϵ-optimal control law for two example problems. Its theoretical convergence properties have been established under moderate assumptions, though deviation from these assumptions in practice is not necessarily detrimental to learning a good control law.

We are currently investigating methods to generalize these learned control laws to continuous state spaces through procedures based on fuzzy set theory. Application to more diverse problems of interest is also underway, especially those problems that require the use of a hierarchical reinforcement learning approach.

6 Acknowledgments

This research is sponsored in part by the National Science Foundation under Grant ECS-9216004, the Electric Power Research Institute under Grant RP 8030-16, and the National Aeronautics and Space Administration under Grant NAG 9-726.

References

[1] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

[2] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[3] H.R. Berenji and P. Khedkar. Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on Neural Networks*, 3(5):724–740, 1992.

[4] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[5] S.-J. Cheng, Y.S. Chow, O.P Malik, and G.S. Hope. An adaptive synchronous machine stabilizer. *IEEE Transactions on Power Systems*, PWRS-1(3):101–107, August 1986.

[6] S.-J. Cheng, O.P Malik, and G.S. Hope. Self-tuning stabilizer for a multimachine power system. In *IEE Proceedings*, volume 133-C, pages 176–185, May 1986.

[7] S.E. Dreyfus and A.M. Law. *The Art and Theory of Dynamic Programming.* Academic Press, New York, 1977.

[8] A.O. Esogbue and W.E. Hearnes. Approximate policy improvement for continuous action set-point regulation problems. In *Proceedings of the 1998 World Congress on Compuational Intelligence*, pages 1692–1697, Anchorage, AK, May 4-9 1998.

[9] A.O. Esogbue and J.A. Murrell. A fuzzy adaptive controller using reinforcement learning neural networks. In *Proceedings of Second IEEE International Conference on Fuzzy Systems*, pages 178–183, March 28–April 1 1993.

[10] A.O. Esogbue, Q. Song, and W.E. Hearnes. Application of a self-learning fuzzy-neuro controller to the power system stabilization problem. In *Proceedings of the 1995 World Congress on Neural Networks*, volume II, pages 699–702, Washington, DC, July 17-21 1995.

[11] A.O. Esogbue, Q. Song, and W.E. Hearnes. Defuzzification filters and applications to power system stabilization problems. In *Proceedings of the 1996 IEEE International Conference on Systems, Man and Cybernetics Information, Intelligence and Systems*, Beijing, China, October 14 - 17 1996.

[12] Y.-Y. Hsu and C.-Y. Hsu. Design of a proportional-integral power system stabilizer. *IEEE Transactions on Power Systems*, PWRS-1(2):46–53, May 1986.

[13] Thomas III MIller, Richard S. Sutton, and Paul J. Werbos, editors. *Neural Networks for Control.* MIT Press, Cambridge, MA, 1990.

[14] James A. Murrell. *A Statistical Fuzzy Associative Learning Approach To Intelligent Control.* PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, December 1993.

[15] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, San Diego, CA, 1983.

[16] J. Shi, L.H. Herron, and A. Kalam. A fuzzy logic controller applied to power system stabilizer for a synchronous machine power system. In *Proceedings of IEEE Region 10 Conference, Tencon 92*, Melbourne, Australia, November 11-13 1992.

[17] R.S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[18] C.J.C.H. Watkins. *Learning from delayed rewards.* Ph.d. thesis, Cambridge University, Cambridge, England, 1989.

[19] C.J.C.H. Watkins and P. Dayan. *q*-learning. *Machine Learning*, 8:279–292, 1992.

Learning by PUNISH/REWARD and Learning by Supervision

Bernard Widrow
(widrow@stanford.edu)

Abstract

I will discuss two topics. One is about the efficiency of adaptive algorithms and explains the popularity of the LMS algorithm and Backprop. The other is about learning by PUNISH/REWARD in multilayer networks. I will also discuss the relationship/difference between the two.

Direct Neural Dynamic Programming, a Design-Centric Approach vs. Prior Results

Jennie Si
(si@asu.edu)

Abstract

For the ease of discussion in the talk, I will use the terms “discrete event” approaches and “continuous state” approaches to discuss solutions of approximate dynamic programming. The former refers to the fact that controls/actions are obtained by search algorithms and the problems are discrete event in nature. The latter refers to that (approximate) gradient information is used in value function approximation and action generation, and the problems can be in continuous state spaces. My talk centers around two issues. First I will show that the two approaches are closely related, especially in value function approximation. They are complimentary to one another when it comes to solve large, hybrid systems problems. I will then show that our direct NDP design shows promise as a robust algorithm (measured by learning statistics, problem scalability, the range of problems handled) to the approximate dynamic programming problems. Details of the learning system design, the algorithm, and possibilities of further qualitative analysis using stochastic approximation, will be presented. I will also show a comprehensive case study of this design in stabilizing, tracking, and re-configuring an Apache helicopter using a full-scale Boeing model. This is probably the first time that approximate dynamic programming is systematically applied to and evaluated on a complex, continuous state, MIMO, nonlinear system with uncertainty.

On-Line Learning Control by Association and Reinforcement

Jennie Si, *Senior Member, IEEE*, and Yu-Tsung Wang, *Member, IEEE*

***Abstract*—This paper focuses on a systematic treatment for developing a generic on-line learning control system based on the fundamental principle of reinforcement learning or more specifically neural dynamic programming. This on-line learning system improves its performance over time in two aspects. First, it learns from its own mistakes through the reinforcement signal from the external environment and tries to reinforce its action to improve future performance. Second, system states associated with the positive reinforcement is memorized through a network learning process where in the future, similar states will be more positively associated with a control action leading to a positive reinforcement. A successful candidate of on-line learning control design will be introduced. Real-time learning algorithms will be derived for individual components in the learning system. Some analytical insight will be provided to give guidelines on the learning process took place in each module of the on-line learning control system. The performance of the on-line learning controller is measured by its learning speed, success rate of learning, and the degree to meet the learning control objective. The overall learning control system performance will be tested on a single cart-pole balancing problem, a pendulum swing up and balancing task, and a more complex problem of balancing a triple-link inverted pendulum.**

***Index Terms*—Neural dynamic programming (NDP), on-line learning, reinforcement learning.**

I. INTRODUCTION

WE ARE considering a class of learning decision and control problems in terms of optimizing a performance measure over time with the following constraints. First, a model of the environment or the system that interacts with the learner is not available *a priori*. The environment/system can be stochastic, nonlinear, and subject to change. Second, learning takes place “on-the-fly” while interacting with the environment. Third, even though measurements from the environment are available from one decision and control step to the next, a final outcome of the learning process from a generated sequence of decisions and controls may come as a delayed signal in only an indicative “win” or “loose” format.

Dynamic programming has been applied in different fields of engineering, operations research, economics, and so on for many years [2], [5], [6], [22]. It provides truly optimal solutions to nonlinear stochastic dynamic systems. However, it is well understood that for many important problems the computation costs of dynamic programming are very high, as a result of the “curse of dimensionality” [8]. Other complications in applications include a user supplied explicit performance measure and a stochastic model of the system [2]. Incremental optimization methods come in handy to approximate the optimal cost and control policies [3], [11].

Reinforcement learning has held great intuitive appeal and has attracted considerable attention in the past. But only recently it has made major advancements by implementing the temporal difference (TD) learning method [1], [16], [21]. The most noteworthy result is a TD-Gammon program that has learned to play Backgammon at a grandmaster level [17]–[19]. Interestingly enough, the development history of Gammon programs also reflects the potentials and limitations of various neural networks learning paradigms. With the success of TD-Gammon, the TD algorithm is no doubt a powerful learning method in Markovian environments such as game playing.

How does one ensure successful learning in a more generic environment? Heuristic dynamic programming (HDP) was proposed in the 1970s [22] and the ideas were firmed up in the early 1990s [23]–[25] under the names of adaptive critic designs. The original proposition for HDP was essentially the same as the formulation of reinforcement learning (RL) using TD methods. Specifically a critic network “critiques” the generated action value in order to optimize a future “reward-to-go” by propagating a temporal difference between two consecutive estimates from the critic/prediction network. This formulation falls exactly into the Bellman equation. Even with the same intention at the beginning, the two approaches started to differentiate by the way the actions were generated. HDP and the adaptive critics in general train a network to associate input states with action values. On the other hand, TD-based Gammon programs, as well as Q-learning, opted for search algorithms to determine the optimal moves and, hence, avoid additional error during the initial action network training, with a price paid for search speed.

Existing adaptive critic designs [26] can be categorized as: 1) HDP; 2) dual heuristic dynamic programming (DHP); and 3) globalized dual heuristic dynamic programming (GDHP). Variations from these three basic design paradigms are also available, such as action dependent (AD) versions of the above architectures. AD refers to the fact that the action value is an additional input to the critic network. Action dependent variants from the original three paradigms will be denoted with an abbreviation of “AD” in front of their specific architecture. For example, ADHDP and ADDHP denote “action dependent heuristic dynamic programming” and “action dependent dual heuristic dynamic programming,” respectively. Our proposed

Manuscript received October 7, 1999; revised March 20, 2000 and November 20, 2000. This work was supported by NSF under Grants ECS-9553202 and ECS-0002098 and in part by EPRI-DOD under Grant WO8333-01, by DARPA under Grant MDA 972-00-1-0027, and by Motorola.

The authors are with Department of Electrical Engineering, Arizona State University, Tempe, AZ 85287-7606 USA (e-mail: si@asu.edu).

Publisher Item Identifier S 1045-9227(01)01404-7.

on-line learning system is most relevant to ADHDP. Once again, the basic idea in adaptive critic design is to adapt the weights of the critic network to make the approximating function, J, satisfy the modified Bellman equation. In this framework, instead of finding the exact minimum, an approximate solution is provided for solving the following equation:

$$J^*(X(t)) = \min_{u(t)}\{J^*(X(t+1)) + g(X(t), X(t+1)) - U_0\} \tag{1}$$

where $g(X(t), X(t+1))$ is the immediate cost incurred by $u(t)$ at time t, and U_0 is a heuristic term used to balance [24]. To adapt $J(X(t))$ in the critic network, the target on the right-hand side of (1) must be known *a priori*. To do so, one may wait for a time step until the next input becomes available. Consequently, $J(X(t+1))$ can be calculated by using the critic network at time $t+1$. Another approach is to use a *model network*, which is a pretrained network to approximate the system dynamics. In principle, such a network can be trained on-line.

One major difference between HDP and DHP is within the objective of the critic network. In HDP, the critic network outputs J directly, while DHP estimates the derivative of J with respect to its input vector. Since DHP builds derivative terms over time directly, it reduces the probability of error introduced by backpropagation. GDHP is a combination of DHP and HDP, approximating both $J(X(t))$ and $\partial J(X(t))/\partial X(t)$ simultaneously with the critic network. Therefore, the performance of GDHP is expected to be superior to both DHP and HDP. However, the complexities of computation and implementation are high for GDHP. The second derivative terms, $\partial^2 J(X(t))/\partial X(t)\partial w_c(t)$, need to be calculated at every time step. Analysis and simulation results in [12] and [13] are consistent with this observation.

Adaptive critic designs such as HDP, DHP, and GDHP, as well as their action dependent versions have been applied to an autolanding problem [12]. In implementations, the critic networks of HDP and ADHDP are used to approximate J. To obtain the value of J at time $t+1$, the states or/and actions are predicted by using a model network. The model network approximates plant dynamics for a given state $X(t)$ and action $u(t)$, and the model network outputs $X(t+1)$. In [12], the model network was trained off-line. Results from [12] show that GDHP and DHP are better designs than the HDP and ADHDP for the autolanding problem. The auto-landers trained with wind shear for GDHP and DHP successfully landed 73% of all 600 trials while those for HDP and ADHDP were below 50%.

From the previous discussions, we can also categorize adaptive critic designs by whether or not a model was used in the learner, as shown in [26]. Note that in adaptive critic designs, there are two partial derivative terms in the backpropagation path from the Bellman equation. They are $\partial J(t)/\partial W_c(t)$ and $\partial J(t+1)/\partial W_c(t)$. When adaptive critic designs were implemented without a model network (i.e., two-network design), the second partial derivative term was simply ignored. The price paid for omitting this term can be high. Results in [12] and [13], seem to agree with this observation. In later implementations such as DHP and GDHP, a model network was employed to take into account the $\partial J(t+1)/\partial W_c(t)$ term.

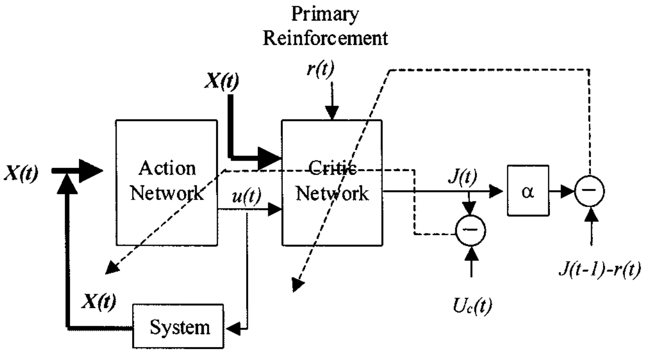


Fig. 1. Schematic diagram for implementations of neural dynamic programming. The solid lines represent signal flow, while the dashed lines are the paths for parameter tuning.

Our proposed approach in this paper is closely related to ADHDP. One major difference is that we do not use a system model to predict the future system state value and consequently the cost-to-go for the next time step. Rather, we store the previous J value. Together with the current J value, we can obtain the temporal difference used in training. We have thus resolved the dilemma of either ignoring the $\partial J(t+1)/\partial W_c(t)$ term by sacrificing learning accuracy or including an additional system model network by introducing more computation burden. In this paper, we present a systematic examination on our proposed neural dynamic programming (NDP) design that includes two networks, the action and the critic, as building blocks. In the next two sections, we first introduce the building blocks of the proposed NDP implementations and then the associated on-line learning algorithms. In Section III, we provide evaluations on the on-line NDP designs for a single cart-pole balancing problem. Section IV gives evaluations of NDP designs in a pendulum swing up and balancing task. Section V includes simulation results of a more difficult on-line learning control problem, namely the triple-link inverted pendulum balancing task. After the presentation on NDP designs, algorithms, and performance evaluations, we try to provide some initial results on analytical insight of our on-line NDP designs using stochastic approximation argument. Finally, a section on conclusions and discussions is provided where we also provide some preliminary findings on improving the scalability of our proposed NDP designs.

II. A General Framework for Learning Through Association and Reinforcement

Fig. 1 is a schematic diagram of our proposed on-line learning control scheme. The binary reinforcement signal $r(t)$ is provided from the external environment and may be as simple as either a "0" or a "-1" corresponding to "success" or "failure," respectively.

In our on-line learning control design, the controller is "naive" when it just starts to control, namely the action network and the critic network are both randomly initialized in their weights/parameters. Once a system state is observed, an action will be subsequently produced based on the parameters in the action network. A "better" control value under the specific system state will lead to a more balanced equation of the principle of optimality. This set of system operations will be reinforced through memory or association between states and control output in the action network. Otherwise, the control value will be adjusted through tuning the weights in the action network in order to make the equation of the principle of optimality more balanced.

To be more quantitative, consider the critic network as depicted in Fig. 1. The output of the critic element, the J function, approximates the discounted total reward-to-go. Specifically, it approximates $R(t)$ at time t given by

$$R(t) = r(t+1) + \alpha r(t+2) + \cdots \tag{2}$$

where $R(t)$ is the future accumulative reward-to-go value at time t, α is a discount factor for the infinite-horizon problem ($0 < \alpha < 1$). We have used $\alpha = 0.95$ in our implementations. $r(t+1)$ is the external reinforcement value at time $t+1$.

A. The Critic Network

The critic network is used to provide $J(t)$ as an approximate of $R(t)$ in (2). We define the prediction error for the critic element as

$$e_c(t) = \alpha J(t) - [J(t-1) - r(t)] \tag{3}$$

and the objective function to be minimized in the critic network is

$$E_c(t) = \tfrac{1}{2}\, e_c^2(t). \tag{4}$$

The weight update rule for the critic network is a gradient-based adaptation given by

$$\mathbf{w}_c(t+1) = \mathbf{w}_c(t) + \Delta\mathbf{w}_c(t) \tag{5}$$

$$\Delta\mathbf{w}_c(t) = l_c(t)\left[-\frac{\partial E_c(t)}{\partial \mathbf{w}_c(t)}\right] \tag{6}$$

$$\frac{\partial E_c(t)}{\partial \mathbf{w}_c(t)} = \left[-\frac{\partial E_c(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial \mathbf{w}_c(t)}\right] \tag{7}$$

where $l_c(t) > 0$ is the learning rate of the critic network at time t, which usually decreases with time to a small value, and $\mathbf{w}_c$ is the weight vector in the critic network.

B. The Action Network

The principle in adapting the action network is to indirectly backpropagate the error between the desired ultimate objective, denoted by U_c, and the approximate J function from the critic network. Since we have defined "0" as the reinforcement signal for "success," U_c is set to "0" in our design paradigm and in our following case studies. In the action network, the state measurements are used as inputs to create a control as the output of the network. In turn, the action network can be implemented by either a linear or a nonlinear network, depending on the complexity of the problem. The weight updating in the action network can be formulated as follows. Let

$$e_a(t) = J(t) - U_c(t). \tag{8}$$

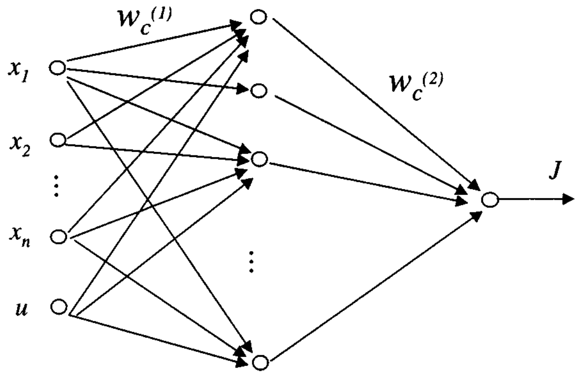


Fig. 2. Schematic diagram for the implementation of a nonlinear critic network using a feedforward network with one hidden layer.

The weights in the action network are updated to minimize the following performance error measure:

$$E_a(t) = \tfrac{1}{2}\, e_a^2(t). \tag{9}$$

The update algorithm is then similar to the one in the critic network. By a gradient descent rule

$$\mathbf{w}_a(t+1) = \mathbf{w}_a(t) + \Delta\mathbf{w}_a(t) \tag{10}$$

$$\Delta\mathbf{w}_a(t) = l_a(t)\left[-\frac{\partial E_a(t)}{\partial \mathbf{w}_a(t)}\right] \tag{11}$$

$$\frac{\partial E_a(t)}{\partial \mathbf{w}_a(t)} = \frac{\partial E_a(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial u(t)}\,\frac{\partial u(t)}{\partial \mathbf{w}_a(t)} \tag{12}$$

where $l_a(t) > 0$ is the learning rate of the action network at time t, which usually decreases with time to a small value, and $\mathbf{w}_a$ is the weight vector in the action network.

C. On-Line Learning Algorithms

Our on-line learning configuration introduced above involves two major components in the learning system, namely the action network and the critic network. In the following, we devise learning algorithms and elaborate on how learning takes place in each of the two modules.

In our NDP design, both the action network and the critic network are nonlinear multilayer feedforward networks. In our designs, one hidden layer is used in each network. The neural network structure for the nonlinear multilayer critic network is shown in Fig. 2.

In the critic network, the output $J(t)$ will be of the form

$$J(t) = \sum_{i=1}^{N_h} w_{c_i}^{(2)}(t) p_i(t) \tag{13}$$

$$p_i(t) = \frac{1 - \exp^{-q_i(t)}}{1 + \exp^{-q_i(t)}}, \qquad i = 1, \cdots, N_h \tag{14}$$

$$q_i(t) = \sum_{j=1}^{n+1} w_{c_{ij}}^{(1)}(t) x_j(t), \quad i = 1, \cdots, N_h \tag{15}$$

where

q_i ith hidden node input of the critic network;
p_i corresponding output of the hidden node;
N_h total number of hidden nodes in the critic network;

$n+1$ total number of inputs into the critic network including the analog action value $u(t)$ from the action network.

By applying the chain rule, the adaptation of the critic network is summarized as follows.

1) $\Delta\mathbf{w}_c^{(2)}$ (hidden to output layer)

$$\Delta w_{c_i}^{(2)}(t) = l_c(t)\left[-\frac{\partial E_c(t)}{\partial w_{c_i}^{(2)}(t)}\right] \tag{16}$$

$$\frac{\partial E_c(t)}{\partial w_{c_i}^{(2)}(t)} = \frac{\partial E_c(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial w_{c_i}^{(2)}(t)} = \alpha e_c(t) p_i(t). \tag{17}$$

2) $\Delta\mathbf{w}_c^{(1)}$ (input to hidden layer)

$$\Delta w_{c_{ij}}^{(1)}(t) = l_c(t)\left[-\frac{\partial E_c(t)}{\partial w_{c_{ij}}^{(1)}(t)}\right] \tag{18}$$

$$\frac{\partial E_c(t)}{\partial w_{c_{ij}}^{(1)}(t)} = \frac{\partial E_c(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial p_i(t)}\,\frac{\partial p_i(t)}{\partial q_i(t)}\,\frac{\partial q_i(t)}{\partial w_{c_{ij}}^{(1)}(t)} \tag{19}$$

$$= \alpha e_c(t) w_{c_i}^{(2)}(t)\left[\tfrac{1}{2}\left(1 - p_i^2(t)\right)\right] x_j(t). \tag{20}$$

Now, let us investigate the adaptation in the action network, which is implemented by a feedforward network similar to the one in Fig. 2 except that the inputs are the n measured states and the output is the action $u(t)$. The associated equations for the action network are

$$u(t) = \frac{1-\exp^{-v(t)}}{1+\exp^{-v(t)}} \tag{21}$$

$$v(t) = \sum_{i=1}^{N_h} w_{a_i}^{(2)}(t) g_i(t) \tag{22}$$

$$g_i(t) = \frac{1-\exp^{-h_i(t)}}{1+\exp^{-h_i(t)}}, \qquad i = 1, \ldots, N_h \tag{23}$$

$$h_i(t) = \sum_{j=1}^{n} w_{a_{ij}}^{(1)}(t) x_j(t), \qquad i = 1, \ldots, N_h \tag{24}$$

where v is the input to the action node, and g_i and h_i are the output and the input of the hidden nodes of the action network, respectively. Since the action network inputs the state measurements only, there is no $(n+1)$th term in (24) as in the critic network [see (15) for comparison]. The update rule for the nonlinear multilayer action network also contains two sets of equations.

1) $\Delta\mathbf{w}_a^{(2)}$ (hidden to output layer)

$$\Delta w_{a_i}^{(2)}(t) = l_a(t)\left[-\frac{\partial E_a(t)}{\partial w_{a_i}^{(2)}(t)}\right] \tag{25}$$

$$\frac{\partial E_a(t)}{\partial w_{a_i}^{(2)}(t)} = \frac{\partial E_a(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial u(t)}\,\frac{\partial u(t)}{\partial v(t)}\,\frac{\partial v(t)}{\partial w_{a_i}^{(2)}(t)} \tag{26}$$

$$= e_a(t)\left[\tfrac{1}{2}\left(1-u^2(t)\right)\right] g_i(t)\sum_{i=1}^{N_h} \cdot\left[w_{c_i}^{(2)}(t)\,\tfrac{1}{2}\left(1-p_i^2(t)\right) w_{c_i,\,n+1}^{(1)}(t)\right]. \tag{27}$$

In the above equations, $\partial J(t)/\partial u(t)$ is obtained by changing variables and by chain rule. The result is the summation term. $w_{c_i,\,n+1}^{(1)}$ is the weight associated with the input element from the action network.

2) $\Delta\mathbf{w}_a^{(1)}$ (input to hidden layer)

$$\Delta w_{a_{ij}}^{(1)}(t) = l_a(t)\left[-\frac{\partial E_a(t)}{\partial w_{a_{ij}}^{(1)}(t)}\right], \tag{28}$$

$$\frac{\partial E_a(t)}{\partial w_{a_{ij}}^{(1)}(t)} = \frac{\partial E_a(t)}{\partial J(t)}\,\frac{\partial J(t)}{\partial u(t)}\,\frac{\partial u(t)}{\partial v(t)}\,\frac{\partial v(t)}{\partial g_i(t)}\,\frac{\partial g_i(t)}{\partial h_i(t)}\,\frac{\partial h_i(t)}{\partial w_{a_{ij}}^{(1)}(t)} \tag{29}$$

$$= e_a(t)\left[\tfrac{1}{2}\left(1-u^2(t)\right)\right] w_{a_i}^{(2)}(t)\left[\tfrac{1}{2}\left(1-g_i^2(t)\right)\right] x_j(t) \cdot \sum_{i=1}^{N_h}\left[w_{c_i}^{(2)}(t)\,\tfrac{1}{2}\left(1-p_i^2(t)\right) w_{c_i,\,n+1}^{(1)}(t)\right]. \tag{30}$$

Normalization is performed in both networks to confine the values of the weights into some appropriate range by

$$\mathbf{w}_c(t+1) = \frac{\mathbf{w}_c(t) + \Delta\mathbf{w}_c(t)}{\|\mathbf{w}_c(t) + \Delta\mathbf{w}_c(t)\|_1} \tag{31}$$

$$\mathbf{w}_a(t+1) = \frac{\mathbf{w}_a(t) + \Delta\mathbf{w}_a(t)}{\|\mathbf{w}_a(t) + \Delta\mathbf{w}_a(t)\|_1}. \tag{32}$$

In implementation, (17) and (20) are used to update the weights in the critic network and (27) and (30) are used to update the weights in the action network.

III. Performance Evaluation for Case Study One

The proposed NDP design has been implemented on a single cart-pole problem. To begin with, the self-learning controller has no prior knowledge about the plant but only on-line measurements. The objective is to balance a single pole mounted on a cart, which can move either to the right or to the left on a bounded, horizontal track. The goal for the learning controller is to provide a force (applied to the cart) of a fixed magnitude in either the right or the left direction so that the pole stands balanced and avoids hitting the track boundaries. The controller receives reinforcement only after the pole has fallen.

In order to provide the learning controller measured states as inputs to the action and the critic networks, the cart-pole system was simulated on a digital computer using a detailed model that includes all of the nonlinearities and reactive forces of the physical system such as frictions. Note that these simulated states would be the measured ones in real-time applications.

A. The Cart-Pole Balancing Problem

The cart-pole system used in the current study is the same as the one in [1].

$$\frac{d^2\theta}{dt^2} = \frac{g\sin\theta + \cos\theta\left[-F - ml\dot{\theta}^2\sin\theta + \mu_c\,\mathrm{sgn}(\dot{x})\right] - \dfrac{\mu_p\dot{\theta}}{ml}}{l\left(\dfrac{4}{3} - \dfrac{m\cos^2\theta}{m_c + m}\right)} \tag{33}$$

$$\frac{d^2x}{dt^2} = \frac{F + ml\left[\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right] - \mu_c\mathrm{sgn}(\dot{x})}{m_c + m} \tag{34}$$

where

g 9.8 m/s^2, acceleration due to gravity;
m_c 1.0 kg, mass of cart;
m 0.1 kg, mass of pole;
l 0.5 m, half-pole length;
μ_c 0.0005, coefficient of friction of cart on track;
μ_p 0.000 002, coefficient of friction of pole on cart;
F ±10 Newtons, force applied to cart's center of mass;

$$\text{sgn}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0. \end{cases}$$

The nonlinear differential equations (33) and (34) are numerically solved by a fourth-order Runge–Kutta method. This model provides four state variables: 1) $x(t)$, position of the cart on the track; 2) $\theta(t)$, angle of the pole with respect to the vertical position; 3) $\dot{x}(t)$, cart velocity; 4) $\dot{\theta}(t)$, angular velocity.

In our current study a run consists of a maximum of 1000 consecutive trials. It is considered successful if the last trial (trial number less than 1000) of the run has lasted 600 000 time steps. Otherwise, if the controller is unable to learn to balance the cart-pole within 1000 trials (i.e., none of the 1000 trials has lasted over 600 000 time steps), then the run is considered unsuccessful. In our simulations, we have used 0.02 s for each time step, and a trial is a complete process from start to fall. A pole is considered fallen when the pole is outside the range of $[-12^\circ, 12^\circ]$ and/or the cart is beyond the range of $[-2.4, 2.4]$ m in reference to the central position on the track. Note that although the force F applied to the cart is binary, the control $u(t)$ fed into the critic network as shown in Fig. 1 is continuous.

B. Simulation Results

Several experiments were conducted to evaluate the effectiveness of our learning control designs. The parameters used in the simulations are summarized in Table I with the proper notations defined in the following:

$l_c(0)$ initial learning rate of the critic network;
$l_a(0)$ initial learning rate of the action network;
$l_c(t)$ learning rate of the critic network at time t which is decreased by 0.05 every five time steps until it reaches 0.005 and it stays at $l_c(f) = 0.005$ thereafter;
$l_a(t)$ learning rate of the action network at time t which is decreased by 0.05 every five time steps until it reaches 0.005 and it stays at $l_a(f) = 0.005$ thereafter;
N_c internal cycle of the critic network;
N_a internal cycle of the action network;
T_c internal training error threshold for the critic network;
T_a internal training error threshold for the action network;
N_h number of hidden nodes.

Note that the weights in the action and the critic networks were trained using their internal cycles, N_a and N_c, respectively. That is, within each time step the weights of the two networks were updated for at most N_a and N_c times, respectively, or stopped once the internal training error threshold T_a and T_c have been met.

TABLE I
SUMMARY OF PARAMETERS USED IN OBTAINING THE RESULTS GIVEN IN TABLE II

Parameter	$l_c(0)$	$l_a(0)$	$l_c(f)$	$l_a(f)$	*
Value	0.3	0.3	0.005	0.005	*
Parameter	N_c	N_a	T_c	T_a	N_h
Value	50	100	0.05	0.005	6

TABLE II
PERFORMANCE EVALUATION OF NDP LEARNING CONTROLLER WHEN BALANCING A CART-POLE SYSTEM. THE SECOND COLUMN REPRESENTS THE PERCENTAGE OF SUCCESSFUL RUNS OUT OF 100. THE THIRD COLUMN DEPICTS THE AVERAGE NUMBER OF TRIALS IT TOOK TO LEARN TO BALANCE THE CART-POLE. THE AVERAGE IS TAKEN OVER THE SUCCESSFUL RUNS

Noise type	success rate	# of trials
Noise free	100%	6
Uniform 5% actuator	100%	8
Uniform 10% actuator	100%	14
Uniform 5% sensor	100%	32
Uniform 10% sensor	100%	54
Gaussian $\sigma^2 = 0.1$ sensor	100%	164
Gaussian $\sigma^2 = 0.2$ sensor	100%	193

To be more realistic, we have added both sensor and actuator noise to the state measurements and the action network output. Specifically, we implemented the actuator noise through $u(t) = u(t) + \rho$, where ρ is a uniformly distributed random variable. For the sensor noise, we experimented with adding both uniform and Gaussian random variables to the angle measurements θ. The uniform state sensor noise was implemented through $\theta = (1 + \text{noise percentage}) \times \theta$. Gaussian sensor noise was zero mean with specified variance.

Our proposed configuration of neural dynamic programming has been evaluated and the results are summarized in Table II. The simulation results summarized in Table II were obtained through averaged runs. Specifically, 100 runs were performed to obtain the results reported here. Each run was initialized to random conditions in terms of network weights. If a run is successful, the number of trials it took to balance the cart-pole is then recorded. The number of trials listed in the table corresponds to the one averaged over all of the successful runs. Therefore there is a need to record the percentage of successful runs out of 100. This number is also recorded in the table. A good configuration is the one with a high percentage of successful runs as well as a low average number of trials needed to learn to perform the balancing task.

Fig. 3 shows a typical movement or trajectory of the pendulum angle under NDP controller for a successful learning trial. The system under consideration is not subject to any noise.

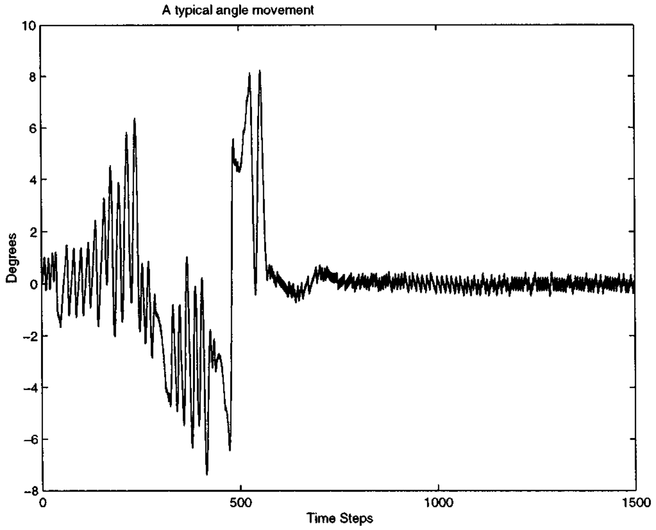


Fig. 3. A typical angle trajectory during a successful learning trial for the NDP controller when the system is free of noise.

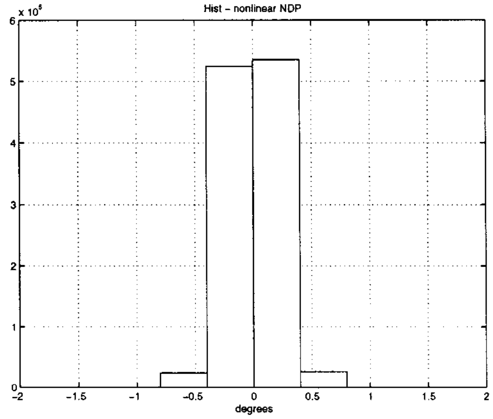


Fig. 4. Histogram of angle variations under the control of NDP on-linear learning mechanism in the single cart-pole problem. The system is free of noise in this case.

Fig. 4 represents a summary of typical statistics of the learning process in histograms. It contains vertical angle histograms when the system learns to balance the cart-pole using ideal state measurements without noise corruption.

IV. Performance Evaluation for Case Study Two

We now examine the performance of the proposed NDP design in a pendulum swing up and balancing task. The case under study is identical to the one in [15].

The pendulum is held by one end and can swing in a vertical plane. The pendulum is actuated by a motor that applied a torque at the hanging point. The dynamics of the pendulum is as follows:

$$\frac{d\omega}{dt} = \frac{3}{4ml^2}(F + mlg\sin(\theta)) \tag{35}$$

$$\frac{d\theta}{dt} = \omega \tag{36}$$

where $m = 1/3$ and $l = 3/2$ are the mass and length of the pendulum bar, respectively, and $g = 9.8$ is the gravity. The action is the angular acceleration F and it is bounded between -3 and 3, namely, $F_{\min} = -3$, and $F_{\max} = 3$. A control action is applied every four time steps. The system states are the current angle θ and the angular velocity ω. This task requires the controller to not only swing up the bar but also to balance it at the top position. The pendulum initially sits still at $\theta = \pi$. This task is considered difficult in the sense that 1) no closed-form analytical solution exists for the optimal solution and complex numerical methods are required to compute it and 2) the maximum and minimum angular acceleration values are not strong enough to move the pendulum straight up from the starting state without first creating angular momentum [15].

TABLE III
Performance Evaluation of NDP Learning Controller to Swing up and then Balance a Pendulum. The Second Column Represents the Percentage of Successful Runs out of 60. The Third Column Depicts the Average Number of Trials it Took to Learning to Successfully Perform the Task. The Average is Taken over the Successful Runs

reinforcement implementation	success rate	# of trials
Setting 1	100%	4.2
Setting 2	96%	3.5

In this study, a run consists of a maximum of 100 consecutive trials. It is considered successful if the last trial (trial number less than 100) of the run has lasted 800 time steps (with a step size of 0.05 s). Otherwise, if the NDP controller is unable to swing up and keep the pendulum balanced at the top within 100 trials (i.e., none of the 100 trails has lasted over 800 time steps), then the run is considered unsuccessful. In our simulations, a trial is either terminated at the end of the 800 time steps or when the angular velocity of the pendulum is greater than 2π, i.e., $\omega > 2\pi$.

In the following, we studied two implementation scenarios with different settings in reinforcement signal r. In **Setting 1**, $r = 0$ when the angle displacement is within 90° from the position of $\theta = 0$; $r = -0.4$ when the angle is in the rest half of the plane; and $r = -1$ when the angular velocity $\omega > 2\pi$. In **Setting 2**, $r = 0$ when the angle displacement is within 10° from the position of $\theta = 0$; $r = -0.4$ when the angle is in the remaining area of the plane; and $r = -1$ when the angular velocity $\omega > 2\pi$.

Our proposed NDP configuration is then used to perform the above described task. We have used the same configuration and the same learning parameters as those in the first case study. NDP controller performance is summarized in Table III. The simulation results summarized in the table were obtained through averaged runs. Specifically, 60 runs were performed to obtain the results reported here. Note that we have used more runs than that in [15] (which was 36) to generate the final result statistics. But we have kept every other simulation condition the same as that in the paper [15]. Each run was initialized to $\theta = \pi$ and $\omega = 0$. The number of trials listed in the table corresponds to the one averaged over all of the successful runs. The percentage of successful runs out of 60 was also recorded

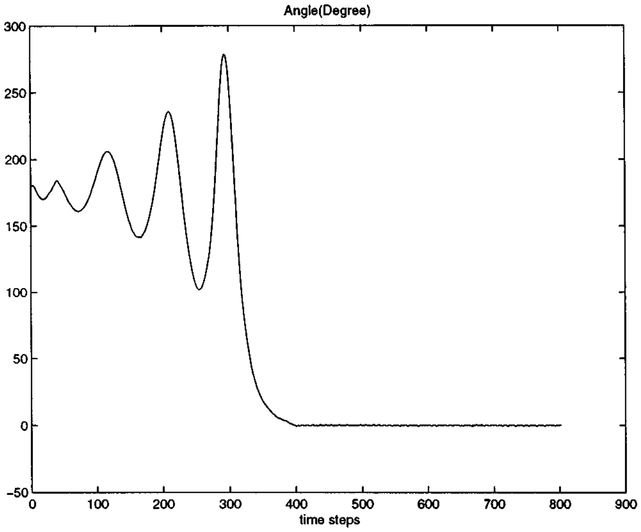


Fig. 5. A typical angle trajectory during a successful learning trial for the NDP controller in the pendulum swing up and balancing task. *Left*: the entire trial. *Right*: portion of the entire trajectory to show the stabilization process in detail.

in the table. Fig. 5 shows a typical trajectory of the pendulum angle under NDP controller for a successful learning trial. This trajectory is characteristic for both Setting 1 and Setting 2.

V. Performance Evaluation for Case Study Three

The NDP design introduced in the previous sections is now applied to a more complex on-line learning control problem than the single cart-pole balancing task, namely, the triple-link inverted pendulum problem with single control input. We have successfully implemented our proposed NDP configuration on this problem. The details of the implementation and results will be given in the following subsections.

A. Triple-Link Inverted Pendulum with Single Control Input

The system model for the triple-link problem is the same as that in [7]. Fig. 6 depicts the notation used in the state equations that govern the system.

The equation governing the system is

$$F(q)\,\frac{d^2q}{dt^2} = -G\left(q,\,\frac{dq}{dt}\right)\frac{dq}{dt} - H(q) + L(q,\,u) \qquad (37)$$

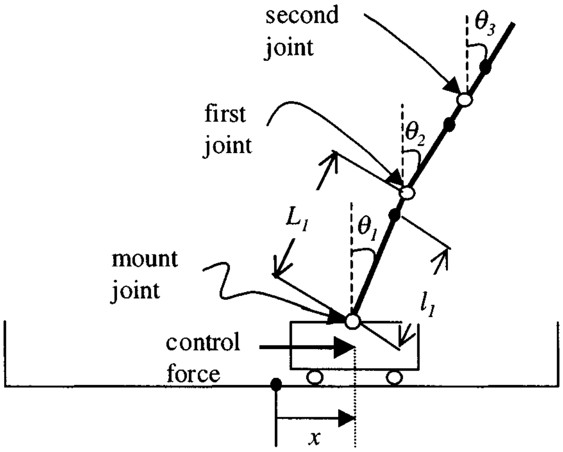


Fig. 6. Definition of notation used in the system equations for the triple-link inverted pendulum problem.

where the components are shown in the equation at the bottom of the page. Note that the μ's in $L(q,\,u)$ are Coulomb friction coefficients for links and they are not linearizable [7]. In our simulations, $\mu_x = 0.07$, and $\mu_1 = \mu_2 = \mu_3 = 0.003$. The A_i coefficients are system constants and are given in Table IV.

The parameters used in the system are defined as follows:

g 9.8 m/s^2, acceleration due to gravity;
M 1.014 kg, mass of the cart;
m_1 0.4506 kg, mass of the first link;
m_2 0.219 kg, mass of the second link;
m_3 0.0568 kg, mass of the third link;
l_1 0.37 m, the length from the mount joint to the center of gravity of the first link;
l_2 0.3 m, the length from the first joint to the center of gravity of the second link;
l_3 0.05 m, the length from the second joint to the center of gravity of the third link;
L_1 0.43 m, total length of the first link;
L_2 0.33 m, total length of the second link;
L_3 0.13 m, total length of the third link;
I_1 0.0042 kgm^2, mass moment of inertia of the first link about its center of gravity;
I_2 0.0012 kgm^2, mass moment of inertia of the second link about its center of gravity;

$$F(q) = \begin{bmatrix} A_1 & A_2\cos(\theta_1) & A_3\cos(\theta_2) & A_4\cos(\theta_3) \\ A_9\cos(\theta_1) & A_{10} & A_{11}\cos(\theta_1-\theta_2) & A_{12}\cos(\theta_1-\theta_3) \\ A_{18}\cos(\theta_2) & A_{19}\cos(\theta_1-\theta_2) & A_{20} & A_{21}\cos(\theta_2-\theta_3) \\ A_{28}\cos(\theta_3) & A_{29}\cos(\theta_1-\theta_3) & A_{30}\cos(\theta_2-\theta_3) & A_{31} \end{bmatrix}$$

$$G\left(q,\,\frac{dq}{dt}\right) = \begin{bmatrix} A_5 & A_6\sin(\theta_1)\dot{\theta}_1 & A_7\sin(\theta_2)\dot{\theta}_2 & A_8\sin(\theta_3)\dot{\theta}_3 \\ 0 & A_{13} & A_{14}\sin(\theta_1-\theta_2)\dot{\theta}_2 + A_{15} & A_{16}\sin(\theta_1-\theta_3)\dot{\theta}_3 \\ 0 & A_{22}\sin(\theta_1-\theta_2)\dot{\theta}_1 + A_{23} & A_{24} & A_{25}\sin(\theta_2-\theta_3)\dot{\theta}_3 + A_{26} \\ 0 & A_{33}\sin(\theta_1-\theta_3)\dot{\theta}_1 & A_{35}\sin(\theta_2-\theta_3)\dot{\theta}_2 + A_{36} & A_{32} \end{bmatrix}$$

$$q = \begin{bmatrix} x \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}, \quad H(q) = \begin{bmatrix} 0 \\ A_{17}\sin(\theta_1) \\ A_{27}\sin(\theta_2) \\ A_{34}\sin(\theta_3) \end{bmatrix}, \quad L(q,\,u) = \begin{bmatrix} K_s u - \operatorname{sgn}(x)\mu_x A_{37} \\ -\operatorname{sgn}(\theta_1)\mu_1 A_{38} \\ -\operatorname{sgn}(\theta_2)\mu_2 A_{39} \\ -\operatorname{sgn}(\theta_3)\mu_3 A_{40} \end{bmatrix}.$$

TABLE IV
SYSTEM CONSTANTS FOR THE TRIPLE LINK INVERTED PENDULUM PROBLEM

Constant	Value	Constant	Value
A_1	$M+m_1+m_2+m_3$	A_{21}	$m_3l_3L_2$
A_2	$m_1l_1+(m_2+m_3)L_1$	A_{22}	$-(m_2l_2+m_3L_2)L_1$
A_3	$m_2l_2+m_3L_2$	A_{23}	$-C_2$
A_4	m_3l_3	A_{24}	C_2+C_3
A_5	C_c	A_{25}	$m_3l_3L_2$
A_6	$-m_1l_1-(m_2+m_3)L_1$	A_{26}	$-C_3$
A_7	$-(m_2l_2+m_3L_2)$	A_{27}	$-g(m_2l_2+m_3L_2)$
A_8	$-m_3l_3$	A_{28}	m_3l_3
A_9	$m_1l_1+(m_2+m_3)L_1$	A_{29}	$m_3l_3L_1$
A_{10}	$I_1+m_2l_1^2+(m_2+m_3)L_1^2$	A_{30}	$m_3l_3L_2$
A_{11}	$(m_2l_2+m_3L_2)L_1$	A_{31}	$I_3+m_3l_3^2$
A_{12}	$m_3l_3L_1$	A_{32}	C_3
A_{13}	C_1+C_2	A_{33}	$-m_3l_3L_1$
A_{14}	$(m_2l_2+m_3L_2)L_1$	A_{34}	$-gm_3l_3$
A_{15}	$-C_2$	A_{35}	$-m_3l_3L_2$
A_{16}	$m_3l_3L_1$	A_{36}	$-C_3$
A_{17}	$-g(m_1l_1+m_2L_1+m_3L_1)$	A_{37}	1.3
A_{18}	$m_2l_2+m_3L_2$	A_{38}	0.506
A_{19}	$(m_2l_2+m_3L_2)L_1$	A_{39}	0.219
A_{20}	$I_2+m_3L_2^2+m_2l_2^2$	A_{40}	0.568

I_3 0.000 106 09 kgm^2, mass moment of inertia of the third link about its center of gravity;

C_c 5.5 Nms, dynamic friction coefficient between the cart and the track;

C_1 0.000 268 75 Nms, dynamic friction coefficient for the first link;

C_2 0.000 268 75 Nms, dynamic friction coefficient for the second link;

C_3 0.000 268 75 Nms, dynamic friction coefficient for the third link.

The only control u (in volts) generated by the action network is converted into force by an analog amplifier through a conversion gain K_s (in Newtons/volt). In simulations, $K_s = 24.7125$ N/V. Each link can only rotate in the vertical plane about the axis of a position sensor fixed to the top of each link. The sampling time interval is chosen to be 5 ms. From the nonlinear dynamical equation in (37), the state-space model can be described as follows:

$$\dot{Q}(t) = f(Q(t),\, u(t)) \tag{38}$$

with

$$\begin{aligned} f(Q(t),\, u(t)) &= \begin{bmatrix} \mathbf{0}_{4\times4} & \mathbf{I}_{4\times4} \\ \mathbf{0}_{4\times4} & -F^{-1}(Q(t))G(Q(t)) \end{bmatrix} Q(t) \\ &\quad + \begin{bmatrix} \mathbf{0}_{4\times4} \\ -F^{-1}(Q(t))[H(Q(t))-L(Q(t),\, u(t))] \end{bmatrix} \end{aligned}$$

and

$$Q(t) = \left[x(t)\; \theta_1(t)\; \theta_2(t)\; \theta_3(t)\; \dot{x}(t)\; \dot{\theta}_1(t)\; \dot{\theta}_2(t)\; \dot{\theta}_3(t)\right]^T.$$

TABLE V
SUMMARY OF PARAMETERS USED IN OBTAINING THE RESULTS GIVEN IN TABLE VI FOR THE TRIPLE-LINK INVERTED PENDULUM PROBLEM

Parameter	$l_c(0)$	$l_a(0)$	$l_c(f)$	$l_a(f)$	*
Value	0.8	0.8	0.001	0.001	*
Parameter	N_c	N_a	T_c	T_a	N_h
Value	10	200	0.01	0.001	14

There are eight state variables in this model: 1) $x(t)$, position of the cart on the track; 2) $\theta_1(t)$, vertical angle of the first link joint to the cart; 3) $\theta_2(t)$, vertical angle of the second link joint to the first link; 4) $\theta_3(t)$, vertical angle of the third link joint to the second link; 5) $\dot{x}(t)$, cart velocity; 6) $\dot{\theta}_1(t)$, angular velocity of $\theta_1(t)$; 7) $\dot{\theta}_2(t)$, angular velocity of $\theta_2(t)$; and 8) $\dot{\theta}_3(t)$, angular velocity of $\theta_3(t)$.

In the triple-link inverted pendulum problem, a run consists of a maximum of 3000 consecutive trials. Similar to the single cart-pole balancing problem, a run is considered successful if the last trial of the run lasts 600 000 time steps. However, now a unit time step is 5 ms instead of 20 ms as in the single cart-pole problem. The constraints for the reinforcement learning are: 1) the cart track extends 1.0 m to both ends from the center position; 2) the voltage applied to the motor is within [−30, 30] V range; and 3) each link angle should be within the range of [−20°, 20°] with respect to the vertical axis. In our simulations, condition 2 is assured to be satisfied by using a sigmoid function at the output of the action node. For conditions 1) and 3), if either one fails or both fail, the system provides an indicative signal $r = -1$ at the moment of failure, otherwise $r = 0$ all the time. Several experiments were conducted to evaluate the effectiveness of the proposed learning control designs. The results are reported in the following section.

B. Simulation Results

Note that the triple-link system is highly unstable. To see this, the positive eigenvalues of the linearized system model are far away from zero (the largest is around 10.0). In obtaining the linearized system model, the Coulomb friction coefficients are assumed to be negligible. Besides, the system dynamics changes fast. It requires a sampling time below 10 ms.

Since the analog output from the action network is directly fed into the system this time, the controller is more sensitive to the actuator noise than the one in the single cart-pole where a binary control is applied. Experiments conducted in this paper include evaluations of NDP controller performance under uniform actuator noise, uniform or Gaussian sensor noise, and the case without noise.

Before the presentation of our results, the learning parameters are summarized in Table V. Simulation results are tabulated in Table VI. Conventions such as noise type and how they are included in the simulations are described in Section III-B.

Fig. 7 shows typical angle trajectories of the triple-link angles under NDP control for a successful learning trial. The system under consideration is not subject to any noise. The corresponding control force trajectory is also given in Fig. 8.

TABLE VI
PERFORMANCE EVALUATION OF NDP LEARNING CONTROLLER WHEN BALANCING A TRIPLE-LINK INVERTED PENDULUM. THE SECOND COLUMN REPRESENTS THE PERCENTAGE OF SUCCESSFUL RUNS OUT OF 100. THE THIRD COLUMN DEPICTS THE AVERAGE NUMBER OF TRIALS IT TOOK TO LEARNING TO BALANCE THE CART POLE. THE AVERAGE IS TAKEN OVER THE SUCCESSFUL RUNS

Noise type	success rate	# of trials
None	97%	1194
Uniform 5% actuator	92%	1239
Uniform 10% actuator	84%	1852
Uniform 5% sensor on θ_1	89%	1317
Uniform 10% sensor on θ_1	80%	1712
Gaussian sensor on θ_1 variance = 0.1	85%	1508
Gaussian sensor on θ_1 variance = 0.2	76%	1993

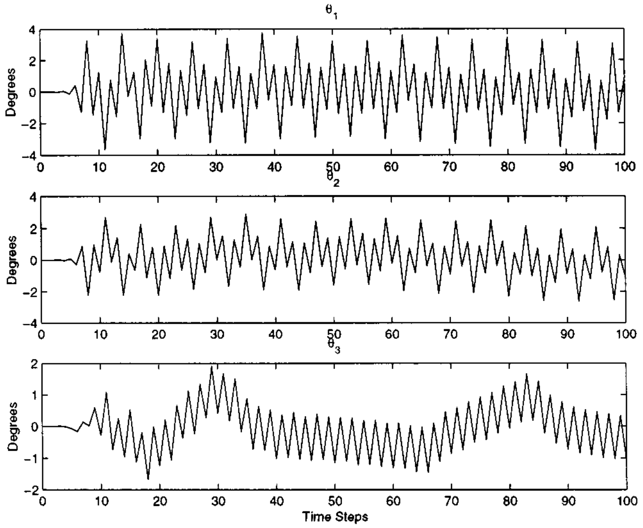


Fig. 7. Typical angle trajectories of the triple-link angles during a successful learning trial using on-line NDP control when the system is free of noise.

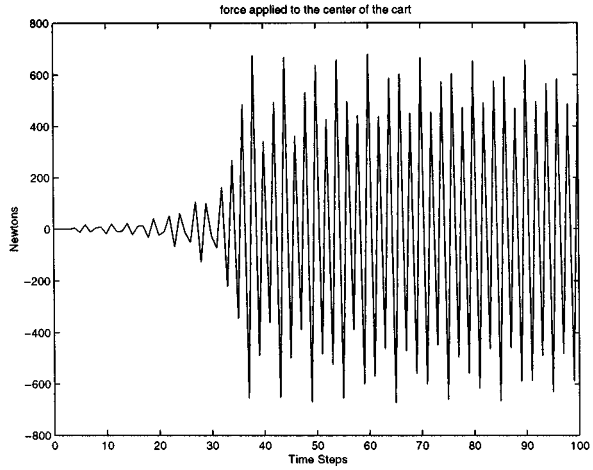


Fig. 8. The force (in Newton) trajectory, which is converted from control u into voltage, applied to the center of the cart for balancing the triple-link inverted pendulum, corresponding to the angle trajectory in Fig. 7.

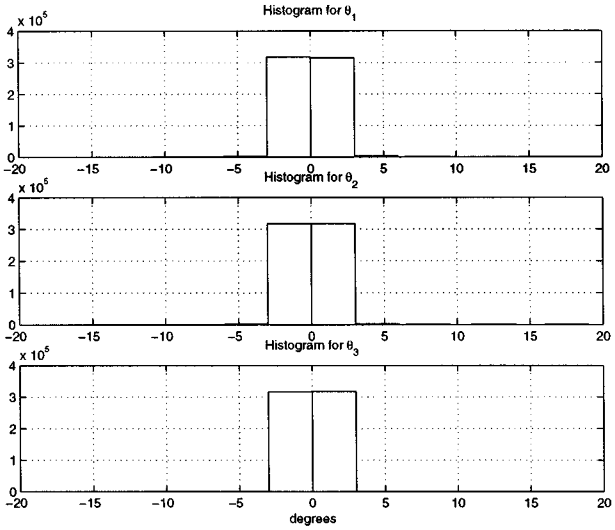


Fig. 9. Histogram of the triple-link angle variations when the system is free of noise.

Fig. 9 represents a summary of statistics of the learning process in histograms.

The results presented in this case study have again demonstrated the validity of the proposed NDP designs. The major characteristics of the learning process for this more complex system are still similar to the single cart-pole problem in many ways. It is worth mentioning that the NDP controlled angle variations are significantly smaller than those using nonlinear control system design as in [7].

VI. Analytical Characteristics of On-Line NDP Learning Process

This section is dedicated to expositions of analytical properties of the on-line learning algorithms in the context of NDP. It is important to note that in contrast to usual neural-network applications, there is no readily available training sets of input–output pairs to be used for approximating J^* in the sense of least squares fit in NDP applications. Both the control action u and the approximated J function are updated according to an error function that changes from one time step to the next. Therefore, the convergence argument for the steepest descent algorithm does not hold valid for any of the two networks, action or critic. This results in a simulation approach to evaluate the cost-to-go function J for a given control action u. The on-line learning takes place aiming at iteratively improving the control policies based on simulation outcomes. This creates analytical and computational difficulties that do not arise in a more typical neural-network training context.

Some analytical results in terms of approximating J function was obtained by Tsitsiklis [20] where a linear in parameter function approximator was used to approximate the J function. The limit of convergence was characterized as the solution to a set of interpretable linear equations, and a bound is placed on the resulting approximation error.

It is worth pointing out that the existing implementations of NDP are usually computationally very intensive [4], and often require a considerable amount of trial and error. Most of the computations and experimentations with different approaches were conducted off-line.

In the following, we try to provide some analytical insight on the on-line learning process for our proposed NDP designs. Specifically we will use the stochastic approximation argument to reveal the asymptotic performance of our on-line NDP learning algorithms in an averaged sense for the action and the critic networks under certain conditions.

A. Stochastic Approximation Algorithms

The original work in recursive stochastic algorithms was introduced by Robbins and Monro, who developed and analyzed a recursive procedure for finding the root of a real-valued function $g(w)$ of a real variable w [14]. The function is not known, but noise-corrupted observations could be taken at values of w selected by the experimenter.

A function $g(w)$ with the form $g(w) = Ex[f(w)]$ ($Ex[\,]$ is the expectation operator) is called a regression function of $f(w)$, and conversely, $f(w)$ is called a sample function of $g(w)$. The following conditions are needed to obtain the Robbins–Monro algorithm [14].

(C1) $g(w)$ has a single root w^*, $g(w^*) = 0$, and

$$g(w) < 0 \quad \text{if } w < w^*$$
$$g(w) > 0 \quad \text{if } w > w^*.$$

This is assumed with little loss of generality since most functions of a single root not satisfying this condition can be made to do so by multiplying the function by -1.

(C2) The variance of $f(w)$ from $g(w)$ is finite

$$\sigma^2(w) = Ex[g(w) - f(w)]^2 < \infty. \tag{39}$$

(C3)

$$|g(w)| < B_1|w - w^*| + B_0 < \infty. \tag{40}$$

(C3) is a very mild condition. The values of B_1 and B_0 need not be known to prove the validity of the algorithm. As long as the root lies in some finite interval, the existence of B_1 and B_0 can always be assumed.

If the conditions (C1) through (C3) are satisfied, the algorithm due to Robbins and Monro can be used to iteratively seek the root w^* of the function $g(w)$:

$$w(t+1) = w(t) - l(t)f[w(t)] \tag{41}$$

where $l(t)$ is a sequence of positive numbers which satisfy the following conditions:

$$\begin{aligned} &1) \quad \lim_{t\to\infty} l(t) = 0 \\ &2) \quad \sum_{t=0}^{\infty} l(t) = \infty \\ &3) \quad \sum_{t=0}^{\infty} l^2(t) < \infty. \end{aligned} \tag{42}$$

Furthermore, $w(t)$ will converge toward w^* in the mean square error sense and with probability one, i.e.,

$$\lim_{t\to\infty} Ex\left[||w(t) - w^*||^2\right] = 0 \tag{43}$$

$$\text{Prob}\left\{\lim_{t\to\infty} w(t) = w^*\right\} = 1. \tag{44}$$

The convergence with probability one in (44) is also called convergence almost truly.

In this paper, the Robbins–Monro algorithm is applied to optimization problems [10]. In that setting, $g(w) = \partial E/\partial w$, where E is an objective function to be optimized. If E has a local optimum at w^*, $g(w)$ will satisfy the condition (C1) locally at w^*. If E has a quadratic form, $g(w)$ will satisfy the condition (C1) globally.

B. Convergence in Statistical Average for the Action and the Critic Networks

Neural dynamic programming is still in its early stage of development. The problem is not trivial due to several consecutive learning segments being updated simultaneously. A practically effective on-line learning mechanism and a step by step analytical guide for the learning process do not co-exist at this time. This paper is dedicated to reliable implementations of NDP algorithms for solving a general class of on-line learning control problem. As demonstrated in previous sections, experimental results in this direction are very encouraging. In the present section, we try to provide some asymptotic convergence results for each component of the NDP system. The Robbins–Monro algorithm provided in the previous section is the main tool to obtain results in this regard. Throughout this paper, we have implied that the state measurements are samples of a continuous state space. Specifically we will assume without loss of generality that the input $X_j \in \mathcal{X} \subset \mathcal{R}^n$ has discrete probability density $p(X) = \sum_{j=1}^{N} p_j\delta(X - X_j)$, where $\delta(\,)$ is the delta function.

In the following, we analyze one component of the NDP system at a time. When one component (e.g., the action network) is under consideration, the other component (e.g., the critic network) is considered to have completed learning, namely their weights do not change any more.

To examine the learning process taking place in the action network, we define the following objective function for the action network:

$$\begin{aligned} \tilde{E}_a &= \tfrac{1}{2}\sum_i p_i[J(X_i) - U_c]^2 \\ &= \tfrac{1}{2}Ex[(J - U_c)^2]. \end{aligned} \tag{45}$$

It can be seen that (45) is an "averaged" error square between the estimated J and a final desired value U_c. To contrast this notion, (9) is an "instantaneous" error square between the two. To obtain a (local) minimum for the "averaged" error measure in (45), we can apply the Robbins–Monro algorithm by first taking a derivative of this error with respect to the parameters, which are the weights in the action network in this case. Let

$$\tilde{e}_a = J - U_c. \tag{46}$$

Since J is smooth in $\tilde{\mathbf{w}}_a$, and $\tilde{\mathbf{w}}_a$ belongs to a bounded set, the derivative of $\tilde{E}_a$ with respect to the weights of the action network is then of the form:

$$\frac{\partial \tilde{E}_a}{\partial \tilde{\mathbf{w}}_a} = Ex\left[\tilde{e}_a \frac{\partial \tilde{e}_a}{\partial \tilde{\mathbf{w}}_a}\right]. \tag{47}$$

According to the Robbins–Monro algorithm, the root (can be a local root) of $\partial\tilde{E}_a/\partial\tilde{\mathbf{w}}_a$ as a function of $\tilde{\mathbf{w}}_a$ can be obtained by the following recursive procedure, if the root exists and if the step size $l_a(t)$ meets all the requirements described in (42):

$$\tilde{\mathbf{w}}_a(t+1) = \tilde{\mathbf{w}}_a(t) - l_a(t)\left[\tilde{e}_a \frac{\partial \tilde{e}_a}{\partial \tilde{\mathbf{w}}_a}\right]. \tag{48}$$

Equation (46) may be considered as an instantaneous error between a sample of the J function and the desired value U_c. Therefore, (48) is equivalent to the update equation for the action network given in (10)–(12). From this viewpoint, the on-line action network updating rule of (10)–(12) is actually converging to a (local) minimum of the error square between the J function and the desired value U_c in a statistical average sense. Or in other words, even though (10)–(12) represent a reduction in instantaneous error square at each iterative time step, the action network updating rule asymptotically reaches a (local) minimum of the statistical average of $(J - U_c)^2$.

By the same token, we can construct a similar framework to describe the convergence of the critic network. Recall that the residual of the principle of optimality equation to be balanced by the critic network is of the following form:

$$e_c(t) = \alpha J(t) - J(t-1) + r(t). \tag{49}$$

And the "instantaneous" error square of this residual is given as

$$E_c(t) = \tfrac{1}{2}\, e_c^2(t). \tag{50}$$

Instead of the "instantaneous" error square, let

$$\tilde{E}_c = Ex[E_c] \tag{51}$$

and assume that the expectation is well defined over the discrete state measurements. The derivative of $\tilde{E}_c$ with respect to the weights of the critic network is then of the form

$$\frac{\partial \tilde{E}_c}{\partial \tilde{\mathbf{w}}_c} = Ex\left[e_c \frac{\partial e_c}{\partial \tilde{\mathbf{w}}_c}\right]. \tag{52}$$

According to the Robbins–Monro algorithm, the root (can be a local root) of $\partial \tilde{E}_c / \partial \tilde{\mathbf{w}}_c$ as a function of $\tilde{\mathbf{w}}_c$ can be obtained by the following recursive procedure, if the root exists and if the step size $l_c(t)$ meets all the requirements described in (42):

$$\tilde{\mathbf{w}}_c(t+1) = \tilde{\mathbf{w}}_c(t) - l_c(t)\left[e_c \frac{\partial e_c}{\partial \tilde{\mathbf{w}}_c}\right]. \tag{53}$$

Therefore, (53) is equivalent to the update rule for the critic network given in (5)–(7). From this viewpoint, the on-line critic network update rule of (5)–(7) is actually converging to a (local) minimum of the residual square of the equation of the principle of optimality in a statistical average sense.

VII. Discussion and Conclusion

This paper focuses on providing a systematic treatment of an NDP design paradigm, from architecture, to algorithm, analytical insights, and case studies. The results presented in this paper represent an effort toward generic and robust implementations of on-line NDP designs. Our design presented in the paper has, in principle, advanced in several aspects from the existing results. First, our proposed configuration is simpler than adaptive critic designs. Even though it is very similar to ADHDP, we have provided a mechanism that does not require a prediction model. The ADHDP design in adaptive critics either ignores the predictive model that results in nontrivial training errors or includes an additional prediction network that results in additional complexities in learning. Also, our design is robust in the sense that it is insensitive to parameters such as initial weights in the action and/or critic network, the values for the ultimate objective, U_c, etc. Key learning parameters are listed clearly in the paper for reproduction of our NDP design. Second, our NDP design has been tested on cases and has shown robustness in different tasks. The triple-link inverted pendulum balancing is a difficult nonlinear control problem in many ways. Our results measured by tightness of the vertical angles to the upright position is much improved over the traditional nonlinear control system designs used in [7]. When compared to the original RL designs by Barto [1], our on-line learning mechanism is more robust when subject to various noise, faster in learning to perform the task, and requires less number of trials to learn the task. The designs in [15] generally require more free parameters than our NDP design. Plus, our reinforcement signal is simpler than that in [15]. However our simulation results have demonstrated very robust performance by the proposed NDP configuration. Third, the fundamental guideline for the on-line learning algorithms is the stochastic gradient. We therefore have an estimate of the asymptotic convergence property for our proposed on-line NDP designs under some conditions in statistical sense. This provides more insight to the on-line learning NDP designs.

Next, we would like to share some of our experience on ways to handle large scale problems or the issue of scalability in NDP design. We developed and tested several designs that included a self-ganizing map (SOM) prior to the action network. The SOM takes the system states as inputs and produces an organized or reduced dimension expression of these input states to be passed onto the action network. In our experiments, we have used standard SOM algorithm developed by Kohonen [9]. The SOM as a state classifier can compress state measurements into a smaller set of vectors represented by the weights of the SOM network. On the other hand, it introduces quantization error that degrades the overall system performance accordingly. In terms of learning efficiency with the added SOM component, on one hand it reduces the feature space that the action network is exposed to and therefore it contributes toward a reduction in learning complexity in the action network. But on the other hand, adding a new network such as SOM (using the standard Kohonen training) can introduce tremendous computation burden on the overall learning system.

To have a more quantitative understanding of the effect of SOM on the overall learning system performance, we performed several learning tasks. First, we studied the system performance on the single cart-pole problem. We added an SOM network right in front of the action network. Refer to Fig. 1, the input to the SOM is the state $X(t)$ and the weight vectors of the SOM become the inputs to the action network. We then performed the same set of experiments as those documented in Table II. When compared to the results in the table without the SOM element in the learning system, we have observed much degraded performance by adding the SOM element. Specifically, we saw a 7.1% decreases in the success rate through the seven different cases (with different noise type), and a 142% increase in the number of trials needed to learn to balance the single cart-pole. Apparently the quantization error and the increased learning burden from SOM are contributing to this performance degradation.

We then performed a similar set of experiments on the triple-link inverted pendulum. Again, significant performance degradation was observed. With a closer examination at the quantization error, we realized that it was as high as 0.2 rad which corresponds to about 11°! It is soon realized that this high quantization error was due to the fast dynamics inherent in the triple-link inverted pendulum, which results in significant variances in the derivatives of the state measurements. The imbalance between the original state measurements (such as the angles) and the derivatives of the angles (the angular velocity) has created high demand in SOM resolution, or if not, significant quantization errors will result. To circumvent this, we divided the action network inputs into two sections. First, an SOM is employed to compress the state measurements, x, θ_1, θ_2, and θ_3, into a finite set of four-dimensional weight vectors, which forms one set of the inputs to the action network. The other set of inputs to the action network comes directly from the derivatives of the state measurements, namely, $\dot{x}$, $\dot{\theta}_1$, $\dot{\theta}_2$, $\dot{\theta}_3$. With such an implementation, the quantization error from the SOM is only about 0.04 rad (or, equivalently, 2°) on average. The overall learning performance was improved also. Specifically, we only see 3.8% decrease in success rate through the seven cases when compared to those in Table VI, and only 1.6% increase in the number of trials needed to learn to balance the triple-link inverted pendulum.

Note that, we believe there is still room for improving the quantization error and the learning speed inherent with the SOM network. One interesting observation from our experiments with SOM is that the learning speed for a controller with the SOM is not that much slower than the one without the SOM, especially in a complex task. The effect of reducing learning complexity has surfaced by using SOM as a compressor. It is quite convincing from our experiments on the single cart-pole that the SOM has added a tremendous computation overhead for the overall learning speed, especially for a relatively simple and low-dimensional problem. However, when one deals with a more complex system with more state variables, although the SOM still introduces computation overhead, its advantage of reducing the number of training patterns for the action network becomes apparent, which as a matter of fact may have reduced the overall learning complexity.

As discussed earlier, a good learning controller should be the one that learns to perform a task quickly, and also, learns almost all the time with a high percentage of success rate. Another factor that may not be explicitly present in the reinforcement signal is the degree of meeting the performance requirement. In all case studies, however, it is quite intriguing to see that the learning controllers are not only trying to balance the poles, but also trying to maintain the poles as centered as possible.

In summary, our results are very encouraging toward designing truly automatic and adaptive learning systems. But this paper only represents a start of this highly challenging task. Many more issues are still open such as how to characterize the overall learning process, instead of isolated and asymptotic guidelines, and how to interpret the controller output and system performance in a more systematic manner. As foreseeable goals, for one thing, the learning speed of the SOM can be improved potentially, and the overall system design should be tested on many more complex systems.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments. They also would like to thank L. Yang for carrying out the simulation study of Case II.

REFERENCES

[1] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuron like adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst., Man, Cybern.*, vol. 13, pp. 834–847, 1983.

[2] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*. Princeton, NJ: Princeton Univ. Press, 1962.

[3] D. P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[4] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, MA: Athena, 1996.

[5] T. Borgers and R. Sarin, "Learning through reinforcement and replicator dynamics," *J. Economic Theory*, vol. 77, no. 1, pp. 1–17, 1997.

[6] J. Dalton and S. N. Balakrishnan, "A neighboring optimal adaptive critic for missile guidance," *Math. Comput. Modeling*, vol. 23, no. 1, pp. 175–188, 1996.

[7] K. D. Eltohamy and C.-Y. Kuo, "Nonlinear optimal control of a triple link inverted pendulum with single control input," *Int. J. Contr.*, vol. 69, no. 2, pp. 239–256, 1998.

[8] D. Kirk, *Optimal Control Theory: An Introduction*. Englewood Cliffs, NJ: Prentice-Hall, 1970.

[9] T. Kohonen, *Self-Organizing Map*. Heidelberg, Germany: Springer-Verlag, 1995.

[10] H. J. Kushner and G. G. Yin, *Stochastic Approximation Algorithms and Applications*. New York: Springer-Verlag, 1997.

[11] R. E. Larson, "A survey of dynamic programming computational procedures," *IEEE Trans. Automat. Contr.*, vol. AC-12, pp. 767–774, 1967.

[12] D. V. Prokhorov, R. A. Santiago, and D. C. Wunsch II, "Adaptive critic designs: A case study for neuro-control," *Neural Networks*, vol. 8, pp. 1367–1372, 1995.

[13] D. V. Prokhorov and D. C. Wunsch II, "Adaptive critic designs," *IEEE Trans. Neural Networks*, vol. 8, pp. 997–1007, Sept. 1997.

[14] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, vol. 22, pp. 400–407, 1951.

[15] "COINS Tech. Rep.," Univ. Mass., Amherst, 96–88, Dec. 1996.

[16] R. S. Sutton, "Learning to predict by the methods of temporal difference," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[17] G. Tesauro, "Neurogammon: A neural-network backgammon program," in *Proc. Int. Joint Conf. Neural Networks*, San Diego, CA, 1990, pp. 33–40.

[18] ——, "Practical issues in temporal difference learning," *Machine Learning*, vol. 8, pp. 257–277, 1992.

[19] ——, "TD-Gammon, a self-teaching backgammon program achieves master-level play," *Neural Comput.*, vol. 6, 1994.

[20] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Trans. Automat. Contr.*, vol. 42, pp. 674–690, May 1997.

[21] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 257–277, 1992.

[22] P. Werbos, "Advanced forecasting methods for global crisis warning and models of intelligence," *General System Yearbook*, vol. 22, pp. 25–38, 1977.

[23] ——, "A menu of design for reinforcement learning over time," in *Neural Networks for Control*, W. T. Miller III, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT Press, 1990, ch. 3.

[24] ——, "Neuro-control and supervised learning: An overview and valuation," in *Handbook of Intelligent Control*, D. White and D. Sofge, Eds. New York: Van Nostrand, 1992.

[25] ——, "Approximate dynamic programming for real-time control and neural modeling," in *Handbook of Intelligent Control*, D. White and D. Sofge, Eds. New York: Van Nostrand, 1992.

[26] ——, “Tutorial on neurocontrol, control theory and related techniques: from back propagation to brain-like intelligent systems,” in *Proc. 12th Int. Conf. Math Comput Modeling Sci. Comput.*, 1999, www.iamcm.org/pwerbos/.

Jennie Si (S’90–M’91–SM’99) received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 1985 and 1988, respectively, and the Ph.D. degree from University of Notre Dame, Notre Dame, IN, in 1991.

She has been with Arizona State University, Tempe, since 1991 and is now Professor in the Department of Electrical Engineering. Her current research interest includes theory and application of artificial neural learning systems, specifically learning algorithms for statistical signal modeling and data analysis. The objective is to provide adaptive human–machine interface to solve large-scale signal and data analysis problems. Applications include semiconductor process optimization and yield management at both equipment and factory floor levels; spatial-temporal biological signal modeling in motor cortical and auditory systems, 2-D visual information processing through region of interest and robust feature selection; mining and interpretation of large data sets.

Dr. Si is a recipient of the 1995 NSF/White House Presidential Faculty Fellow Award and a recipient of the Motorola Excellence Award in 1995. She was Associate Editor of the IEEE TRANSACTIONS ON AUTOMATIC CONTROL in 1998 and 1999, has been an Associate Editor of the IEEE TRANSACTIONS ON SEMICONDUCTOR MANUFACTURING since 1998.

Yu-Tsung Wang (M’00) was born in 1971. He received the Master of Science degree in engineering and the Ph.D. degree in electrical engineering from Arizona State University (ASU), Tempe, in 1994 and 1999, respectively.

He was a Research Assistant in the Department of Electrical Engineering at ASU from 1995 to 1999. The main research was in the field of dynamic programming using neural network structures and mechanisms. In 1999, he joined Scientific Monitoring, Inc., as a Senior Engineer. His major work is in data trending algorithm, data analysis of aircraft engines, and software development. He is also involved in developing a turbine engine health monitoring system for the United States Air Force.

Helicopter Tracking Control Using Direct Neuro-Dynamic Programming [1]

Russell Enns and Jennie Si
Department of Electrical Engineering
Arizona State University
Tempe, AZ 85287-7606

Abstract

This paper advances a newly introduced neural learning control mechanism for helicopter flight control design. Based on direct neural dynamic programming (DNDP), the control system is tailored to learn to maneuver a helicopter in addition to its trimming and stabilization capabilities presented in earlier works. The paper consists of a comprehensive treatise of DNDP and extensive simulation studies of DNDP designs for controlling an Apache helicopter. Design robustness is addressed by performing simulations under various disturbance conditions. All designs are tested using FLYRT, a sophisticated industry-scale non-linear validated model of the Apache helicopter. Though illustrated for helicopters, our DNDP control system framework should be applicable for general purpose tracking control.

1 Introduction

While neural networks have been proposed for a variety of control systems for well over the last decade, their application has been either limited to low dimensional plants with only one or two controls or has been limited to higher dimensional systems that can be suitably decoupled into simpler subsystems [1, 2, 3]. A number of researchers have applied neural networks to the more specific domain of aircraft flight control but they too suffer from the same limitations, as well as others [4, 5, 6]. For instance, many of the flight control studies are limited to either single axis (longitudinal or lateral) control and are often for a linear model. In other cases, such as [4], the neural networks usage is limited to augmenting some non-neural network baseline control system.

DNDP is a highly promising class of neural network based techniques for decision and control designs in realistic nonlinear and dynamic systems with multiple inputs and multiple outputs in a noisy environment. DNDP is able to control more realistic higher dimensional systems such as helicopters by providing an approximate solution to optimal control problems that are often solved by dynamic programming. Previous work has shown the potential of a DNDP based control system for trimming and stabilizing a helicopter [7, 8]. This paper develops a more sophisticated DNDP based control system that is based on these earlier works and is able to perform maneuver tracking capability.

Results are provided to show the control system's ability to perform a variety of aircraft maneuvers in various operating conditions. More specifically, statistical results showing DNDP's ability to learn acceleration maneuvers from hover to 50 ft/s at various accelerations, up to the aircraft's upper limit of 0.25 g ($8 ft/s^2$), are provided. Results are also shown for deceleration maneuvers from 100 ft/s to 50 ft/s at various decelerations. Simulations are performed in both clear air and in the presence of turbulence and step gusts. Plots of the typical and statistical tracking performance is also shown for two representative cases. Unlike many results which are based on linearized models and corresponding assumptions, our DNDP designs and simulations are conducted using the FLYRT model. We thus are dealing with a very realistic system with nonlinearities, actuator dynamics, etc..

2 Direct Neuro-Dynamic Programming for MIMO Rotorcraft Control

The objective of a neuro-dynamic programming controller is to optimize a desired performance measure by learning to create appropriate control actions through interaction with the environment. The controller is designed to learn to perform better over time using only sampled measurements and with no prior knowledge about the system. The comprehensive development of the DNDP and a historical reflection of this simulation-based dynamic programming technique is expounded on in [9].

[1] Supported in part by NSF under grant ECS-9553202 and by EPRI-DOD under grant WO8333-01.

Figure 1 outlines the neuro-dynamic programming control structure. It consists of a structured cascaded series of networks forming the action network, a critic network, and a trim network. The action network provide the required controls for a given system state. The critic network is used to provide an estimate/approximation of the cost function if an explicit cost function does not exist. The trim network provides nominal trim control positions as a function of the system's desired operating condition. Implementation details are provided in [7].

2.1 The Critic Network

The critic network approximates a cost function should an explicit cost function not be convenient or possible to represent. For example, the network output $J(t)$ can approximate a cost function such as the discounted total reward-to-go,

$$R(t) = r(t+1) + \alpha r(t+2) + \alpha^2 r(t+3) + \cdots \tag{1}$$

where $R(t)$ is the future accumulative reward-to-go value at time t, α is a discount factor for the infinite-horizon problem $(0 < \alpha < 1)$, and $r(t+1)$ is the external reinforcement value at time $t+1$.

Typically DNDP has been applied to systems where explicit feedback is not available at each time step. In such cases the reinforcement signal, $r(t)$, takes a simple binary form with $r(t) = 0$ when the final event is successful (an objective is met), or $r(t) = -1$ if the final event is a failure (the objective is not met). For the tracking problem at hand, we defined a more informative quadratic reinforcement signal

$$r(t) = -\sum_{i=1}^{n} \left(\frac{(x_i - x_{i,d})}{x_{i,max}} \right)^2 \tag{2}$$

where x_i is the i-th state of the state vector $\mathbf{x}$, $x_{i,d}$ is the desired reference state and $x_{i,max}$ is the nominal maximum state value.

The critic network can be implemented with a standard multi-layer feedforward neural network. The network can be linear or having a nonlinear sigmoid function to fan out outputs depending on the complexity of the problem. We typically use a two layer weight neural network with sigmoid functions for the network nonlinearities. The critic network output, shown in Figure 1, is simply the scalar J.

The critic network is trained to minimize the following objective function

$$E_c(t) = \frac{1}{2} e_c^2(t) \tag{3}$$

where

$$e_c(t) = \alpha J(t) - [J(t-1) - r(t)]. \tag{4}$$

The weights of the critic network are updated according to a gradient descent algorithm,

$$\Delta w_c(t) \quad = \quad \beta_c(t) \left[-\frac{\partial E_c(t)}{\partial J(t)} \frac{\partial J(t)}{\partial w_c(t)} \right], \tag{5}$$

where $\beta_c(t)$ is the learning rate of the critic network at time t, which usually decreases with time to a small value,

$$\frac{\partial E_c(t)}{\partial J(t)} = \frac{\partial E_c(t)}{\partial e_c(t)} \frac{\partial e_c(t)}{\partial J(t)} = \alpha e_c(t) \tag{6}$$

and $\partial J(t)/\partial w_c(t)$ is a function of the critic network's structure.

2.2 The Action Network

The action network generates the desired plant control given either measurements of the plant states or plant features from the feature extractor. As with the critic network, the action network can be implemented with a standard multi-layer linear or nonlinear feedforward neural network. For the action network the number of network outputs equals the control space dimension.

The principle in adapting the action network is to back-propagate the error between the desired ultimate objective, denoted by U_c, and the cost function $R(t)$. Either the actual cost function $R(t)$, or an approximation to it $J(t)$, is used depending on whether an explicit cost function or a critic network is available. In the latter case back-propagation is done through the critic network. For notational simplicity $J(t)$ represents either the actual or approximate cost function, depending on which is being used, for the remainder of the paper.

The weight updating in the action network adjusts the action network's weights to minimize the following objective function,

$$E_a(t) = \frac{1}{2} e_a^2(t), \tag{7}$$

where

$$e_a(t) = J(t) - U_c(t). \tag{8}$$

The weights in the action network are then updated according to

$$\Delta w_a(t) \quad = \quad \beta_a(t) \left[-\frac{\partial E_a(t)}{\partial w_a(t)} \right]. \tag{9}$$

In this paper we introduce the concept of a structured ANN. The explicit structure embedded in this ANN, lacking in earlier DNDP designs, allows the network to more easily learn and take advantage of the physical relationships and dependencies of the system. Such structure is similar to classic controllers for helicopters, providing

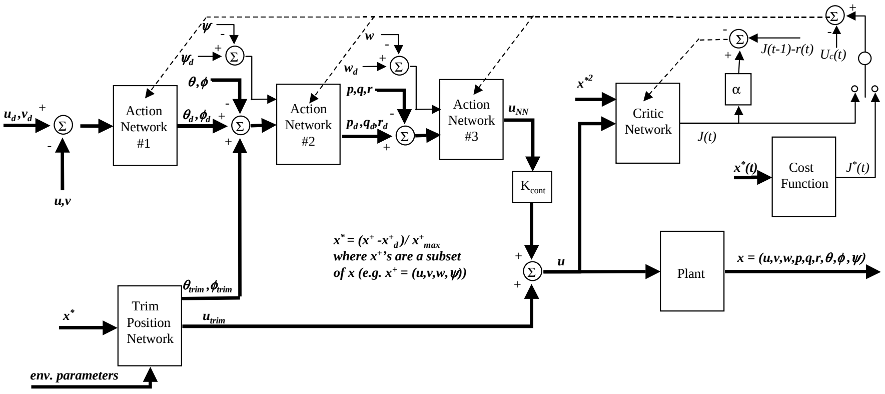


Figure 1: Neuro-dynamic Programming Based Controller.

for inner loop body rate control, attitude control and outer loop velocity control. In this way, the explicit relationships between body angular rates, attitudes and translational velocities are taken advantage of. The potential advantage of the structured ANN over classic design methodologies is that it permits full cross-axes control coupling that many SISO PID controller designs do not. The disadvantage of the structured ANN is that it introduces a level of human knowledge/expertise to it.

It can be shown that a classic proportional controller can be equated to one instance (one set of weights) of our network if the network's non-linearities are removed (or approximated by a linearity about the network's operating point). Such a relationship between the two designs can be used to provide a good first guess of the action network's weights should one want to apply this "expert" knowledge to the learning system.

2.3 The Trim Network

The trim network schedules the nominal control trim position as a function of aircraft state and environmental/flight parameters (such as aircraft weight, air density etc.) Having DNDP determine the control trim position is key to successfully using DNDP to controlling general systems.

Previous DNDP control designs were successful because the systems that were tested (e.g. the inverted pendulum) had a zero trim requirement [9]. Many flight control papers have assumed linear models in which case there is also a zero trim requirement since the linear model is linearized about a trim condition. However, in general trim requirements cannot be ignored for non-linear models, including those for aircraft. A DNDP-based method for determining the trim position for each flight condition has been developed and is presented in [7].

3 The Helicopter Model

The controller is tested using a helicopter model, run at 50 Hz, that consists of three parts: an actuator model, an actuator to blade geometry model, and FLYRT. The inputs to the model are the three main rotor actuator positions and the tail rotor actuator position. The outputs from the model are numerous; for flight control purposes they are limited to the aircraft's translational (u, v, w) and rotational (p, q, r) velocities and the aircraft's orientation (θ, ϕ, ψ) for a total of 9 states.

At the heart of the helicopter model is FLYRT, a sophisticated non-linear flight simulation model of the Apache helicopter developed by Boeing over the past two decades [10]. FLYRT models all the forces and moments acting on the helicopter. The rotor is modeled using a blade element model. FLYRT dynamically couples the six degree of freedom rigid body of the helicopter to the main rotor through Euler equations. The drive train is represented as a single degree of freedom model and is coupled to the main rotor, tail rotor and engine. The engine is modeled in sufficient detail to cover performance over all phases of flight, including ground modes. The landing gear is modeled as three independent units interfacing with a rigid airframe. Quaternions are used during state integration to accommodate large attitude maneuvers.

In addition to FLYRT, our model also consists of actuator models as well as a model of the mechanical geometry between the actuators and the helicopter blades. Each actuator is modeled as a first order lag with time constant $\tau = 0.03$, reflective of a typical actuator. Actuator rate and position limits are also modeled. The mechanical interface between the actuators and the helicopter blades is also modeled.

The operating conditions for which our simulation studies are performed are shown in Table 1. The center of gravity (C.G.) is listed in the standard Apache FS/WL/BL coordinate frame [10].

Weight	16324 lb
C.G. - FS/BL/WL	201.6 in, 0.2 in, 144.3 in
Temperature	59^o F
Altitude	1770 ft

Table 1: Helicopter Operating Conditions.

4 Results

This section presents results showing the controller's performance in controlling the Apache helicopter for a range of maneuvers. Characteristic to prior NDP research, the controller's performance is summarized statistically in tables. Fourteen maneuvers are considered: 7 accelerations from hover to 50 ft/s at various accelerations and 7 decelerations from 100 ft/s to 50 ft/s at various decelerations. Each maneuver is tested in three wind conditions: case **A**) no wind, case **B**) 10 ft/s step gust for 5 seconds, and case **C**) turbulence simulated using a Dryden model with a spatial turbulence intensity of $\sigma = 5ft/s$ and a turbulence scale length of $L_W = 1750ft$. In addition to the tabular statistics provided, both statistical and typical time history plots of the aircraft states are provided for two cases, a hover to 50 ft/s at $4ft/s^2$ maneuver with turbulence and hover to 50 ft/s at $-5ft/s^2$ maneuver with a step gust.

The objective is to train the controller to perform the specified maneuver regardless of the operating conditions or the vehicle's initial conditions. The states of interest are the aircraft's translational (u, v, w) and rotational (p, q, r) velocities and the aircraft's Euler angles pitch (θ), roll (ϕ) and yaw (ψ). Failure criteria are used to bound each state's allowed error. The allowed errors, shown in Table 2, are initially large and decrease as a function of time to an acceptable minimum.

These failure criteria were chosen judiciously but no claims are made to their optimality. The results show that these criteria create a control system that can control the helicopter both in nominal conditions and when subjected to disturbances. Heuristic failure criteria is one of the advantages of DNDP if one does not have an accurate account of the performance measure. This is also one characteristic of the DNDP design that differs from other neural control designs. The critic network plays the role of working out a more precise account of the performance measure for credit/blame assignment derived from the heuristic criteria. If the networks have converged, an explicitly desired state has been achieved which is reflected in the U_c term in the DNDP structure.

Aircraft State	Initial Allowed Error	Final Allowed Error	Error Rate
u, v, w	$20ft/s$	$4ft/s$	$-0.8ft/s^2$
p, q, r	$30^o/s$	$6^o/s$	$-1.2^o/s^2$
θ, ϕ, ψ	30^o	6^o	$-1.2^o/s$

Table 2: Failure Criterion For Helicopter Stabilization.

The statistical success of the controller's ability to learn to control the helicopter is evaluated for the five flight conditions. For each flight condition 100 runs were performed to evaluate the controller's performance, where for each run the neural networks' initial weights were set randomly. Each run consists of up to 5000 attempts (trials) to learn to successfully control the system. An attempt is deemed successful if the helicopter stays within the failure criteria bounds described in Table 2 for the entire flight duration (50 seconds). If the controller successfully controls the helicopter within 500 trials the run is considered successful, if not, the run is considered a failure.

Table 3 statistically summarizes the controller's learning ability to perform a hover to 50 ft/s maneuver at a number of different accelerations. Results for 100 ft/s to 50 ft/s decelerations at various deceleration rates are not provided due to lack of space. The results however are similar to those for acceleration statistically. In both cases results are provided for the three wind conditions cited above. The success percentage reflects the percentage of runs for which the system successfully learns to control the helicopter. The average number of trials is the average number of trials that it takes the learning system to learn to control the helicopter. The learning variance reflects the statistical variance in the number of trials required to learn to control the system.

Figure 2 shows both the statistical average state error and error deviation and a typical plot of the controller performance for a hover to $50ft/s$ maneuver at an aggressive $5ft/s^2$ acceleration in the presence of turbulence. Figure 3 shows both the statistical average state error and error deviation and a typical plot of the controller performance for a $100ft/s$ to $50ft/s$ maneuver at $-4ft/s^2$ acceleration in the presence of a step gust. Plots for the other maneuvers are similar.

The neural network parameters used during training are provided in Table 5. The learning rates, β, for the action network (ANN) and critic network (CNN) are potentially

Condition	Acceleration (ft/s^2)	2	3	4	5	6	7	8
Case A	Success Percentage	94%	62%	67%	65%	66%	66%	74%
	Average No. of Trials	1600	2019	2115	1950	1983	2028	1870
	Learning Variance	214	339	324	307	293	306	252
Case B	Success Percentage	95%	98%	97%	85%	60%	56%	58%
	Average No. of Trials	642	824	1126	1843	1842	2145	2379
	Learning Variance	115	128	165	263	313	333	403
Case C	Success Percentage	96%	66%	76%	70%	50%	57%	53%
	Average No. of Trials	1367	1720	1770	1874	2173	1970	2419
	Learning Variance	191	280	255	275	381	337	400

Table 3: Learning Statistics for Hover to 50 ft/s Maneuver at Various Accelerations for Three Wind Conditions.

scheduled to decrease linearly with time (typically over a few seconds). In every time frame the weight equations are updated until either the error has sufficiently converged ($E < E_{tol}$) or N_{cyc} internal update cycles of the weights have occurred. N_h is the number of hidden nodes in the neural networks.

Parameter	Value
$\beta_a(t_0)$	0.02
$\beta_a(t_f)$	0.02
$\beta_c(t_0)$	0.1
$\beta_c(t_f)$	0.01
$N_{cyc,a}$	200
$N_{cyc,c}$	100
$E_{tol,a}$	0.005
$E_{tol,c}$	0.1
N_h	6
K_{cont}	2.5

Table 4: Neural Network Parameter Values.

These parameters were chosen based on experience but were not tuned to optimize the results. Tuning parameters is very difficult to do because of the large number of parameters to be tuned and the large number of maneuvers and operating conditions that can be tuned for. For example, the action network's learning rate can be tuned to improve learning performance depending on the maneuver performed. Lower learning rates improve the success for more aggressive maneuvers but decrease the learning ability (increase the number of trials) required to learn for less aggressive maneuvers. This is illustrated by Table 6 which shows the system's performance when the ANN's learning rate β_a is 0.2 versus 0.02 for both more and less aggressive accelerations.

As with previous DNDP designs, a large number of trials must occur to successfully learn to perform the maneuver. Further, the more aggressive the maneuver (the higher the acceleration), the more trials that are required. This is not surprising for a learning system that is learning from experience without any a priori system knowledge. The ramification is that this training is done off-line (i.e. not in a real helicopter), where failures can be afforded, until the controller is successfully trained. Once trained, the neural network weights are frozen and the controller structure shown in Figure 1 can be implemented in a helicopter. Limited authority on-line training can then be performed to improve system performance if desired.

β_a	Acceleration (ft/s^2)	2	6	8
0.02	Success Percentage	94%	66%	74%
	Average No. of Trials	1600	1983	1870
	Learning Variance	214	293	252
0.2	Success Percentage	100%	80%	20%
	Average No. of Trials	248	1708	2809
	Learning Variance	71	421	439

Table 5: Learning Statistics for Hover to 50 ft/s Maneuvers as a Function of Learning Rate.

The results also show that the DNDP controller is most often able to learn faster and more reliably in the presence of turbulence.

5 Conclusions

This paper has advanced neural-dynamic programming control research by extending an existing DNDP control structure to support control tracking, contrasting to earlier related works that have been limited to stabilization only. A sophisticated nonlinear validated model of the Apache helicopter was used to test the controller and its

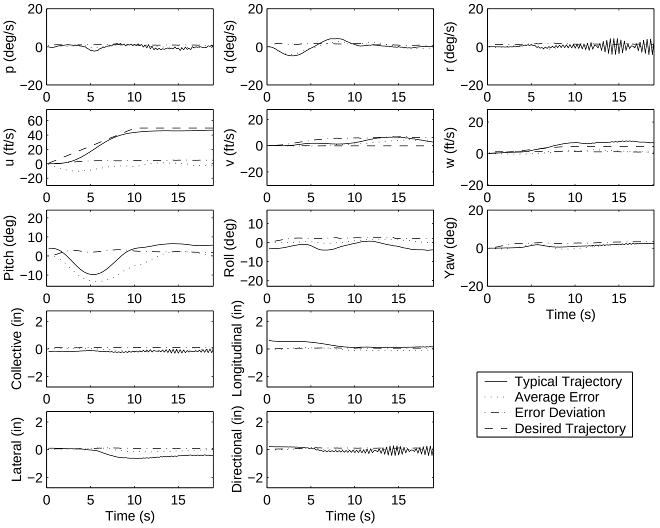


Figure 2: Statistical and Typical State and Control Trajectories for Hover to 50 ft/s Maneuver at $5ft/s^2$ Acceleration in Turbulence.

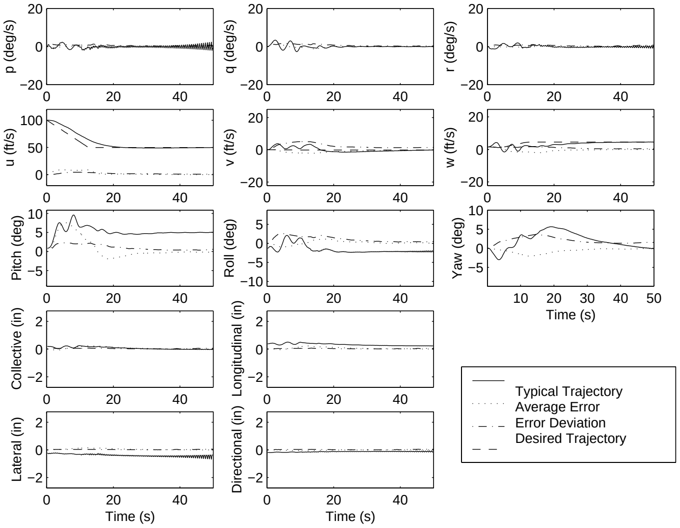


Figure 3: Statistical and Typical State and Control Trajectories for 100 ft/s to 50 ft/s Maneuver at $-4ft/s^2$ Acceleration in a Step Gust.

ability to learn to perform a number of maneuvers. Our research has shown the DNDP controller able to successfully control the Apache helicopter for a wide range of maneuvers and over a wide range of flight conditions, a few examples of which are presented in this paper. Thus it appears that DNDP is a viable candidate for controlling systems and is suited particularly well for model-free and complex multiaxes coupling control applications.

References

[1] Omidvar, O. and Elliot, D. (Eds.), *Neural Systems For Control*, San Diego: Academic Press, 1997.

[2] Gupta, M. and Rao D. (Eds.), *Neuro-Control Systems Theory and Applications*, New York: IEEE Press, 1994.

[3] Vemuri, V. (Ed.), *Artificial Neural Networks Concepts and Control Applications*, Los Alamitos: IEEE Computer Society Press, 1992.

[4] Kim, B. and Calise, A., "Nonlinear flight control usig neural networks", *AIAA Journal of Guidance, Control, and Dynamics*, vol. 20, no. 1, pp. 26-32, Jan.-Feb. 1997.

[5] Balakrishnan, S. and Biega, V., "Adaptive-critic-based neural networks for aircraft optimal control", *AIAA Journal of Guidance, Control, and Dynamics*, vol. 19, no. 4, pp. 731-739, Jul.-Aug. 1996.

[6] Ha, C., "Neural networks approach to AIAA control design challenge", *AIAA Journal of Guidance, Control, and Dynamics*, vol. 18, no. 4, pp. 731-739, Jul.-Aug. 1995.

[7] Enns, R. and Si, J., "Neuro-dynamic programming applied to helicopter flight control", *Proceedings of the AIAA Guidance, Navigation and Control Conference*, Denver, CO, August 15-17, 2000.

[8] Enns, R. and Si, J., “Helicopter flight control design using a learning control approach”, *Proceedings of the 2000 IEEE Conference on Decision and Control*, Sydney, Australia, December 2000, pp. 1754-1759.

[9] Si, J. and Wang, J., “On-line learning by association and reinforcement”, *IEEE Transactions on Neural Networks*, March 2001.

[10] Kumar, S., Harding, J., and Bass, S., *AH-64 Apache engineering simulation non-real time validation manual*, USAAVSCOM TR 90-A-010, October 1990.

New Directions in Hierarchical Reinforcement Learning: Concurrency,Multi-Agency, and Partial Observability

Sridhar Mahadevan, U. Mass.

Abstract

Hierarchical reinforcement learning is a general framework that uses temporal abstraction to simplify the problem of sequential decision-making. Actions are viewed as multi-step closed-loop programs, instead of unit time actions, which reduces the number of decision points. Thus far, work in this area has assumed that agents act alone, and are able to perceive the complete state of the environment.

In this talk, I present recent work of my group on extending the framework of ierarchical reinforcement learning to address three key challenges that are often present in many real-world settings: concurrency, multi-agency, and partial observability.

Concurrent action is an essential ingredient of many real-world problems, from driving on a freeway to making breakfast. Here, agents can execute multiple temporally extended actions in parallel, which interact in subtle ways. I present three termination conditions for composing sequences of parallel actions, and compare their performance analytically and empirically.

Concurrency forms a foundation for modeling cooperative multi-agency where several agents act in concert to solve an overall task. Using the results of our parallel termination analysis, I describe a hierarchically optimal multiagent reinforcement learning algorithm, and illustrate its effectiveness on a large factory autonomous guided vehicle (AGV) problem.

Agents can rarely sense the complete state of their environment, particularly in a multiagent setting. Joint state and joint actions are often hidden, and agents need to act based on information states, or probability distributions over the latent variables. I present a hierarchical partially observable Markov decision process model, which enables agents to construct information states at multiple spatiotemporal abstraction levels. I show how this multiresolution model can be used in robot navigation, allowing a robot to navigate in a large indoor environment without initial location information.

RELEVANT PAPERS: (available online at www.cs.umass.edu/~mahadeva/)

1. Khashayar Rohanimanesh and Sridhar Mahadevan, "Decision-Theoretic Planning with Concurrent Temporally Extended Actions" , Seventeenth Conference on Uncertainty in Artificial Intelligence , August 3-5, 2001

2. Mohammad Ghavamzadeh and Sridhar Mahadevan "Continuous-time Hierarchical Reinforcement Learning", Eighteenth International Conference on Machine Learning (ICML) , June 28-July 1, 2001, Williams College, Massachusetts

3. Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh "Hierarchical Multi-Agent Reinforcement Learning", Fifth International Conference on Autonomous Agents, Montreal, 2001.

4. Georgios Theocharous and Sridhar Mahadevan, "Approximate Planning with Hierarchical Partially Observable Markov Decision Processes for Robot Navigation" ,IEEE Conference on Robotics and Automation (ICRA) , Washington, D.C. May 2002.

5. Georgios Theocharous, Khashayar Rohanimanesh, and Sridhar Mahadevan "Learning Hierarchical Partially Observable Markov Decision Processes for Robot Navigation", IEEE Conference on Robotics and Automation, (ICRA), 2001, Seoul, South Korea.

Learning Hierarchical Partially Observable Markov Decision Process Models for Robot Navigation

Georgios Theocharous
theochar@cse.msu.edu

Khashayar Rohanimanesh
khash@cse.msu.edu

Sridhar Mahadevan
mahadeva@cse.msu.edu

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823

Abstract— **We propose and investigate a general framework for hierarchical modeling of partially observable environments, such as office buildings, using Hierarchical Hidden Markov Models (HHMMs). Our main goal is to explore hierarchical modeling as a basis for designing more efficient methods for model construction and useage. As a case study we focus on indoor robot navigation and show how this framework can be used to learn a hierarchy of models of the environment at different levels of spatial abstraction. We introduce the idea of model *reuse* that can be used to combine already learned models into a larger model. We describe an extension of the HHMM model to includes actions, which we call hierarchical POMDPs, and describe a modified hierarchical Baum-Welch algorithm to learn these models. We train different families of hierarchical models for a simulated and a real world corridor environment and compare them with the standard "flat" representation of the same environment. We show that the hierarchical POMDP approach, combined with model reuse, allows learning hierarchical models that fit the data better and train faster than flat models.**

I. Introduction

Recent work in artificial intelligence (AI) on sequential decision-making under uncertainty has adopted the framework of Partially Observable Markov Decision Processes (POMDP) [1], [2]. This framework allows modeling environments where the underlying states are "hidden", and only partially observable through noisy sensory observations and actions. POMDPs extend the well-known Hidden Markov Model (HMM) [3] to include actions and rewards. There are well-developed algorithms for both building and using POMDP models, such as the Baum-Welch procedure and dynamic programming. However, past work on POMDPs has been restricted to "flat" uniform scale models, and the model-learning and planning algorithms scale poorly with the size of the model. Several studies have shown that the HMM/POMDP framework can be used to program autonomous mobile robots to navigate in real office environments [1], [4], [2], but these systems use flat models. As the size of the environment grows, it becomes increasingly difficult to learn and use flat models, and it would be desirable to have a natural way of reusing previously learned sub-models.

In this paper, we address these limitations of earlier standard HMM/POMDP systems using the Hierarchical Hidden Markov Model (HHMM) framework [5]. Using the HHMM approach, a robot learns and uses a hierarchy of homogeneous representations of the environment in which each layer of the hierarchy maintains a probabilistic model of the environment defined at some resolution. The tree structure of HHMM models also provides a natural approach for rapid model learning by reusing previously learned sub-models. We extend HHMM models to include primitive actions, and show a procedure for converting hierarchical models into flat POMDPs.

Intuitively, hierarchical modeling should allow us to learn the environment in a modular fashion and therefore be able to learn faster, and even be able to discover relationships at abstract levels that would not be trivial in flat representations. Fortunately, many natural environments can be viewed at different levels of spatial abstraction. For example, an office environment modeled at a coarse level would consist of nodes for corridors, intersections and dead-ends; at a finer level, each model element would represent a fixed-length region of a corridor.

In this paper we demonstrate through experiments that the HHMM framework extended to the hierarchical POMDP case appears to have some compelling advantages as a basis for designing scalable spatial learning algorithms. We will provide empirical results from learning in both real-world and simulated robot navigation to illustrate the potential of the hierarchical POMDP framework.

II. Hierarchical Hidden Markov Models

The Hierarchical Hidden Markov Model (HHMM) [5] generalizes the standard hidden Markov model (HMM) [3], by allowing hidden states to represent stochastic processes themselves. An HHMM is visualized as a tree structure in which there are three types of states, production states (leaves of the tree) which emit observations, and internal states which are (unobservable) hidden states that represent entire stochastic processes. Each production state is associated with an observation vector which maintains distribution functions for each observation defined for the model. Each internal state is associated with a horizontal transition matrix, and a vertical transition vector. The horizontal transition matrix of an internal state defines the transition probabilities among its children. The vertical transition vectors define the probability of an internal state to activate any of its children. Each internal state is also associated with a child called an *end-state* which returns control to its parent. The end-states do not produce observations and cannot be activated through a vertical transition from their parent. The HHMM is formally defined as a 5 tuple $\langle S, T, \Pi, Z, O\rangle$:

- S denotes the set of states. The functions $p(s)$ denotes the parent of state s. The function $c(s, j)$ returns the j^{th} child of state s. The end-state child of an abstract state s is denoted by e^s. The set of children of a state s is denoted by C^s and the number of children by $|C^s|$. There are three types of states.
  – *Production states*
  – *Abstract states*
  – *End-states*
- $T^s : C^s \times C^s \rightarrow (0, 1)$ denotes the horizontal transition functions, defined separately for each abstract state. A horizontal transition function maps each child state of s into a probability distribution over the children states of s. We write $T^s(c(s, i), c(s, j))$ to denote the horizontal transition probability from the i^{th} to the j^{th} child of state s. As an example, in Figure 1, $T^{s4}(s7, s8) = 0.8$.
- $\Pi^s : \{C^s - e^s\} \rightarrow (0, 1)$ denotes the vertical transition function for each abstract state s. This function defines the initial distribution over the children states of state s, except from the end-state child e^s. For example, in Figure 1, $\Pi^{s4}(s6) = 0.5$.
- Z denotes the set of observations.
- $O : S^{product} \rightarrow (0, 1)$ denotes a function that maps every product state to a distribution over the observation set. We write $O(s, z)$ for the probability of observing z in state s.

Figure 1 shows a graphical representation of an example HHMM. The HHMM produces observations as follows:

1. If the current node is the root, then it chooses to activate one of its children according to the vertical transition vector from the root to its children.
2. If the child activated is a product state, it produces an observation according to an observation probability output vector. It then transitions to another state within the same level. If the state reached after the transition is the end-state, then control is returned to the parent of the end-state.
3. If the child is an abstract state then it chooses to activate one of its children. The abstract state waits until control is returned to it from its child end-state. Then it transitions to another state within the same level. If the resulting transition is to the end-state then control is returned to the parent of the abstract state.

Fine et al. [5] describe a hierarchical Baum-Welch algorithm that is able to re-estimate the model parameters λ (including transitions matrices, vertical vectors, and observation vectors) of an HHMM given observation sequences $M = z_1, z_2, ..., z_T$. We have extended this algorithm to be able to learn the parameters of an HHMM that includes actions, as we describe next.

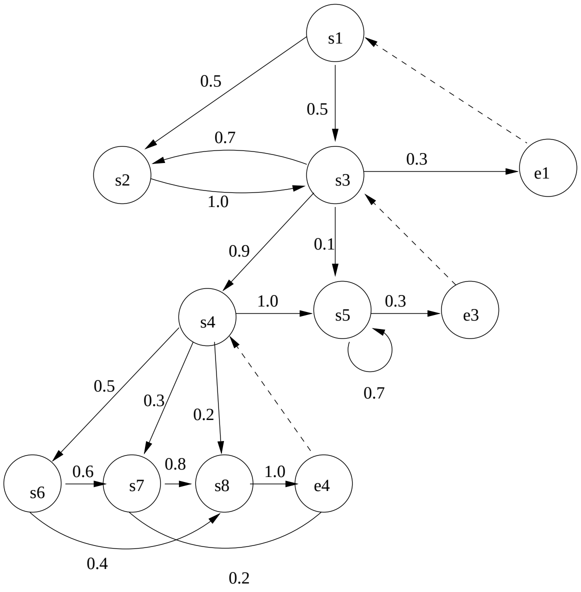


Fig. 1. An example hierarchical HMM. Only leaf (production) states (s2, s5, s6, s7, and s8) have associated observations.

III. Hierarchical Partially Observable Markov Decision Process Models

A. Model definition

Robot navigation is an example of a planning problem that requires extending the HHMM model to include both primitive and abstract actions, as well as reward functions for specifying goals. We call this extended model *hierarchical POMDP* and we formally define it as follows:

- S denotes the set of states which are exactly the same as an HHMM.
- A denotes the set of primitive actions. The primitive actions initiate and terminate in product states. The product states represent the lowest resolution of a physical environment and therefore every action has to start in some product state and end in some other product state. For example, in an indoor robot navigation environment, if abstract states are corridors and product states are 2 meter locations in the corridors, then every primitive action such as "go-forward" will always take the robot from one product state to another product state. However, the transition probability from one product state $s1$ to some other product state $s2$ is not simply a lookup operation into a global transition matrix as in flat POMDPs. In the hierarchical POMDP model we may have to look at more than a single horizontal transition matrix and even multiple vertical vectors. An example of this calculation is shown in Figure 2.
- $T^{s,a} : C^s \times C^s \rightarrow (0, 1)$ denotes the horizontal transition functions, which are defined separately for each abstract state s and action a. A horizontal transition function maps each child state of s and action pair into a probability distribution over the children states of s.
- $\Pi^s : \{C^s - e^s\} \rightarrow (0, 1)$ denotes the vertical transition function for each abstract state s. This function defines the initial distribution over the children states of state s.

- Z denote the set of observations as before.
- $O : S^{product} \rightarrow (0,1)$ is a function that maps every product state to a distribution over the observation set.
- $R : S^{product} \times A \rightarrow \Re$ denotes an immediate reward function defined on the product states.

Figure 2 shows an example hierarchical POMDP. A hierarchical Baum-Welch algorithm for the hierarchical POMDP can be defined by extending the hierarchical Baum-Welch algorithm for HHMMs. First we define the α variable as shown in Equation 1

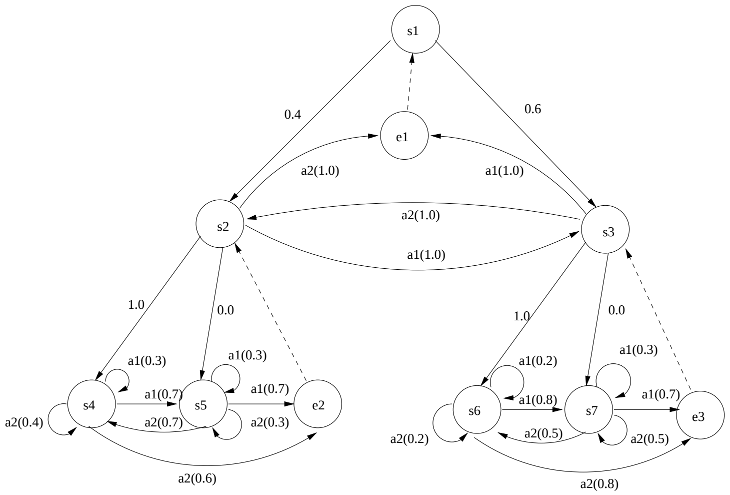


Fig. 2. An example hierarchical POMDP with two primitive actions, $a1$ and $a2$. To calculate the transition from state $s4$ to state $s4$ under action $a2$ we have to consider all non-zero probability paths. One path is $(s4, s4)$ and the other path is $(s4, e2, s2, e1, s1, s2, s4)$. Thus while $T^{s_2,a_2}(s_4, s_4) = 0.4$, the probability of transitioning from s_4 to s_4 under action $a_2 = 0.4 + 0.6 \times 1.0 \times 0.4 \times 1.0 = 0.64$.

$$\alpha(t, t+k, s) = P(z_t, ...z_{t+k}, s\ finished\ at\ t+k\ | a_t, ...a_{t+k-1}, p(s)\ started\ at\ t, \lambda) \tag{1}$$

The α variables gives us the probability of the observation sequence $z_t...z_{t+k}$ and that state s finished at time $t+k$ given that actions $a_t, ...a_{t+k-1}$ were taken and the parent of s, $p(s)$ was started at time t. We say that a product state finishes at time t after the production of observation z_t and an abstract state finishes when control is returned to it from its child end-state after one of its children produces its last observation z_t. We also say that a product state s, started at time t, if at time t it produced observation z_t. An abstract state s starts at time t if at time t one of its children produced observation z_t but observation z_{t-1} was generated before s was activated by its parent or any other horizontal transition.

Note that using the α variable we can calculate the probability of an observation sequence given the model as shown in Equation 2.

$$P(M|\lambda) = \sum_{s=1|c(root,s)\neq e^{root}}^{|C^{root}|} \alpha(1, T, c(root, s)) \tag{2}$$

The backward variable β, defined in Equation 3, denotes the probability that a state s was entered at time t and that the observations $z_t...z_{t+k}$ were produced by the parent of s, which is $p(s)$ and actions $a_t, ...a_{t+k}$ were taken and p(s) terminated at time $t+k$.

$$\beta(t, t+k, s) = P(z_t, ...z_{t+k}|a_t, ...a_{t+k},\ s\ started\ at\ t,\ p(s)\ generated\ z_t...z_{t+k}\ and\ finished\ at\ t+k, \lambda) \tag{3}$$

The next important variable is ξ, defined in Equation 4, which denotes the probability of making a horizontal transition from s to s' at time t.

$$\xi(t, s, s') = P(s\ finished\ at\ time\ t,\ s'\ started\ at\ time\ t+1\ |a_1, ...a_T, z_1, ...z_T, \lambda) \tag{4}$$

Another important variable is $\chi(t, s)$, shown in Equation 5, which defines the probability that state s was activated by its parent $p(s)$ at time t given the action and observation sequence

$$\chi(t, s) = P(p(s)\ started\ at\ time\ t, s\ started\ at\ time\ t| a_1, ...a_T, z_1, ...z_T, \lambda) \tag{5}$$

Based on these variables we can re-estimate the model parameters. The vertical vectors are re-estimated in Equations 6 and 7.

$$\Pi^s(n) = \chi(0, n),\ if\ s = root,\ n \in C^s \tag{6}$$

$$\Pi^s(n) = \frac{\sum_{t=1}^{T} \chi(t, n)}{\sum_{i=1|c(s,i)\neq e^s}^{|C^s|} \sum_{t=1}^{T} \chi(t, c(s, i))},\ if\ s \neq root,\ n \in C^s \tag{7}$$

The horizontal transition matrices are re-estimated in Equation 8. The re-estimation calculates the average number of times the process went from state s to state s' over the number of times the process exited state s under action a.

$$T^{p(s),a}(s, s') = \frac{\sum_{t=1|a_t=a}^{T} \xi(t, s, s')}{\sum_{t=1|a_t=a}^{T} \sum_{i=1}^{|C^{p(s)}|} \xi(t, s, c(p(s), i))} \tag{8}$$

The observation vectors are re-estimated in Equation 9. The re-estimation of the observation model calculates the average number of times the process was in state s and perceived observation z over the number of times the process was in state s.

$$O(s,z) = \frac{\sum_{t=1,z_t=z}^{T} \chi(t,s) + \sum_{t=2,z_t=z} \gamma_{in}(t,s)}{\sum_{t=1}^{T} \chi(t,s) + \sum_{t=2}^{T} \gamma_{in}(t,s)}$$

$$where\ \gamma_{in}(t,s) = \sum_{i=1|c(p(s),i)\neq e^{p(s)}}^{|C^{p(s)}|} \xi(t-1, c(p(s),i), s) \tag{9}$$

B. Planning using Hierarchical POMDPs

Solving a POMDP means that we have to find a mapping from each "belief state" (probability distribution over states) to actions that will achieve the best long term sum of rewards [6]. The belief state is a sufficient statistic that summarizes the past history of observations and actions. There is an efficient Bayesian update procedure that can be used to calculate the belief state probability distribution over all the product states given a sequence of observations and actions. Unfortunately the number of belief states is infinite and exact solutions to large POMDPs are computationally infeasible. Fortunately, many heuristics solutions such as the most likely state heuristic (MLS) and the Q-MDP method are known to provide satisfactory approximate solutions for robot navigation [7]. We have extended these approximate methods to hierarchical POMDPs, by implementing hierarchical versions of these methods (e.g. for the MLS procedure, we compute the most likely abstract state, and then recursively the most likely product state).

Any hierarchical POMDP model with actions can be converted into an equivalent flat POMDP. The states of the equivalent flat POMDP are the product states of the hierarchical POMDP and are associated with a global transition matrix that is calculated from the vertical and horizontal transition matrices of the hierarchical POMDP. To construct this global transition matrix of the equivalent flat POMDP, for every pair of states (s_1, s_2), we need to sum up the probabilities of all the paths that will transition the system from $s1$ to $s2$ under some action a. One such example is shown in Figure 2. However, "flattening" a hierarchical model in this manner will destroy the ability to learn faster by reusing sub-models, and negate one of the primary advantages of the hierarchical approach.

IV. Learning hierarchical POMDPs for robot navigation

We now describe a detailed set of experiments comparing hierarchical POMDP models with flat models in terms of learning speed and fit to the data. We have conducted experiments using both a real robot platform (a Nomad 200 robot) and a simulated environment. Using the Nomad 200 simulator we constructed a model of the second floor of the MSU Engineering building, shown in Figure 3. We also used a real indoor environment, shown in Figure 10.

Such topological maps can be automatically compiled into a Markov representation, either as a flat POMDP or as a hierarchical POMDP model. In our experiments, we used both "good" initial models as well as uninformed weak "ergodic" initial models. Figures 4, 5, and 6 show hierarchical and flat POMDP models. In a good initial model, we provide a priori the appropriate connectivity of the actions and we also initialize the observation models according to the location of a state. An observation consists of 5 components: the action taken and the probabilities that the robot has seen a wall or an opening on its four sides, front, left, back and right. The probabilities of the observations are computed by neural nets which take as input local occupancy grid maps around the robot (constructed from 16 sonars) and output the probability of a wall and opening for every direction [8]. For example, if a product state is facing in the corridor direction, then the initial observation model would be front (wall:0.4, opening:0.6), left (wall:0.6, opening:0.4), back (wall:0.4, opening:0.6), and right (wall:0.6, opening:0.4). In the case of uninformed ergodic initial models, we also provide good observation models, but every state is connected with every other state under the same parent for every action at all levels.

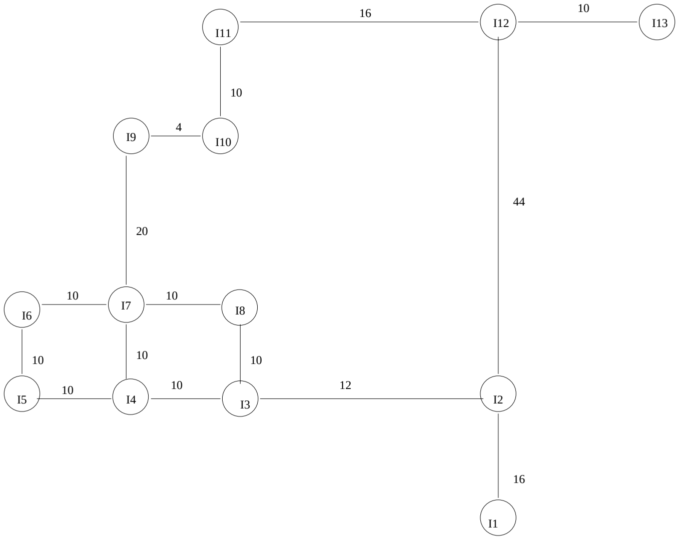


Fig. 3. A topological map of the 2nd floor of the Engineering building. Numbers indicate distances in meters. For each edge and each vertex there are four states at the second level of the hierarchical POMDP. States representing the (North-South or East-West) direction of corridors are expanded to a number of third level product states, each one representing two meters.

In the first experiment, we compared hierarchical and flat POMDPs, where we biased the training with a good initial model. We collected 26 "short" observation sequences, where a short sequence is one where the robot goes from one topological node to the next. We trained a hierarchical model, a "hierarchical with reuse" model (in which all the low level submodels were trained separately) and a flat model as shown in Figure 7. For the "hierarchical with reuse" model we also collected a separate sequence of observations for every abstract state. The result in Figure 7 shows that the hierarchical with reuse model converges a lot faster, and the fit to the the data is very good right from the beginning due to the fact that the submodels were already trained. We say a model converges when the mean

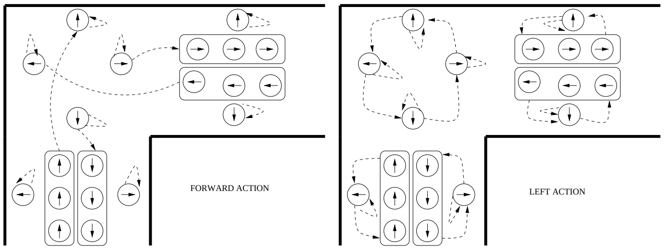


Fig. 4. These figures show how we represent a corridor environment as a hierarchical POMDP model. The circles represent product states and the solid arrows inside the circle indicate their orientation. The rectangles indicate abstract states (level 2) and the circles inside the rectangles are product states at level 3. The dashed arrows on the left show the transition matrix for the forward action at level 2 for a "good initial model". In the ergodic initial model (not shown), transitions are possible between all states under a given parent (for both abstract and product states). Before training we set all transitions to 0.5. The figure on the right shows the initial transition matrix for the "turn left" action. The "turn right" action is similar, except that the transitions are reversed.

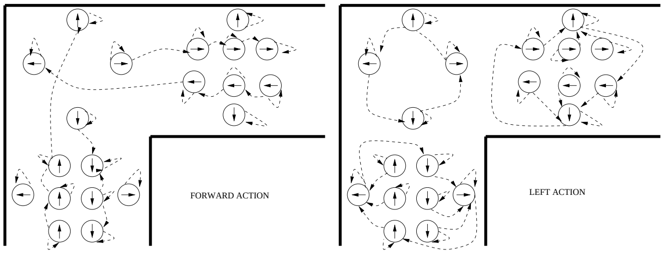


Fig. 5. These figures show the equivalent flat POMDP corridor model that can be automatically derived from the hierarchical POMDP model. The figure on the left show the transitions for the forward action and the one on the right shows the left action (both for good initial models).

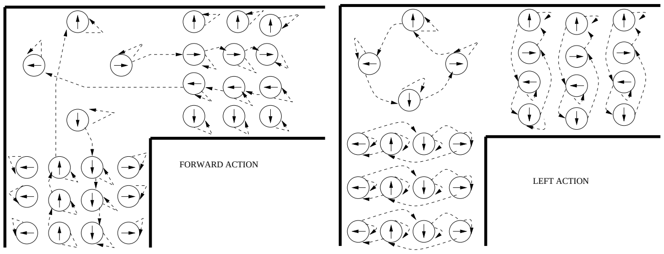


Fig. 6. These figures show how we represent a corridor environment as a flat POMDP model. The figure on the left shows the forward action and the figure on the right the turn left action (for good initial models).

square error across successive log likelihood values for all training traces remains stable within some threshold.

In the second experiment, we compared hierarchical and flat POMDPs, beginning with weak uninformed ergodic models. The training convergence is shown in Figure 8. In this experiment, we see that the hierarchical with reuse model fits the data significantly better than the hierarchical and the flat models.

In a third experiment we trained the same models as those used in experiment 2 using one long sequence of 150 observations as shown in Figure 9. We observe that the hierarchical with reuse model and its equivalent flat model fit the data better than all the other models. The flat POMDP provides a poorer fit to the data than the other models. Table I shows both the size (number of states) of the various models and the time (number of seconds) it takes for a training epoch in each experiment. The hierarchical models and their equivalent flat models train faster than the flat POMDP model, partly due to the smaller number of states in these models as compared to the flat model.

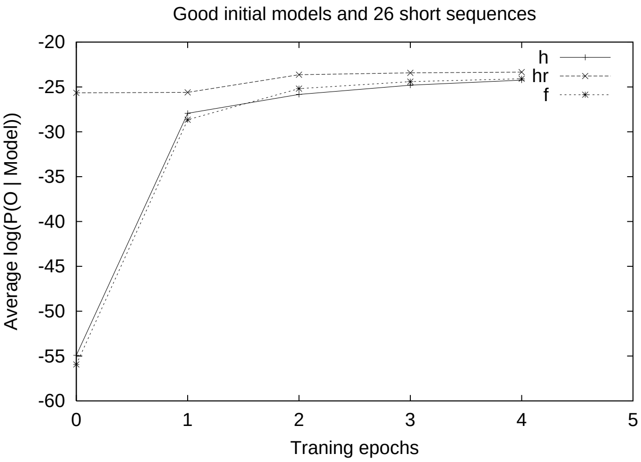


Fig. 7. The graph shows the training convergence between 3 different models. Model convergence is measured by mean square error across successive log likelihood values. The horizontal axis represents the number of training epochs and the vertical axis the average log likelihood fit of 26 different sequences of observations. *h* stands for a hierarchical model, *hr* stands for a hierarchical model whose initial model was created by first training separately the different ergodic models at level 3, and *f* stands for a flat model. Good initial models were provided for the training cases.

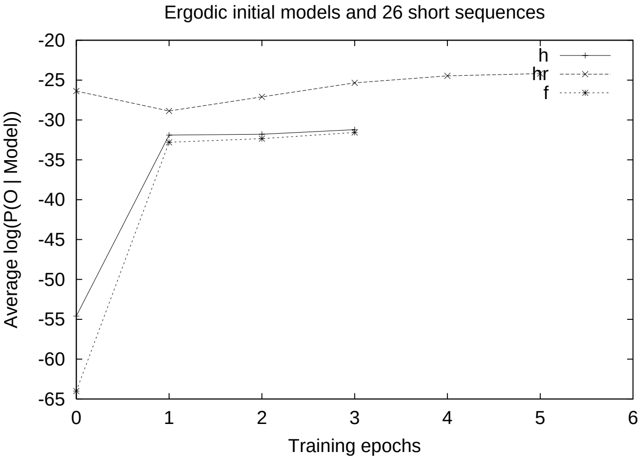


Fig. 8. The graph shows the training convergence between 3 different models. The X axis represents the number of training steps and the Y axis the average log likelihood fit of 26 different sequences of observations. *h* stands for a hierarchical model, where the starting model was ergodic, *hr* stands for a hierarchical model whose initial model was created by first training separately the different ergodic models at level 3, and *f* stands for a flat model. Here the starting model was also ergodic.

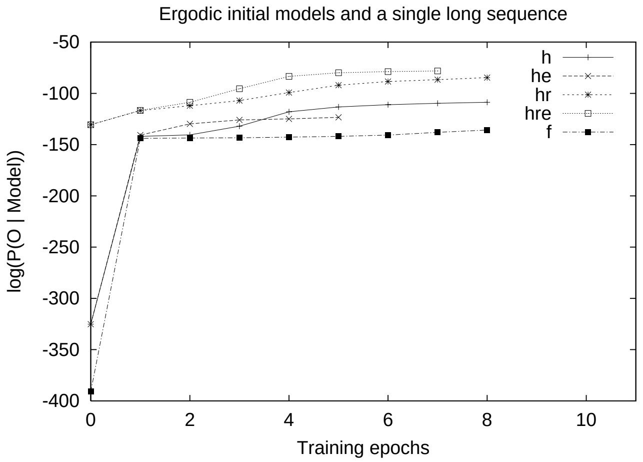


Fig. 9. The graph shows the training convergence between 5 different models for one long observation sequence. *h* stands for a hierarchical model, *he* stands for a flat model equivalent to the hierarchical, *hr* stands for a hierarchical with reuse model whose initial model was created by first training separately the different ergodic modules at level 3, *hre* is the equivalent flat model of the hierarchical with reuse model, and *f* stands for a flat model. For all cases, the starting models were ergodic.

	Figure 7, 8	Figure 9	Figure 11
Model	Time (Size)	Time (Size)	Time (Size)
h, hr	328 (346)	2200 (346)	300 (256)
he, hre	443 (286)	677 (286)	490 (232)
f	1416 (458)	21550 (458)	560 (410)

TABLE I

This table shows size of each model (number of states), and the time it takes (number of seconds) to train per epoch.

V. Real Robot Experiments

We also did experiments in an actual indoor environment as shown in Figure 10, where we collected 1 long sequence of a 100 observations using the mobile robot shown in the figure. The training convergence is shown in Figure 11, where the hierarchical with reuse model fits the data better than the rest of the models. The flat POMDP model once again provides the worst fit to the data and took the longest time to train, due to its size.

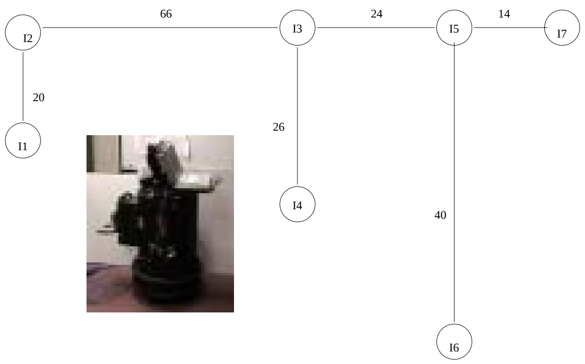


Fig. 10. The figure shows the topological map of a real indoor environment that is learned by PAVLOV (a Nomad 200 platform).

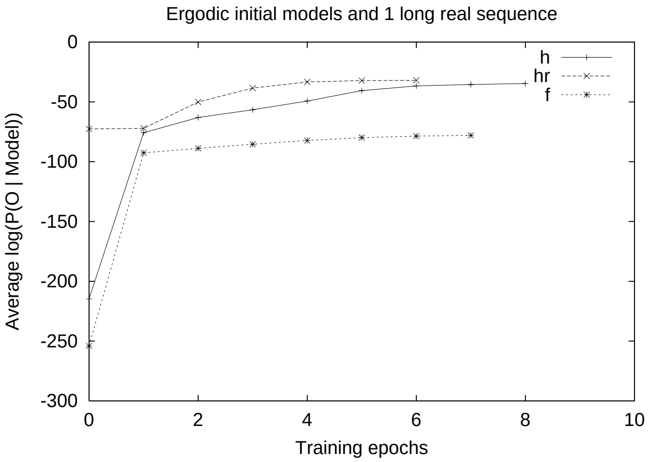


Fig. 11. The graph compares two hierarchical (*h* and *hr*)and one flat (*f*) model in terms of the goodness of fit and convergence (training epochs) for a real corridor environment.

VI. Conclusions and Future Work

In this paper, we describe a framework for learning hierarchical models of partially observable indoor corridor environments. We presented a modified Baum-Welch algorithm for learning hierarchical POMDP models. Using the new algorithm we compared flat and hierarchical POMDP models for both simulated and real world robot navigation environments. Our experimental results show that the hierarchical POMDP model is smaller in size, and correspondingly faster to train, even starting from a weak ergodic initial model. Furthermore, hierarchical models provide a natural way of reusing previously learned submodels. The fit to the data of the hierarchical models was also better than the flat model. We are now investigating alternate ways of defining abstract actions using the hierarchical nature of our POMDP model, and how abstract actions can speed up planning.

References

[1] Sven Koenig and Reid Simmons, "A Robot Navigation Architecture Based on Partially Observable Markov Decision Process Models," in *Artificial Intelligence Based Mobile Robotics:Case Studies of Successful Robot Systems.* MIT press, 1998.

[2] I. Nourbakhsh, R. Powers, and S. Birchfield, "Dervish: An office-navigation robot," in *AI Magazine 16(2):53-60.* 1995.

[3] Lawrence Rabiner, "Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," in *Proceedings of the IEEE*, February 1989, vol. 77.

[4] Hagit Shatkay and Leslie Kaebling, "Learning Topological Maps with Weak Local Odeometric Information," in *IJCAI97*, 1997.

[5] Shai Fine, Yoram Singer, and Naftali Tishby, "The Hierarchical Hidden Markov Model: Analysis and Applications," *Machine Learning*, vol. 32, no. 1, July 1998.

[6] Leslie Pack Kaebling, Michael L. Litman, and Anthony R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, 1998.

[7] Anthony R. Cassandra, Leslie Pack Kaelbling, and James A. Kurien, "Acting under uncertainty: Discrete bayesian models for mobile robot navigation," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1996.

[8] Sridhar Mahadevan, Georgios Theocharous, and Nikfar Khaleeli, "Fast concept learning for mobile robots," in *Autonomous Robots Journal, vol. 5, pp. 239-251.* 1998.

Continuous-Time Hierarchical Reinforcement Learning

Mohammad Ghavamzadeh GHAVAMZA@CSE.MSU.EDU
Sridhar Mahadevan MAHADEVA@CSE.MSU.EDU
Department of Computer Science, Michigan State University, East Lansing, MI 48824-1226, USA

Abstract

Hierarchical reinforcement learning (RL) is a general framework which studies how to exploit the structure of actions and tasks to accelerate policy learning in large domains. Prior work in hierarchical RL, such as the MAXQ method, has been limited to the discrete-time discounted reward semi-Markov decision process (SMDP) model. This paper generalizes the MAXQ method to continuous-time discounted and average reward SMDP models. We describe two hierarchical reinforcement learning algorithms: *continuous-time discounted reward MAXQ* and *continuous-time average reward MAXQ*. We apply these algorithms to a complex multiagent AGV scheduling problem, and compare their performance and speed with each other, as well as several well-known AGV scheduling heuristics.

1. Introduction

Hierarchical methods provide a general framework for scaling reinforcement learning to problems with large state spaces by using the task (or action) structure to restrict the space of policies. Prior work in hierarchical RL, including HAMs (Parr, 1998), options (Sutton et al., 1999) and MAXQ (Dietterich, 2000), has been limited to the discrete-time discounted reward SMDP model. This paper extends the MAXQ hierarchical RL framework to the continuous-time SMDP model, and introduces two new versions of MAXQ: one for the continuous-time discounted model, and one for the continuous-time average reward model. Although average reward RL has been extensively studied, using both the discrete-time MDP model (Schwartz, 1993; Mahadevan, 1996; Tadepalli & Ok, 1996) as well as the continuous-time SMDP model (Mahadevan et al., 1997; Wang & Mahadevan, 1999), prior work has been limited to "flat" algorithms. Both the proposed algorithms are tested on a complex multiagent AGV scheduling task.

The rest of this paper is organized as follows. Section 2 briefly introduces the continuous-time SMDP framework under both discounted and average reward paradigms. Section 3 describes the MAXQ method using an automated guided vehicle (AGV) task. Section 4 and 5 illustrate the continuous-time discounted reward MAXQ and continuous-time average reward MAXQ algorithms, respectively. Section 6 presents experimental results of using proposed algorithms in a multiagent AGV scheduling problem. Finally, section 7 summarizes the paper and discusses some directions for future work.

2. Semi-Markov Decision Processes

Semi-Markov decision processes (SMDPs) are useful in modeling temporally extended actions. They extend the discrete-time MDP model in several aspects. Time is modeled as a continuous entity and decisions are only made at discrete points in time (or *events*). The state of the system may change continually between decisions, unlike MDPs where state changes are only due to actions.

An SMDP is defined as a five tuple (S,A,P,R,F), where S is a finite set of states, A is the set of actions, P is a set of state and action dependent transition probabilities, R is the reward function, and F is a function giving probability of transition times for each state-action pair. $P(s'|s,a)$ denotes the probability that action a will cause the system to transition from state s to state s'. This transition is at decision epochs only. Basically, the SMDP represents snapshots of the system at decision points, whereas the so-called *natural process* describes the evolution of the system over all times. $F(t|s,a)$ is the probability that the next decision epoch occurs within t time units after the agent chooses action a in state s at a decision epoch. From F and P, we can compute Φ by

$$\Phi(t,s'|s,a) = P(s'|s,a)F(t|s,a)$$

where Φ denotes the probability that the system will be in state s' for the next decision epoch, at or before t time units after choosing action a in state s, at the last decision epoch. The reward function for SMDPs is more complex than in the MDP model. In addition to the fixed reward of taking action a in state s, $k(s,a)$, an additional reward may be accumulated at rate $c(s',s,a)$ for the time the natural process remains in state s' between decision epochs. Formally, the expected reward between two decision epochs, given that the system is in state s and chooses action a in the first decision epoch, is expressed as

$$r(s,a) = k(s,a) + E_s^a\{\textstyle\int_0^\tau c(W_t,s,a)dt\}$$

where τ is the transition time to the second decision epoch and W_t denotes the state of the natural process during this transition.

2.1 Discounted Models

We begin with a short overview of infinite-horizon discounted semi-Markov decision processes (Puterman, 1994; Bradtke & Duff, 1995). We assume continuous-time discounting at rate $\beta > 0$, which means that the present value of one reward unit received t time units in the future equals $e^{-\beta t}$. In this model, for policy π, $v^\pi(s)$ denotes the expected infinite-horizon discounted reward, given that the process occupies state s at the first decision epoch and is defined by

$$v^\pi(s) = E_s^\pi\{\sum_{n=0}^{\infty} e^{-\beta\sigma_n}[k(s_n,a_n) + \int_{\sigma_n}^{\sigma_{n+1}} e^{-\beta(t-\sigma_n)} c(W_t,s_n,a_n)dt]\} \tag{1}$$

In the above expression, $\sigma_0, \sigma_1, \ldots$ represents the times of successive decision epochs and $e^{-\beta\sigma_n}$ transforms the reward to values at the first decision epoch. In this model, the expected discounted reward between two decision epochs is defined as

$$r(s,a) = k(s,a) + \int_0^\infty \sum_{s'\in S}[\int_0^u e^{-\beta t} c(s',s,a)P(s'|s,a)dt]F(du|s,a) \tag{2}$$

Using Equation 2, we can reexpress the value function in Equation 1 as

$$v^\pi(s) = r(s,\pi(s)) + \sum_{s'\in S}\int_0^\infty e^{-\beta t} v^\pi(s')\Phi(dt,s'|s,\pi(s))$$

The action value function $Q^\pi(s,a)$ represents the discounted cumulative reward of doing an action a in state s once, and then following policy π subsequently.

$$Q^\pi(s,a) = r(s,a) + \sum_{s'\in S} P(s'|s,a)\int_0^\infty e^{-\beta t} Q^\pi(s',\pi(s'))F(dt|s,a)$$

2.2 Average Reward Models

The theory of infinite-horizon semi-Markov decision processes with the average reward criterion is more complex than that for discounted models. (Puterman, 1994; Mahadevan, 1996). To simplify exposition we assume that for every stationary policy, the embedded Markov chain has a unichain transition probability matrix. Under this assumption, the expected average reward of every stationary policy does not vary with the initial state. For policy π, state $s \in S$ and time $t \geq 0$, $v_t^\pi(s)$ denotes the expected total reward generated by the process up to time t, given that the system occupies state s at time 0 and is defined as

$$v_t^\pi(s) = E_s^\pi\{\textstyle\sum_{n=0}^{v_t-1} k(s_n,a_n) + \int_0^t c(W_u, s_{v_u}, a_{v_u})du\}$$

where v_u is the number of decisions made up to time t. In this model, the expected total reward between two decision epochs is defined as

$$r(s,a) = k(s,a) + \int_0^\infty \sum_{s'\in S}[\int_0^u c(s',s,a)P(s'|s,a)dt]F(du|s,a)$$

The average expected reward or gain $g^\pi(s)$ for a policy π at state s can be defined by taking the limit inferior of the ratio of the expected total reward up until the nth decision epoch to the expected total time until the nth epoch. So, the gain of a policy $g^\pi(s)$ can be expressed as the ratio

$$g^\pi(s) = \lim_{n\to\infty} \frac{E_s^\pi\{\sum_{i=0}^n[k(s_i,a_i) + \int_{\sigma_i}^{\sigma_{i+1}} c(W_t,s_i,a_i)dt]\}}{E_s^\pi\{\sum_{i=0}^n \tau_i\}}$$

For unichain MDPs, the gain of any policy is state independent and we can write $g^\pi(s) = g^\pi$. For each transition, the expected transition time is defined as:

$$y(s,a) = E_s^a\{\tau\} = \textstyle\int_0^\infty t \sum_{s'\in S} \Phi(dt,s'|s,a)$$

In unichain average reward SMDPs, the expected average adjusted sum of rewards h^π for stationary policy π is defined as

$$h^\pi(s) = V_t^\pi(s) - g^\pi t \tag{3}$$

where t is the time at which the decision epoch occurs. The Bellman equation for unichain average reward SMDPs is defined based on the h function in

Equation 3 and can be written as

$$h^{\pi}(s) = r(s,\pi(s)) - g^{\pi} y(s,\pi(s)) + \sum_{s' \in S} P(s'|s,\pi(s)) h^{\pi}(s')$$

The action value function $R^{\pi}(s,a)$ represents the average adjusted value of doing an action a in state s once, and then following policy π subsequently.

$$R^{\pi}(s,a) = r(s,a) - g^{\pi} y(s,a) + \sum_{s' \in S} P(s'|s,a) R^{\pi}(s',\pi(s'))$$

3. The MAXQ Framework

The continuous-time hierarchical reinforcement learning algorithms introduced in this paper are extensions of the MAXQ method for discrete-time hierarchical reinforcement learning (Dietterich, 2000). This approach involves the use of a graph to store a distributed value function. The overall task is first decomposed into subtasks up to the desired level of detail, and the task graph is constructed. We illustrate the idea using the AGV scheduling task used as the experimental testbed in this paper. Automated Guided Vehicles (AGVs) are used in flexible manufacturing systems (FMS) for material handling. Any FMS system using AGVs faces the problem of optimally scheduling the paths of AGVs in the system. Also, when a vehicle becomes available, and multiple move requests are queued, a decision needs to be made as to which request should be serviced by that vehicle. Hence, AGV scheduling requires dynamic dispatching rules, which are dependent on the state of the system like the number of parts in each buffer, the state of the AGV, and the processing going on at the workstations. The system performance is usually measured in terms of the *throughput*, which is the number of finished assemblies deposited at the unloading deck per unit time. Figure 1 shows the layout of a factory environment. Parts of type i have to be carried to drop-off station at machine i (D_i) and the assembled parts brought back into the warehouse. This is a task which can be parallelized, if we have more than one AGV working on it. Note the agents need to learn three skills here. First, how to do each subtask, such as deliver material to stations or navigation and when to perform *Pickup* or *Load* action. Second, agents also need to learn the order to do subtasks (e.g. go to load station and load part i before heading to the drop off station at machine i). Finally, AGVs need to learn how to coordinate with other AGVs (i.e. $AGV1$ can deliver part to machine i whereas $AGV2$ can deliver the finished assembly from machine j). The strength of the MAXQ framework is that it can serve as a substrate for learning all these three types of skills.

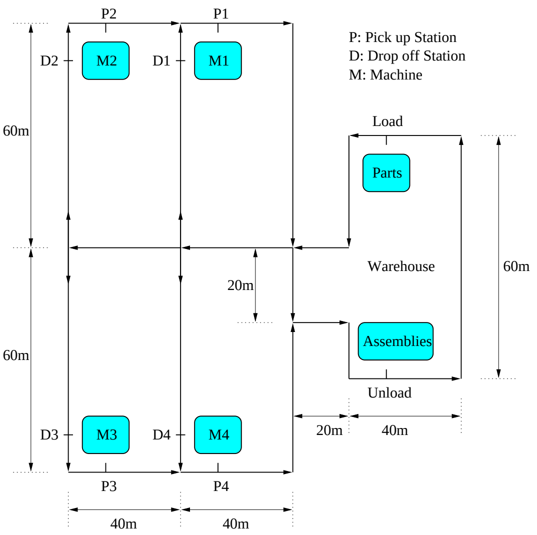


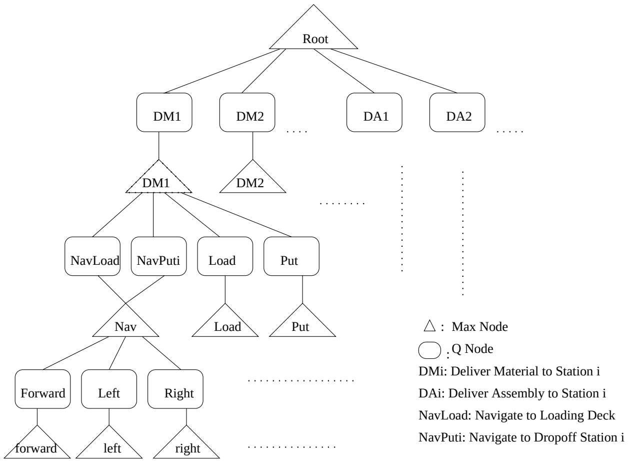
Figure 1. An AGV optimization task with four AGV agents (not shown) which carry raw materials and finished parts between the machines and the warehouse.

First, the AGV scheduling task is decomposed into subtasks and its task graph is built. Then its task graph is converted to the MAXQ graph, which is shown in figure 2. The MAXQ graph has two types of nodes: *MAX* nodes (triangles) and *Q* nodes (rectangles), which represent the different actions that can be done under their parents.

More formally, the MAXQ method decomposes an MDP M into a set of subtasks $M_0, M_1, ..., M_n$. Each subtask is a three tuple $(T_i, A_i, \tilde{R}_i)$ defined as:

- $T_i(s)$ is a termination predicate which partitions the state space S into a set of active states S_i, and a set of terminal states T_i. The policy for subtask M_i can only be executed if the current state $s \in S_i$.
- A_i is a set of actions that can be performed to achieve subtask M_i. These actions can either be primitive actions from A, the set of primitive actions for the MDP M, or they can be other subtasks.
- $\tilde{R}_i(s')$ is the pseudo reward function, which specifies a *pseudo-reward* for each transition to a terminal state $s' \in T_i$. This *pseudo-reward* tells how desirable each of the terminal states is for this particular subtask.

Each primitive action a is a primitive subtask in the MAXQ decomposition, such that a is always executable, it terminates immediately after execution,


Root
DM1
DM2
DA1
DA2
DM1
DM2
NavLoad
NavPuti
Load
Put
Nav
Load
Put
Forward
Left
Right
forward
left
right
△ : Max Node
: Q Node
DMi: Deliver Material to Station i
DAi: Deliver Assembly to Station i
NavLoad: Navigate to Loading Deck
NavPuti: Navigate to Dropoff Station i

(b)

Figure 2. MAXQ graph for the AGV scheduling task.

and its *pseudo-reward* function is uniformly zero. The projected value function V^π is the value of executing hierarchical policy π starting in state s, and at the root of the hierarchy. The completion function $(C^\pi(i, s, a))$ is the expected cumulative discounted reward of completing subtask M_i after invoking the subroutine for subtask M_a in state s.

The value function $V(i, s)$ in the MAXQ method is calculated by decomposing it into two parts: the value of the subtask which is independent of the parent task, and the value of the completion of the task, which of course depends on the parent task.

$$V(i,s) = \begin{cases} max_a Q(i,s,a) & if\ i\ is\ composite \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & if\ i\ is\ primitive \end{cases}$$

$$Q(i,s,a) = V(a,s) + C(i,s,a) \tag{4}$$

The Q values and the C values can be learned through a standard temporal-difference learning method, based on sample trajectories (see (Dietterich, 2000) for details). One important point to note here is that since subtasks are temporally extended in time, the Q-learning update rule used here is based on the SMDP model (Puterman, 1994).

Let us assume that an agent is at state s while doing task i, and chooses subtask j to execute. Let this subtask terminate after N steps and result in state s'. Then, the SMDP Q-learning rule used to update the completion function is given by

$$C_{t+1}(i,s,j) \leftarrow (1-\alpha)C_t(i,s,j) + \alpha\gamma^N(\max_{a'} V_t(a',s') + C_t(i,s',a'))$$

A *hierarchical* policy π is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0 \ldots \pi_n\}$. The projected value function in the hierarchical case, denoted by $V^\pi(s)$, is the value of executing hierarchical policy π starting in state s and starting at the root of the task hierarchy. A *recursively optimal* policy for MDP M with MAXQ decomposition $\{M_0 \ldots M_n\}$ is a hierarchical policy $\pi = \{\pi_0 \ldots \pi_n\}$ such that for each subtask M_i the corresponding policy π_i is optimal for the SMDP defined by the set of states S_i, the set of actions A_i, the state transition probability function $P^\pi(s', N|s, a)$, and the reward function given by the sum of the original reward function $R(s'|s, a)$ and the pseudo-reward function $\tilde{R}_i(s')$. The MAXQ learning algorithm has been proven to converge to the unique recursively optimal policy for MDP M and MAXQ graph H, where M is a discounted infinite horizon MDP with discount factor γ, and H is a MAXQ graph defined over subtasks $\{M_0 \ldots M_n\}$.

4. Continuous-Time Discounted Reward MAXQ Algorithm

At the center of the MAXQ method for hierarchical reinforcement learning is the MAXQ value function decomposition. We show how the overall value function for a policy is decomposed into a collection of value functions for individual subtasks for the continuous-time discounted reward model. The projected value function of hierarchical policy π on subtask M_i, denoted $V^\pi(i, s)$, is the expected cumulative discounted reward of executing π_i (and the policies of all descendents of M_i) starting in state s until M_i terminates. The value $V^\pi(i, s)$ has the following form in the continuous-time discounted reward framework:

$$V^\pi(i,s) = E_s^\pi\{\sum_{n=0}^{\infty} e^{-\beta\sigma_n} r(s_n, a_n)\} \tag{5}$$

where $r(s_n, a_n)$ is defined using Equation 2. Now let us suppose that the first action chosen by π_i is invoked and executes for a number of steps N and terminates in state s' according to $P_i^\pi(s'|s, a)$. We can rewrite Equation 5 as

$$V^\pi(i,s) = E_s^\pi\{\sum_{n=0}^{N-1} e^{-\beta\sigma_n} r(s_n, a_n) + e^{-\beta\sigma_N} \sum_{n=0}^{\infty} e^{-\beta\sigma_n} r(s_{N+n}, a_{N+n})\} \tag{6}$$

The first summation on the right-hand side of Equation 6 is the discounted sum of rewards for executing subroutine $\pi_i(s)$ starting in state s until it terminates, in other words, it is $V^\pi(\pi_i(s), s)$, the projected value function for the child task $M_{\pi_i(s)}$. The second term on the right-hand side of the equation is the value of s' for the current task i, $V^\pi(i, s')$, discounted by $e^{-\beta t}$, where s' is the current state when subroutine $\pi_i(s)$ terminates and t is the sample transition time from state s to state s'. We can write Equation 6 in the form of a Bellman equation:

$$V^\pi(i,s) = V^\pi(\pi_i(s), s) + \sum_{s' \in S_i} P_i(s'|s, \pi_i(s)) \int_0^\infty e^{-\beta t} V^\pi(i, s') F_i(dt|s, \pi_i(s)) \tag{7}$$

Equation 7 can be restated for action-value function decomposition as follows:

$$Q^\pi(i,s,a) = V^\pi(a,s) + \sum_{s' \in S_i} P_i(s'|s,a) \int_0^\infty e^{-\beta t} Q^\pi(i, s', \pi_i(s')) F_i(dt|s,a)$$

The right-most term in this equation is the expected discounted cumulative reward of completing task M_i after executing action a in state s. This term is called the *completion function* and is denoted by $C^\pi(i,s,a)$. With this definition, we can express the Q function recursively as

$$Q^\pi(i,s,a) = V^\pi(a,s) + C^\pi(i,s,a)$$

and we can reexpress the definition for V as

$$V^\pi(i,s) = \begin{cases} Q^\pi(i,s,\pi_i(s)) & if \;\; i \;\; is \; composite \\ k_i(s,i) + \displaystyle\int_0^\infty \sum_{s' \in S_i} P_i(s'|s,i) \\ \qquad [\displaystyle\int_0^u e^{-\beta t} c_i(s',s,i) dt] F_i(du|s,i) & \\ & if \;\; i \;\; is \; primitive \end{cases}$$

We can use the above formulas to obtain update equations for value function V, outside completion function C and inside completion function $\tilde{C}$ in the continuous-time discounted reward model. Pseudo-code for the resulting algorithm is shown in Algorithm 1 [1].

Algorithm 1 The continuous-time discounted reward MAXQ algorithm.

```
function MAXQ(MaxNode $i$, State $s$)
let Seq={} be the sequence of (states visited, transition times) while executing $i$
if $i$ is a primitive MaxNode then
    execute action $i$ in state $s$, observe state $s'$ in $\tau$ time units, receive lump portion of reward $k(s,i)$ and continuous portion of reward with rate $r(s',s,i)$
        $V_{t+1}(i,s) \xleftarrow{\alpha} [k(s,i) + \frac{1-e^{-\beta\tau}}{\beta} r(s',s,i)]$
    push (state $s$, transition time $\tau$) into the beginning of Seq
else
    while $i$ has not terminated do
        choose action $a$ according to the current exploration policy $\pi_i(s)$
        let ChildSeq=MAXQ($a$,$s$), where ChildSeq is the sequence of (states visited, transition times) while executing action $a$
        observe result state $s'$
        let $a^* = argmax_{a' \in A_i(s')}[\tilde{C}_t(i,s',a') + V_t(a',s')]$
        $T = 0$;
        for ($s$,$\tau$) in ChildSeq from the beginning do
            $T = T + \tau$
                $\tilde{C}_{t+1}(i,s,a) \xleftarrow{\alpha} e^{-\beta T}[\tilde{R}_i(s') + \tilde{C}_t(i,s',a^*) + V_t(a^*,s')]$
                $C_{t+1}(i,s,a) \xleftarrow{\alpha} e^{-\beta T}[C_t(i,s',a^*) + V_t(a^*,s')]$
        end for
        append ChildSeq onto the front of Seq
        $s = s'$
    end while
end if
return Seq
end MAXQ
```

[1] We use the notation $u \xleftarrow{\alpha} v$ in Algorithm 1 and Algorithm 2 as an abbreviation for the stochastic approximation update rule $u \leftarrow (1-\alpha)u + \alpha v$.

5. Continuous-Time Average Reward MAXQ Algorithm

We now describe a new average reward hierarchical reinforcement learning algorithm based on the MAXQ framework. To simplify exposition, we assume that for every possible stationary policy of each subtask in the hierarchy, the embedded Markov chain has a unichain transition probability matrix. Under this assumption every subtask in the hierarchy is a unichain SMDP. This means the expected average reward of every stationary policy for each subtask in the hierarchy does not vary with initial state. As we mentioned earlier, value function decomposition is the heart of the MAXQ method. We show how the overall h function for a policy is decomposed into a collection of h functions for individual subtasks in the continuous-time average reward MAXQ method. The projected h function of hierarchical policy π on subtask M_i, denoted $h^\pi(i,s)$, is the average adjusted sum of rewards earned of following policy π_i (and the policies of all descendents of M_i) starting in state s until M_i terminates:

$$h^\pi(i,s) = \lim_{N\to\infty} E_s^\pi\{\sum_{t=0}^{N-1}(r(s_t,a_t) - g^i\tau_t)\} \tag{8}$$

where τ's and g^i are the length of decision epochs and gain of subtask M_i respectively. Now let us suppose that the first action chosen by π is invoked and executes for a number of steps and terminates in state s' according to $P_i^\pi(s'|s,a)$. We can write Equation 8 in the form of a Bellman equation:

$$\begin{aligned} h^\pi(i,s) = r(s,\pi_i(s)) - g^i y_i(s,\pi_i(s)) \\ + \sum_{s'\in S_i} P_i(s'|s,\pi_i(s))h^\pi(i,s') \end{aligned} \tag{9}$$

Since $r(s,\pi_i(s))$ is the expected total reward between two decision epochs of subtask i, given that the system occupies state s at the first decision epoch and decision maker chooses action $\pi_i(s)$ and the expected length of time until next decision epoch is $y_i(s,\pi_i(s))$, we have

$$\begin{aligned} r(s,\pi_i(s)) = V^\pi_{y_i(s,\pi_i(s))}(\pi_i(s),s) = h^\pi(\pi_i(s),s) \\ + g^{\pi_i(s)}y_i(s,\pi_i(s)) \end{aligned}$$

By replacing $r(s,\pi_i(s))$ from the above expression, Equation 9 can be written as

$$\begin{aligned} h^\pi(i,s) = h^\pi(\pi_i(s),s) - (g^i - g^{\pi_i(s)})y_i(s,\pi_i(s)) \\ + \sum_{s'\in S_i} P_i(s'|s,\pi_i(s))h^\pi(i,s') \end{aligned} \tag{10}$$

We can restate Equation 10 for action-value function decomposition as follows:

$$\begin{aligned} R^\pi(i,s,a) = h^\pi(a,s) - (g^i - g^a)y_i(s,a) \\ + \sum_{s'\in S_i} P_i(s'|s,a)R^\pi(i,s',\pi_i(s')) \end{aligned}$$

In the above equation, the term

$$-(g^i - g^a)y_i(s,a) + \textstyle\sum_{s'\in S_i} P_i(s'|s,a)R^\pi(i,s',\pi_i(s'))$$

denotes the average adjusted reward of completing task M_i after executing action a in state s. This term is called the completion function and is denoted by $C^\pi(i,s,a)$. With this definition, we can express the R function recursively as

$$R^\pi(i,s,a) = h^\pi(a,s) + C^\pi(i,s,a)$$

and we can reexpress the definition for h as

$$h^\pi(i,s) =$$

$$\begin{cases} R^\pi(i,s,\pi_i(s)) & if \;\; i \;\; is\; composite \\ k(s,i) + \int_0^\infty \sum_{s'\in S_i} P_i(s'|s,i)[\int_0^u c(s',s,i)dt]F_i(du|s,i) \\ \quad - g^i \sum_{s'\in S_i} P_i(s'|s,i)\int_0^\infty tF_i(dt|s,i) \\ & if \;\; i \;\; is\; primitive \end{cases} \tag{11}$$

The above formulas can be used to obtain update equations for h function, outside completion function C and inside completion function $\tilde{C}$ in the continuous-time average reward model. Pseudo-code for the resulting algorithm is shown in Algorithm 2. As mentioned above, all subtasks in the hierarchy, even primitive actions, are modeled by a unichain SMDP.

6. Experimental Results

We now apply the two proposed continuous-time algorithms to the AGV scheduling task described in section 3 and compare their performance and speed with each other, as well as several well-known AGV scheduling heuristics.

The experimental results were generated with the following model parameters. There are four AGVs in the environment, the inter-arrival time for parts at the warehouse is uniformly distributed with a mean of 4 sec and variance of 1 sec. The percentage of Part1, Part2, Part3 and Part4 in the part arrival process are

Algorithm 2 The continuous-time average reward MAXQ algorithm.

```
function MAXQ(MaxNode $i$, State $s$)
let Seq={} be the sequence of (states visited, transition times, reward) while executing $i$
if $i$ is a primitive MaxNode then
    execute action $i$ in state $s$, observe state $s'$ in $\tau$ time units, receive lump portion of reward $k(s,i)$ and continuous portion of reward with rate $r(s',s,i)$
        $h_{t+1}(i,s) \xleftarrow{\alpha} [k(s,i) + r(s',s,i)\tau - g_t^i\tau]$
    if $i$ is a non-random action then
        update average reward or gain of subtask $i$
            $g_{t+1}^i = \frac{r_{t+1}(i)}{t_{t+1}(i)} = \frac{r_t(i) + k(s,i) + r(s',s,i)\tau}{t_t(i) + \tau}$
    end if
    push (state $s$, transition time $\tau$, reward $\rho = k(s,i) + r(s',s,i)\tau$) into the beginning of Seq
else
    while $i$ has not terminated do
        choose action $a$ according to the current exploration policy $\pi_i(s)$
        let ChildSeq=MAXQ($a$,$s$), where ChildSeq is the sequence of (states visited, transition times) while executing action $a$
        observe result state $s'$
        let
        $a^* = argmax_{a' \in A_i(s')}[\tilde{C}_t(i,s',a') + V_t(a',s')]$
        $T = 0; \quad R = 0;$
        for (s,$\tau$,$\rho$) in ChildSeq from the beginning do
            $T = T + \tau; \quad R = R + \rho;$
            $\tilde{C}_{t+1}(i,s,a) \xleftarrow{\alpha} [\tilde{R}_i(s') - (g_t^i - g_t^a)T + \tilde{C}_t(i,s',a^*) + V_t(a^*,s')]$
            $C_{t+1}(i,s,a) \xleftarrow{\alpha} [C_t(i,s',a^*) + V_t(a^*,s') - (g_t^i - g_t^a)T]$
        if $a$ is a non-random action then
            update average reward or gain of subtask $i$
                $g_{t+1}^i = \frac{r_{t+1}(i)}{t_{t+1}(i)} = \frac{r_t(i) + R}{t_t(i) + T}$
        end if
        end for
        append ChildSeq onto the front of Seq
        $s = s'$
    end while
end if
return Seq
end MAXQ
```

20, 28, 22 and 30 respectively. The time required for assembling the various parts is normally distributed with means 15, 24, 24 and 30 sec for Part1, Part2, Part3 and Part4 respectively, and the variance 2 sec. The time required for primitive actions are also normally distributed. Each experiment was conducted five times and the results averaged. Since this is a multiagent task, we extend both proposed algorithms to the multiagent case using the approach introduced in (Makar et al., 2001).

In this approach (which we call cooperative MAXQ), each agent uses the same MAXQ hierarchy to decompose the task into subtasks. Learning is decentralized and coordination skills among agents are learned by using joint actions at the highest level of the hierarchy. The Q (or R) nodes at the highest level of the hierarchy are configured to represent the joint action space among multiple agents. In this approach, each agent only knows what other agents are doing at the level of high level subtasks, and is unaware of their lower level actions. This idea allows agents to learn coordination faster by sharing information at the level of subtasks, rather than attempting to learn coordination taking into account primitive joint state-action values.

Figure 3 compares the proposed MAXQ algorithms with several well-known AGV scheduling rules, showing clearly the improved performance of the reinforcement learning methods. As seen in this figure, the agents learn a little faster initially in the discounted MAXQ framework, but the final system throughput achieved using the average reward algorithm is higher than the discounted reward case. This result is consistent with the assumption that the undiscounted optimality framework is more appropriate for cyclical tasks same as AGV scheduling than the discounted framework.

7. Conclusions and Future Work

This paper describes two new continuous-time hierarchical RL algorithms based on the MAXQ framework. The effectiveness of both algorithms was demonstrated by applying them to a large scale multiagent AGV scheduling problem. The first algorithm extends the original discrete-time MAXQ algorithm to the continuous-time discounted SMDP model, whereas the second algorithm extends MAXQ to the continuous-time average reward SMDP model. As such, since the second algorithm is the first hierarchical average-reward algorithm proposed to our knowledge, it deserves further discussion. In particular, the MAXQ approach assumes that subtasks always terminate. How-

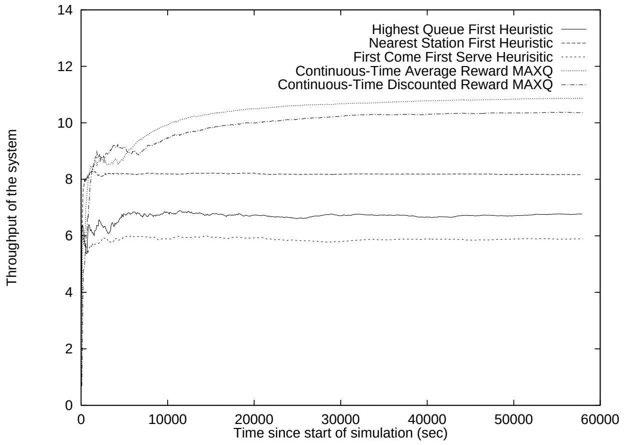


Figure 3. This plot shows both continuous-time average reward and discounted reward multiagent MAXQ algorithms outperform three well-known widely used (industrial) heuristics for AGV scheduling.

ever, for complete generality, we need to also consider the case where the subtasks are also cyclical and non-terminating, just as the overall task was in the AGV problem. One way to handle such non-terminating subtasks is to use *interruptions*, as implemented in the options framework (Sutton et al., 1999). Alternatively, one could imagine a mixed-mode framework where subtasks are optimized using an undiscounted total-reward criterion, and the parent task is formulated using an average reward criterion.

Many practical and theoretical issues remain unexplored in this research. We have not demonstrated a proof of convergence of the two algorithms. Analyzing the proof of average reward extension of MAXQ is particularly interesting. It is obvious that many other manufacturing and robotics problems can benefit from a MAXQ-like approach, particularly in the multiagent case where task-level coordination can greatly accelerate learning (Makar et al., 2001).

Acknowledgements

This work is supported by the Defense Advanced Research Projects Agency, DARPA contract No. DAANO2-98-C-4025.

References

Bradtke, S. J., & Duff, M. O. (1995). Reinforcement Learning Methods for Continuous-Time Markov Decision Problems. In G. Tesauro, D. Touretzky and T. Leen (Eds.), *Advances in neural information processing systems*, vol. 7, 393–400. Cambridge, MA.: The MIT Press.

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Mahadevan, S. (1996). Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results. *Machine Learning*, *22*, 159–196.

Mahadevan, S., Marchalleck, N., Das, T., & Gosavi, A. (1997). Self-Improving Factory Simulation using Continuous-Time Average Reward Reinforcement Learning. *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 202–210).

Makar, R., Mahadevan, S., & Ghavamzadeh, M. (2001). Hierarchical Multiagent Reinforcement Learning. *Proceedings of the Fifth International Conference on Autonomous Agents.*

Parr, R. E. (1998). *Hierarchical Control and Learning for Markov Decision Processes.* Doctoral dissertation, Department of Computer Science, University of California, Berkeley.

Puterman, M. L. (1994). *Markov Decision Processes.* New York, USA: Wiley Interscience.

Schwartz, A. (1993). A Reinforcement Learning Method for Maximizing Undiscounted Rewards. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 298–305).

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Journal of Artificial Intelligence*, 181–211.

Tadepalli, P., & Ok, D. (1996). Auto-Exploratory Average Reward Reinforcement Learning. *Proceedings of the Thirteenth AAAI* (pp. 881–887).

Wang, G., & Mahadevan, S. (1999). Hierarchical Optimization of Policy-Coupled Semi-Markov Decision Processes. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 466–473).

Hierarchical Multi-Agent Reinforcement Learning

Rajbala Makar[*]
Department of Computer Science
Michigan State University
East Lansing, MI 48824
makarraj@cse.msu.edu

Sridhar Mahadevan
Department of Computer Science
Michigan State University
East Lansing, MI 48824
mahadeva@cse.msu.edu

Mohammad Ghavamzadeh
Department of Computer Science
Michigan State University
East Lansing, MI 48824
ghavamza@cse.msu.edu

ABSTRACT

In this paper we investigate the use of hierarchical reinforcement learning to speed up the acquisition of cooperative multi-agent tasks. We extend the MAXQ framework to the multi-agent case. Each agent uses the same MAXQ hierarchy to decompose a task into sub-tasks. Learning is decentralized, with each agent learning three interrelated skills: how to perform subtasks, which order to do them in, and how to coordinate with other agents. Coordination skills among agents are learned by using joint actions at the highest level(s) of the hierarchy. The Q nodes at the highest level(s) of the hierarchy are configured to represent the joint task-action space among multiple agents. In this approach, each agent only knows what other agents are doing at the level of sub-tasks, and is unaware of lower level (primitive) actions. This hierarchical approach allows agents to learn coordination faster by sharing information at the level of sub-tasks, rather than attempting to learn coordination taking into account primitive joint state-action values. We apply this hierarchical multi-agent reinforcement learning algorithm to a complex AGV scheduling task and compare its performance and speed with other learning approaches, including flat multi-agent, single agent using MAXQ, selfish multiple agents using MAXQ (where each agent acts independently without communicating with the other agents), as well as several well-known AGV heuristics like "first come first serve", "highest queue first" and "nearest station first". We also compare the tradeoffs in learning speed vs. performance of modeling joint action values at multiple levels in the MAXQ hierarchy.

1. INTRODUCTION

Consider sending a team of robots to carry out reconnaissance of an indoor environment to check for intruders. This problem is naturally viewed as a multi-agent task [19]. The most effective strategy will require coordination among the individual robots. A natural decomposition of this task would be to assign different parts of the environments, for example rooms, to different robots. In this paper, we are interested in learning algorithms for such *cooperative* multi-agent tasks, where the agents learn the coordination skills by trial and error. The main point of the paper is simply that coordination skills are learned much more efficiently if the robots have a hierarchical representation of the task structure [13]. In particular, rather than each robot learning its response to low-level primitive actions of the other robots (for instance if robot-1 goes forward, what should robot-2 do), it learns high-level coordination knowledge (what is the utility of robot-2 searching room-2 if robot-1 is searching room-1, and so on).

We adopt the framework of reinforcement learning[14], which has been well-studied in both single agent and multi-agent domains. Multi-agent reinforcement learning has been recognized to be much more challenging, since the number of parameters to be learned increases dramatically with the number of agents. In addition, since agents carry out actions in parallel, the environment is usually non-stationary and often non-Markovian as well [9]. We do not address the non-stationary aspect of multi-agent learning in this paper. One approach that has been successful in the past is to have agents learn policies that are parameterized by the modes of interaction [18].

Prior work in multi-agent reinforcement learning can be decomposed into work on competitive models vs. cooperative models. Littman [8], and Hu and Wellman [5], among others, have studied the framework of Markov games for competitive multi-agent learning. Here, we are primarily interested in the cooperative case. The work on cooperative learning can be further separated based on the extent to which agents need to communicate with each other. On the one hand are studies such as Tan [17], which extend (flat) Q-learning to multi-agent learning by using joint state-action values. This approach requires communication of states and actions at every step. On the other hand are approaches such as Crites and Barto [3], where the agents share a common state description and a global reinforcement signal, but do not model joint actions. There are also studies of multi-agent learning which do not model joint states or actions explicitly, such as by Balch [2] and Mataric [9], among others. In such behavior-based systems, each robot maintains

[*]Currently at Agilent Technologies, CA.

AGENTS'01, May 28-June 1, 2001, Montréal, Quebec, Canada.
Copyright 2001 ACM 1-58113-326-X/01/0005 ...$5.00.

its position in the formation depending on the locations of the other robots, so there is some (implicit) communication or sensing of states and actions of other agents. There has also been work on reducing the parameters needed for Q-learning in multi-agent domains, by learning action values over a set of derived features (see Stone and Veloso [12]). These derived features are domain-specific, and have to be encoded by hand, or constructed by a supervised learning algorithm.

Our approach differs from all the above in one key respect, namely the use of explicit task structure to speed up cooperative multi-agent reinforcement learning. Hierarchical methods constitute a general framework for scaling reinforcement learning to large domains by using the task structure to restrict the space of policies. Several alternative frameworks for hierarchical reinforcement learning have been proposed, including options [15], HAMs [10] and MAXQ [4]. We assume each agent is given an initial hierarchical decomposition of the overall task (as described below, we adopt the MAXQ hierarchical framework). However, the learning is distributed since each agent has only a local view of the overall state space. Furthermore, each agent learns joint abstract action-values by communicating with each other only the high-level subtasks that they are doing. Since high-level tasks can take a long time to complete, communication is needed only fairly infrequently (this is another significant advantage over flat methods).

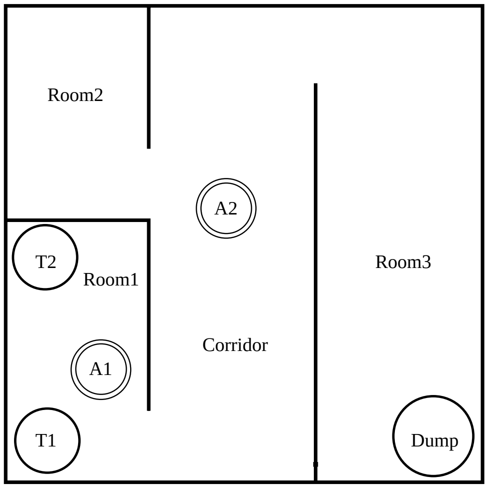
A further advantage of the use of hierarchy in multi-agent learning is that it makes it possible to learn co-ordination skills at the level of abstract actions. The agents learn joint action values only at the highest level(s) of abstraction in the proposed framework. This allows for increased co-operation skills as agents do not get confused by low level details. In addition, each agent has only local state information, and is ignorant about the other agent's location. This is based on the idea that in many cases, an agent can get a rough idea of what state the other agent might be in just by knowing about the high level action being performed by the other agent. Also, keeping track of just this information greatly simplifies the underlying reinforcement learning problem.

These benefits can potentially accrue with using any type of hierarchical learning algorithm, though in this paper we only describe results using the MAXQ framework. The reason that we decided to adopt the MAXQ framework as a basis for our multi-agent algorithm is the fact that the MAXQ method stores the value function in a distributed way in all nodes in the subtask graph. The value function is propagated upwards from the lower level nodes whenever a high level node needs to be evaluated. This propagation enables the agent to simultaneously learn subtasks and high level tasks. Thus, by using this method, agents learn the co-ordination skills and the individual low level tasks and subtasks all at once.

However, it is necessary to generalize the MAXQ framework to make it more applicable to multi-agent learning. A broad class of multi-agent optimization tasks, such as AGV scheduling, can be viewed as discrete-event dynamic systems. For such tasks, the termination predicate used in MAXQ has to be redefined to take care of the fact that the completion of certain subtasks might depend on the occurrence of an event rather than just a state of the environment. We extended the MAXQ framework to continuous-time MDP models, although we will not discuss this extension in this paper.

2. THE MAXQ FRAMEWORK

The multi-agent reinforcement learning algorithm introduced in this paper is an extension of the MAXQ method for single agent hierarchical learning [4]. This approach involves the use of a graph to store a distributed value function. The overall task is first decomposed into subtasks up to the desired level of detail, and the task graph is constructed. We illustrate the idea using a simple two-robot search task shown in Figure 1. Consider the case where a robot is assigned the task of picking up trash from trash cans over an extended area and accumulating it into one centralized trash bin, from where it might be sent for recycling or disposed. This is a task which can be parallelized, if we have more than one agent working on it. An office (rooms and connecting corridors) type environment is shown in figure. A1 and A2 represent the two agents in the figure. Note the agents need to learn three skills here. First, how to do each subtask, such as navigating to $T1$ or $T2$ or $Dump$, and when to perform $Pickup$ or $Putdown$ action. Second, the agents also need to learn the order to do subtasks (for instance go to $T1$ and collect trash before heading to the $Dump$). Finally, the agents also need to learn how to coordinate with other agents (i.e. $Agent1$ can pick up trash from $T1$ whereas $Agent2$ can service $T2$). The strength of the MAXQ framework (when extended to the multi-agent case) is that it can serve as a substrate for learning all these three types of skills.



T1: Location of one trash can.
T2: Location of another trash can.
Dump: Final destination location for depositing all trash.

Figure 1: A (simulated) multi-agent robot trash collection task.

This trash collection task can be decomposed into subtasks and the resulting task graph is shown in figure 2. The task graph is then converted to the MAXQ graph, which is shown in figure 3. The MAXQ graph has two types of nodes: MAX nodes (triangles) and Q nodes (rectangles), which represent the different actions that can be done under their parents. Note that MAXQ allows learning of shared subtasks. For example, the navigation task Nav is common to several parent tasks.

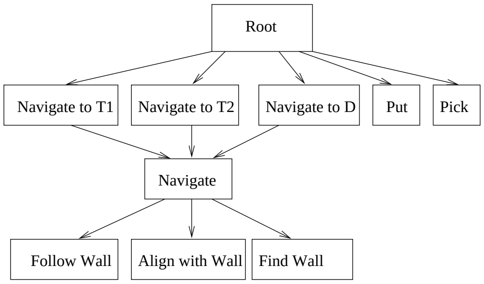


Figure 2: The task graph for the trash collection task.

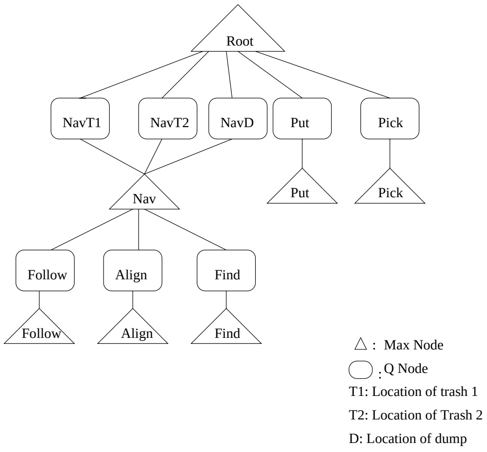


Figure 3: The MAXQ graph for the trash collection task.

The multi-agent learning scenario under investigation in this paper can now be illustrated. Imagine the two robots start to learn this task with the same MAXQ graph structure. We can distinguish between two learning approaches. In the *selfish* case, the two robots learn with the given MAXQ structure, but make no attempt to communicate with each other. In the *cooperative* case, the MAXQ structure is modified such that the Q nodes at the level(s) immediately under the root task include the joint action done by both robots. For instance, each robot learns the joint Q-value of navigating to trash $T1$ when the other robot is either navigating to $T1$ or $T2$ or $Dump$ or doing a Put or $Pick$ action. As we will show in a more complex domain below, cooperation among the agents results in superior learned performance than in the selfish case, or indeed the flat case when the agents do not use a task hierarchy at all.

More formally, the MAXQ method decomposes an MDP M into a set of subtasks $M_0, M_1 ... M_n$. Each subtask is a three tuple $(T_i, A_i, \overline{R_i})$ defined as:

- $T_i(s_i)$ is a termination predicate which partitions the state space S into a set of active states S_i, and a set of terminal states T_i. The policy for subtask M_i can only be executed if the current state $s \in S_i$.

- A_i is a set of actions that can be performed to achieve subtask M_i. These actions can either be primitive actions from A, the set of primitive actions for the MDP, or they can be other subtasks.

- $\overline{R_i}(s^{'}|s,a)$ is the pseudo reward function, which specifies a *pseudo-reward* for each transition from a state $s \in S_i$ to a terminal state $s^{'} \in T_i$. This *pseudo-reward* tells how desirable each of the terminal states is for this particular subtask.

Each primitive action a is a primitive subtask in the MAXQ decomposition, such that a is always executable, it terminates immediately after execution, and it's *pseudo-reward* function is uniformly zero. The projected value function V^{π} is the value of executing hierarchical policy π starting in state s, and at the root of the hierarchy. The completion function $(C^{\pi}(i,s,a))$ is the expected cumulative discounted reward of completing subtask M_i after invoking the subroutine for subtask M_a in state s.

The (optimal) value function $V_t(i,s)$ for doing task i in state s is calculated by decomposing it into two parts: the value of the subtask which is independent of the parent task, and the value of the completion of the task, which of course depends on the parent task.

$$V_t(i,s) = \begin{cases} max_a Q_t(i,s,a) & if\ i\ is\ composite \\ \sum_{s'} P(s' \mid s,i) R(s' \mid s,i) & if\ i\ is\ primitive \end{cases}$$

$$Q_t(i,s,a) = V_t(a,s) + C_t(i,s,a) \qquad (1)$$

where $Q_t(i,s,a)$ is the action value of doing subtask a in state s in the context of parent task i.
The Q values and the C values can be learned through a standard temporal-difference learning method, based on sample trajectories (see [4] for details). One important point to note here is that since subtasks are temporally extended in time, the update rules used here are based on the semi-Markov decision process (SMDP) model [11].

Let us assume that an agent is at state s while doing task i, and chooses subtask j to execute. Let this subtask terminate after N steps and result in state s'. Then, the SMDP Q-learning rule used to update the completion function is given by

$$C_{t+1}(i,s,j) \leftarrow (1-\alpha_t)C(i,s,j) + \alpha_t \gamma^N (\max_{a'} V(a',s') + C_t(i,s',a'))$$

A *hierarchical* policy π is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0 \dots \pi_n\}$. The projected value function in the hierarchical case, denoted by $V^{\pi}(s)$, is the value of executing hierarchical policy π starting in state s and starting at the root of the task hierarchy. A *recursively optimal* policy for MDP M with MAXQ decomposition $\{M_0 \dots M_n\}$ is a hierarchical policy $\pi = \{\pi_0 \dots \pi_n\}$ such that for each subtask M_i the corresponding policy π_i is optimal for the SMDP defined by the set of states S_i, the set of actions A_i, the state transition probability function $P^{\pi}(s', N|s,a)$, and the reward function given by the sum of the original reward function $R(s'|s,a)$ and the pseudo-reward function $\overline{R}_i(s')$. The MAXQ learning algorithm has been proven to converge to π_r^*, the unique recursively optimal policy for MDP M and MAXQ graph H, where M = (S, A, P, R, P_0) is a discounted infinite horizon MDP with discount factor γ, and H is a MAXQ graph defined over subtasks $\{M_0 \dots M_n\}$.

3. MULTI-AGENT MAXQ ALGORITHM

The MAXQ decomposition of the Q-function relies on a key principle: the reward function for the parent task is the value function of the child task (see Equation 1). We show how this idea can be extended to joint-action values. The most salient feature of the extended MAXQ algorithm, which is proposed in this paper, is that the top level(s) (the level immediately below the root, and perhaps lower levels) of the hierarchy is (are) configured to store the completion function (C) values for joint (abstract) actions of all agents. The completion function $C^j(i, s, a^1, a^2 ... a^j ... a^n)$ is defined as the expected discounted reward of completion of subtask a^j by agent j in the context of the other agents performing subtasks $a^i, \forall i \in \{1, ..., n\}, i \neq j$.

More precisely, the decomposition equations used for calculating the projected value function V have the following form (for agent j)

$$V_t^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n) =$$

$$\begin{cases} max_{a^j} Q_t^j(i, s, a^1 \dots a^j \dots a^n) \; if \; composite(i) \\ \sum_{s'} P(s' \mid s, i) R(s' \mid s, i) \; if \; primitive(i) \end{cases}$$

$$Q_t^j(i, s, a^1 \dots a^j \dots a^n) = V_t^j(a^j, s) + C_t^j(i, s, a^1 \dots a^j \dots a^n) \quad (2)$$

at the highest (or lower than the highest as needed) level(s) of the hierarchy, where joint action values are being modeled, and a^j is the action being performed by agent j. Compare the decomposition in Equation 1 with Equation 2. Given a MAXQ hierarchy M for any given task, we need to find the highest level at which this equation provides a sufficiently good approximation of the true value. For both the AGV and the trash collection domain, the subtasks immediately below the root seem to be a good compromise between good performance and reducing the number of joint state action values that need to be learned.

To illustrate the multi-agent MAXQ algorithm, for the two-robot trash collection task, if we set up the joint action-values at only the highest level of the MAXQ graph, we get the following value function decomposition for $Agent1$:

$$Q_t^1(Root, s, NavT1, NavT2) = V_t^1(NavT1, s) + C_t^1(Root, s, NavT1, NavT2)$$

which represents the value of $Agent1$ doing task $NavT1$ in the context of the overall $Root$ task, when $Agent2$ is doing task $NavT2$. Note that this value is decomposed into the value of the $NavT1$ subtask itself and the completion cost of the remainder of the overall task. In this example, the multi-agent MAXQ decomposition embodies the heuristic that the value of $Agent1$ doing the subtask $NavT1$ is independent of whatever $Agent2$ is doing.

A recursive algorithm is used for learning the C values. Thus, an agent starts from the root task and chooses a subtask till it gets to a primitive action. The primitive action is executed, the reward observed, and the leaf V values updated. Whenever any subtask terminates, the $C(i,s,a)$ values are updated for all states visited during the execution of that subtask. Similarly, when one of the tasks at the level just below the root task terminates, the $C(i, s, a^1, \dots, a^n)$ values are updated according to the MAXQ learning algorithm.

4. THE AGV SCHEDULING TASK

Automated Guided Vehicles (AGVs) are used in flexible manufacturing systems (FMS) for material handling [1]. They are typically used to pick up parts from one location, and drop them off at another location for further processing. Locations correspond to workstations or storage locations. Loads which are released at the dropoff point of a workstation wait at its pick up point after the processing is over so the AGV is able to take it to the warehouse or some other locations. The pickup point is the machine or workstation's output buffer. Any FMS system using AGVs faces the problem of optimally scheduling the paths of AGVs in the system[7]. For example, a move request occurs when a part finishes at a workstation. If more than one vehicle is empty, the vehicle which would service this request needs to be selected. Also, when a vehicle becomes available, and multiple move requests are queued, a decision needs to be made as to which request should be serviced by that vehicle. These schedules obey a set of constraints that reflect the temporal relationships between activities and the capacity limitations of a set of shared resources.

The uncertain and ever changing nature of the manufacturing environment makes it virtually impossible to plan moves ahead of time. Hence, AGV scheduling requires dynamic dispatching rules, which are dependent on the state of the system like the number of parts in each buffer, the state of the AGV and the processing going on at the workstations. The system performance is generally measured in terms of the throughput, the online inventory, the AGV travel time and the flow time, but the throughput is by far the most important factor. In this case, the throughput is measured in terms of the number of finished assemblies deposited at

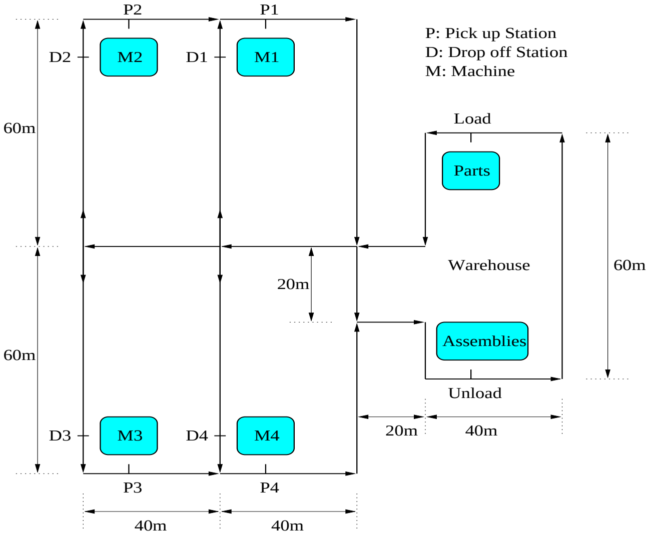


Figure 4: A multiple automatic guided vehicle (AGV) optimization task. There are four AGV agents (not shown) which carry raw materials and finished parts between the machines and the warehouse.

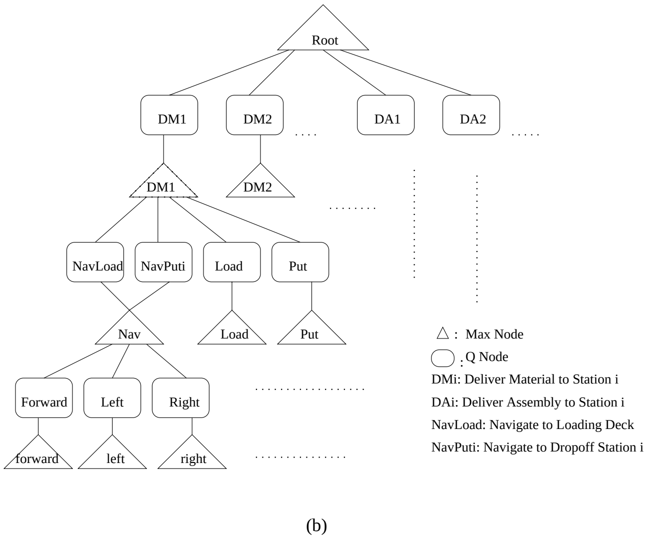

Root
DM1
DM2
DA1
DA2
DM1
DM2
NavLoad
NavPuti
Load
Put
Nav
Load
Put
Forward
Left
Right
forward
left
right
△ : Max Node
◯ : Q Node
DMi: Deliver Material to Station i
DAi: Deliver Assembly to Station i
NavLoad: Navigate to Loading Deck
NavPuti: Navigate to Dropoff Station i
(b)

Figure 5: MAXQ graph for the AGV scheduling task.

the unloading deck per unit time. Since this problem is analytically intractable, various heuristics and their combinations are generally used to schedule AGVs[6, 7]. However, the heuristics perform poorly when the constraints on the movement of the AGVs are reduced.

Previously, Tadepalli and Ok [16] studied a single-agent AGV scheduling task using "flat" average-reward reinforcement learning. However, the multi-agent AGV task we study is more complex. Figure 4 shows the layout of the system used for experimental purposes. Parts of type i have to be carried to drop off station at workstation i and the assembled parts brought back into the warehouse. The AGV travel is unidirectional (as the arrows show).

The termination predicate has been redefined to take care of the fact that the completion of certain tasks might depend on the occurrence of an event rather than just a state of the environment. For example, if we consider the $DM1$ subtask in the AGV problem (see Figure 5), the state of the system at the beginning of the subtask might be the same as that at the end, as the system is very dynamic. New parts continuously arrive at the warehouse, and the machines start and end work on parts at random intervals. Also, the actions of a number of agents affects the environment. This kind of discrete event model makes it necessary to have termination of subtasks to be defined in terms of events. Hence, a subtask terminates when the event associated with that subtask is triggered by the robot performing the subtask, for example DM1 subtask terminates when the *"unload of material 1 at drop off station of machine 1"* event occurs.

4.1 State Abstraction

The state of the environment consists of the number of parts in the pickup station and in the dropoff station of each machine, and whether the warehouse contains parts of each of the four types. In addition, each agent keeps track of its own location and state as a part of the state space. Thus, in the flat case, the size of the state space is ≈ 100 locations, 3 parts in each buffer, 9 possible states of the AGV (carrying Part1, ..., carrying Assembly1, ..., Empty), and 2 values for each part in the warehouse, i.e. $100 \times 4^8 \times 9 \times 2^4 \approx 2^{30}$, which is enormous. The MAXQ state abstraction helps in reducing the state space considerably. Only the relevant state variables are used while storing the completion functions in each node of the task graph. For example, for the *Navigate* subtask, only the location state variable is relevant, and this subtask can be learned with 100 values. Hence, for the highest level with 8 actions, i.e. DM1, ..., DM4, and DA1, ..., DA4, the relevant state variables would be $100 \times 9 \times 4 \times 2 \approx 2^{13}$. For the lower level state space, the action with the largest state space is Navigate with 100 values. This state abstraction gives us a compact way of representing the C functions, and speeds up the algorithm.

5. EXPERIMENTAL RESULTS

We first describe experiments in the simple two-robot trash collection problem, and then we will turn to the more complex AGV task.

5.1 Trash Collection Task

We first provide more details of how we implemented the trash collection task. In the single agent scenario, one robot starts in the middle of Room 1 and learns the task of picking up trash from T1 and T2 and depositing it into the Dump. The goal state is reached when trash from both T1 and T2 has been deposited in Dump. The state space here is the orientation of the robot (N,S,W,E), and another component based on its percept. We assume that a ring of 16 sonars would enable the robot to find out whether it is in a corner, (with two walls perpendicular to each other on two sides of the robot), near a wall (with wall only on one side), near a door (wall on either side of an opening), in a corridor (parallel walls on either side) or in an open area (the middle of the room). Thus, each room is divided into 9 states, and the corridor into 4 states. Thus, we have $((9 \times 3) + 4) \times 4$, or 124 locations for a robot. Also, the trash object from trash basket $T1$ can be at $T1$, carried with robot, or at $Dump$, and the trash object from trash basket $T2$ can be at $T2$, carried by robot, or at $Dump$. Thus the total number of environment states is $124 \times 3 \times 3$, or 1116 for the single agent case. Going to the two-agent case would mean that the trash can be at either $T1$ or $T2$, $Dump$, or carried by one of the two robots. Thus, in the flat case, the size of the state space would grow to $124 \times 124 \times 4 \times 4$, or $\approx 24 \times 10^4$.

The environment is fully observable given this state decomposition, as the direction which the robot is facing, in combination with the percept (which includes the room the agent is in) gives a unique value for each location. The primitive actions considered here are behaviors to find a wall in one of four directions, align with the wall on left or right side, follow wall, enter or exit door, align south or north in the corridor, or move in the corridor.

In the two-robot trash collection task, examination of the learned policy in Figure 6 reveals that the robots have nicely learned all three skills: how to achieve a subtask, what order to do them in, and how to coordinate with other agents. In addition, as Figure 7 confirms, the number of steps needed to do the trash collection task is greatly reduced when the two agents coordinate to do the task, compared to when a single agent attempts to carry out the whole task.

Learned Policy for Agent 1

```
root
  navigate to trash 1
      go to location of trash 1 in room 1
  pick trash 1
  navigate to bin
      exit room 1
      enter room 3
      go to location of dump in room 3
  put trash 1 in dump
end
```

Learned Policy for Agent 2

```
root
  navigate to trash 2
      go to location of trash 2 in room 1
  pick trash 2
  navigate to bin
      exit room 1
      enter room 3
      go to location of dump in room 3
  put trash 2 in dump
end
```

Figure 6: This figure shows the policy learned by the cooperative multi-agent MAXQ algorithm in the trash collection task.

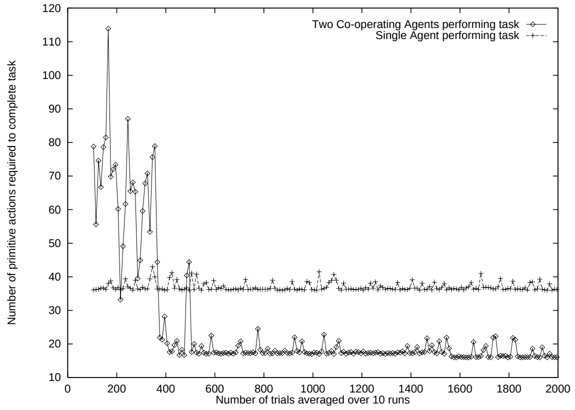


Figure 7: Number of actions needed to complete the trash collection task.

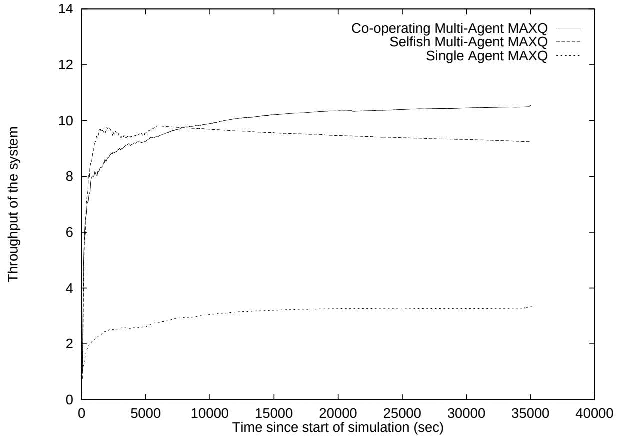


Figure 8: This figure shows that the cooperative multi-agent MAXQ approach outperforms both the selfish (non-cooperative) and single-agent MAXQ approaches when the AGV travel time is very much less compared to the assembly time.

5.2 AGV Domain

We now present detailed experimental results on the AGV scheduling task, comparing several learning agents, including a single agent using MAXQ, selfish multiple agents using MAXQ (where each agent acts independently and learns its own optimal policy), and the new co-operative multi-agent MAXQ approach. In this domain, there are four agents (each AGV is an agent).

The experimental results were generated with the following model parameters. The inter-arrival time for parts at the warehouse is uniformly distributed with a mean of 4 sec and variance of 1 sec. The percentage of Part1, Part2, Part3 and Part4 in the part arrival process are 20, 28, 22 and 30 respectively. The time required for assembling the various parts is normally distributed with means 15, 24, 24 and 30 sec for Part1, Part2, Part3 and Part4 respectively, and the variance 2 sec. Each experiment was conducted five times and the results averaged.

Figure 8 shows the throughput of the system for the three types of approaches. As seen in Figure 8, the agents learn a little faster initially in the selfish multi-agent method, but after some time, undulations are seen in the graph showing not only that the algorithm does not stabilize, but also that it results in sub-optimal performance. This is due to the fact

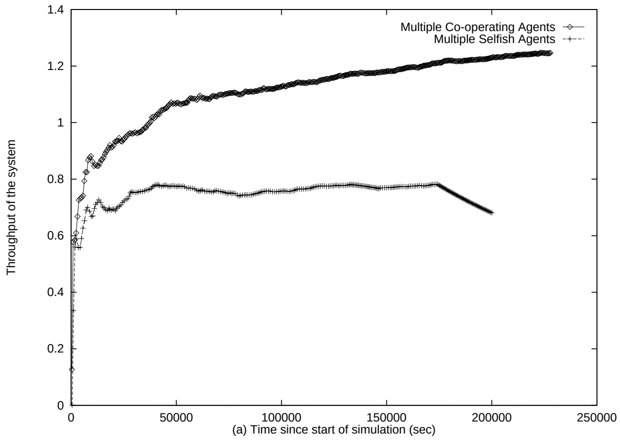


Figure 9: This figure compares the cooperative multi-agent MAXQ approach with the selfish (non-cooperative) MAXQ approach, when the AGV travel time and load/unload time is $\frac{1}{10^{th}}$ the average assembly time.

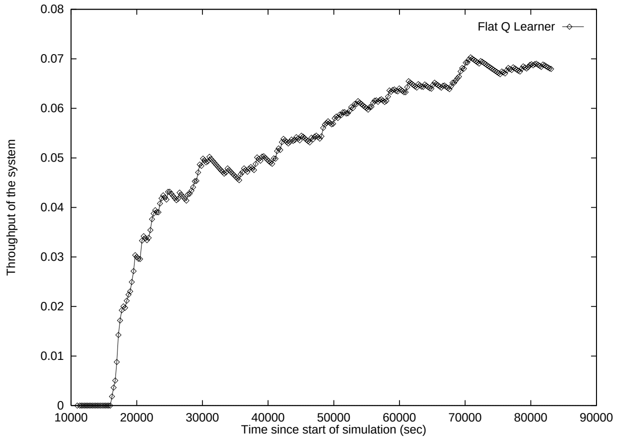


Figure 10: A flat Q-learner learns the AGV domain extremely slowly, showing the need for using a hierarchical task structure.

that two or more agents select the same action, but once the first agent completes the task, the other agents might have to wait for a long time to complete the task, due to the constraints on the number of parts that can be stored at a particular place. The system throughput achieved using the new cooperative multi-agent MAXQ method is significantly higher than the single agent or selfish multi-agent case. This difference is even more significant in figure 9, as when the agents have a longer travel time, the cost of making a mistake is greater.

Figure 10 shows results from an implementation of a single flat Q-Learning agent with the buffer capacity at each station set at 1. As can be seen from the plot on the left, the flat algorithm converges extremely slowly. The throughput at 70,000 sec has gone up to only 0.07, compared with 2.6 for the hierarchical single agent case. Figure 11 compares the cooperative multi-agent MAXQ algorithm with several well-known AGV scheduling rules, showing clearly the improved performance of the reinforcement learning method.

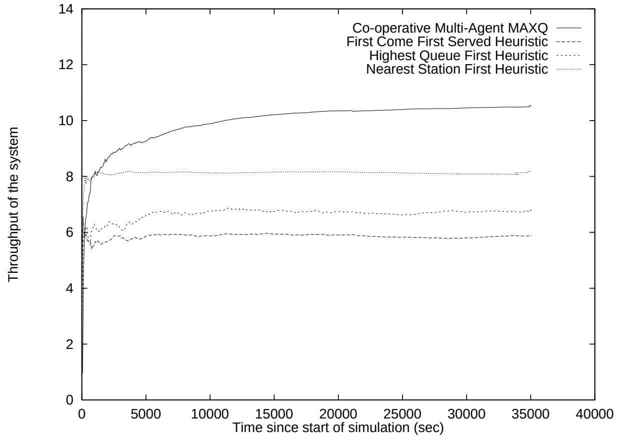


Figure 11: This plot shows the multi-agent MAXQ outperforms three well-known widely used (industrial) heuristics for AGV scheduling.

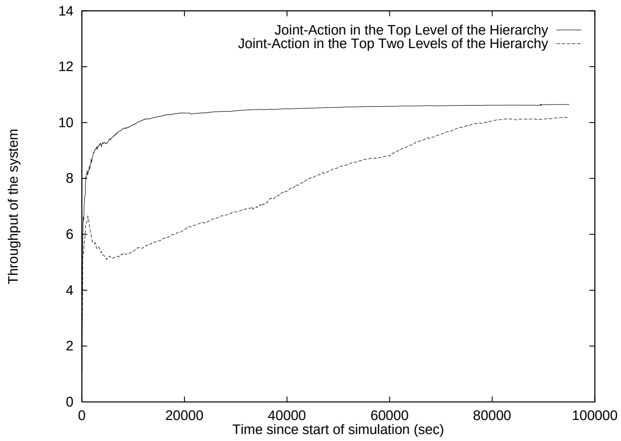


Figure 12: This plot compares the performance of the multi-agent MAXQ algorithm with joint actions at the top level vs. joint actions at the top two levels.

Finally, Figure 12 shows that when the Q-nodes at the top two levels of the hierarchy are configured to represent joint action-values, learning is considerably slower (since the number of parameters is increased significantly), and the overall performance is not better. The lack of improvement is due in part to the fact that the second layer of the MAXQ hierarchy is concerned with navigation. Adding joint actions does not help improve navigation because coordination is not necessary in this environment. However, it might turn out that adding joint actions in multiple layers will be worthwhile, even if convergence is slower, due to better overall task performance.

6. CONCLUSIONS AND FUTURE WORK

We described an approach for scaling multi-agent reinforcement learning by extending the MAXQ hierarchical reinforcement learning method to use joint action values. This decomposition relies on a key principle: the value of a parent task can be factored into the value of a subtask (which is independent of joint action values) and the completion cost (which does depend on joint action values). The effectiveness of this decomposition is most apparent in tasks where agents rarely interact in carrying out cooperative tasks (for example three robots that service a large building may rarely need to exit through the same door at the same time). Since interaction is modeled at an abstract level, coordination skills are learned rapidly. This approach can be easily adapted to constrained environments where agents are constantly running into one another (for example 10 robots in a small room all trying to leave the room at the same time) by using joint action-values at all levels of the hierarchy. However, this will result in a much larger set of action values that need to be learned, and consequently learning will be much slower.

We presented detailed experimental results from a complex AGV scheduling task, which show that the proposed hierarchical cooperative multi-agent MAXQ approach performed better than either the single agent or selfish (non-cooperative) multi-agent MAXQ methods. This novel approach of utilizing hierarchy for learning co-operation skills shows considerable promise as an approach that can be applied to other complex multi-agent domains. We primarily explored the use of the MAXQ hierarchical framework in our study, but we believe that other hierarchical methods could also be adapted to speed up multi-agent learning. The success of this approach depends of course on providing it with a good initial hierarchy.

7. ACKNOWLEDGMENTS

This work is supported by the Defense Advanced Research Projects Agency, DARPA contact No. DAANO2-98-C-4025.

8. REFERENCES

[1] R. Askin and C. Standridge. *Modeling and Analysis of Manufacturing Systems.* John Wiley and Sons, 1993.

[2] T. Balch and R. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Transactions on Robotics and Automation*, 14(6):1–15, 1998.

[3] R. Crites and A. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262, 1998.

[4] T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, volume 13. pages 227–303, 2000.

[5] J. Hu and M. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Fifteenth International Conference on Machine Learning*, pages 242–250, 1998.

[6] C. Klein and J. Kim. Agv dispatching. *International Journal of Production Research*, 34(1):95–110, 1996.

[7] J. Lee. Composite dispatching rules for multiple-vehicle AGV systems. *SIMULATION*, 66(2):121–130, 1996.

[8] M. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, 1994.

[9] M. Mataric. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, 1997.

[10] R. Parr. *Hierarchical Control and Learning for Markov Decision Processes.* PhD Thesis, University of California, Berkeley, 1998.

[11] M. L. Puterman. *Markov Decision Processes.* Wiley Interscience, New York, USA, 1994.

[12] P. Stone and M. Veloso. Team-partitioned, opaque-transition reinforcement learning. *Third International Conference on Autonomous Agents*, pages 86–91, 1999.

[13] T. Sugawara and V. Lesser. Learning to improve coordinated actions in cooperative distributed problem-solving environments. *Machine Learning*, 33:129–154, 1998.

[14] R. Sutton and A. Barto. *An Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA., 1998.

[15] R. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[16] P. Tadepalli and D. Ok. Scaling up average reward reinforcement learning by approximating the domain models and the value function. In *Proceedings of International Machine Learning Conference*, 1996.

[17] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337, 1993.

[18] G. Wang and S. Mahadevan. Hierarchical optimization of policy-coupled semi-markov decision processes. In *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999.

[19] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA., 1999.

Approximate Planning with Hierarchical Partially Observable Markov Decision Process Models for Robot Navigation

Georgios Theocharous
theochar@cse.msu.edu
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48823

Sridhar Mahadevan
mahadeva@cs.umass.edu
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract— **We propose and investigate a planning framework based on the Hierarchical Partially Observable Markov Decision Process model (HPOMDP), and apply it to robot navigation. We show how this framework can be used to produce more robust plans as compared to flat models such as Partially Observable Markov Decision Processes (POMDPs). In our approach the environment is modeled at different levels of resolution, where abstract states represent both spatial and temporal abstraction. We test our hierarchical POMDP approach using a large simulated and real navigation environment. The results show that the robot is more successful in navigating to goals starting with no positional knowledge (uniform initial belief state distribution) using the hierarchical POMDP framework as compared to the flat POMDP approach. In addition, the HPOMDP model allows the robot to efficiently model and navigate large scale environments.**

I. Introduction

One of the most difficult challenges in programming autonomous mobile robots is to enable them to act reliably in large, and perceptually aliased environments (e.g., in an office environment, two corridors or intersections can look alike). Animals use path integration and other long-term memory structures to solve this problem. Humans can easily follow directions that require remembering past decisions (e.g., take a right at the first stop sign, and then a left at the next stop sign). Additionally, to handle large scale environments, humans and animals seem to use multi-resolution representations. For example, in driving from East Lansing, Michigan to Amherst, Massachusetts we use both local city maps as well as interstate maps. An abstract representation of a navigation task not only reduces the size of the representation (state space) but also reduces perceptual aliasing. For example, once we are on the highway we don't care where exactly we are, but rather follow the road until we see the desired exit sign.

An autonomous mobile robot that has to navigate in an indoor office environment needs to know where it is at each point in time so that it can make appropriate decisions (e.g., at this T-junction, should I turn left or right?). Unfortunately indoor office environments are perceptually aliased. Even if the robot had perfect sensors, which is usually not the case, it will not be able to distinguish in which place it is in its environment simply by observing its surroundings since many locations look alike. One way to know exactly where the robot is in the environment is to keep track of odometry. Unfortunately, this also requires noise-free dead-reckoning and knowledge of the starting position of the robot. One solution to the above problems is to model the navigation task as a partially observable Markov decision process (POMDP) [1], [2],[3]. In a POMDP representation of robot navigation, states correspond to fixed length regions, e.g. each square meter in the environment is modeled as four states of the POMDP (one for each robot orientation). The robot can take actions such as "go-forward", "turn-left", and "turn-right". In each state, the robot can observe features such as walls, openings, and doors on the front, left, back and right side. Unfortunately, modeling a real office environment in this manner requires thousands of states and exact POMDP planning methods based on dynamic programming are intractable. Also, heuristic planning methods (e.g. Q-MDP or MLS) are not practical since they are usually effective in belief states with unimodal low entropy distributions [1].

In this paper we investigate a multi-resolution model called Hierarchical Partially Observable Markov Decision Processes (HPOMDPs), which was recently proposed in [4]. We describe a general hierarchical planning algorithm using HPOMDPs and apply the framework to corridor navigation to produce and execute hierarchical plans. At an abstract level we find mappings from abstract states (e.g whole corridors) to macro-actions (e.g, go down the corridor) and at lower resolutions we produce local plans, such as "exit a corridor" or reach a goal location within a corridor. Our experiments show that there is less uncertainty associated with the belief states defined over higher levels of the HPOMDP hierarchy. As a result, heuristic POMDP solutions allow the robot to reach selected destinations of its environment starting from a state of no knowledge of its original starting location. In addition, the hierarchical nature of our model allows us to represent larger environments more efficiently since we don't have to represent all relationships between states in the environment at a uniform resolution. Also, hierarchical plans are computed much faster since they can be constructed from precomputed local plans. In the rest of the paper, we describe the Hierarchical Hidden Markov model (HHMM), its extension to HPOMDP, how it can be used for planning, its application to robot navigation, and experiments using large simulated and real environments.

II. Hierarchical Hidden Markov Models

The HPOMDP model was derived from the HHMM [5] model with the addition of primitive and abstract actions, and reward functions. The HHMM generalizes the standard hidden Markov model (HMM) [6], by allowing hidden states to represent stochastic processes themselves. An HHMM is visualized as a tree structure (see Figure 1) in which there are three types of states, production states (leaves of the tree) which emit observations, and internal states which are (unobservable) hidden states that represent entire stochastic processes. Each production state is associated with an observation vector which maintains distribution functions for each observation defined for the model. Each internal state is associated with a horizontal transition matrix, and a vertical transition vector. The horizontal transition matrix of an internal state defines the transition probabilities among its children. The vertical transition vectors define the probability of an internal state to activate any of its children. Each internal state is also associated with a child called an *end-state* which returns control to its parent. The end-states do not produce observations and cannot be activated through a vertical transition from their parent.

Figure 1 shows a graphical representation of an example HHMM. The HHMM produces observations as follows:
1. If the current node is the root, then it chooses to activate one of its children according to the vertical transition vector from the root to its children.
2. If the child activated is a product state, it produces an observation according to an observation probability output vector. It then transitions to another state within the same level. If the state reached after the transition is the end-state, then control is returned to the parent of the end-state.
3. If the child is an abstract state then it chooses to activate one of its children. The abstract state waits until control is returned to it from its child end-state. Then it transitions to another state within the same level. If the resulting transition is to the end-state then control is returned to the parent of the abstract state.

III. Hierarchical Partially Observable Markov Decision Process Models

An extension to the HHMM model which includes actions and rewards was proposed in [4], which also showed how such models could be learned from sequences of observations. Here we extend the definition to include multiple entry points into abstract states which we call *entry-states*, and multiple *exit-states* (previously defined as end-state). Figure 3 shows a sample HPOMDP with multiple entry/exit states for corridor environments. The exit and entry states represent the spatial borders of each corridor abstract state. If we only had single exit and entry states, the consequences of primitive actions would not be modeled correctly. For example, an exit from the east side of a corridor would have a non-zero transition probability to an adjacent state of the corridor at the west side. On the other hand, we could have two abstract states representing each corridor (one for each direction). However, having two abstract states for the same spatial locations would mean that the abstract states share children which is not allowed by our model (since it would violate our concept of spatial abstraction). We formally describe the elements of an HPOMDP below:

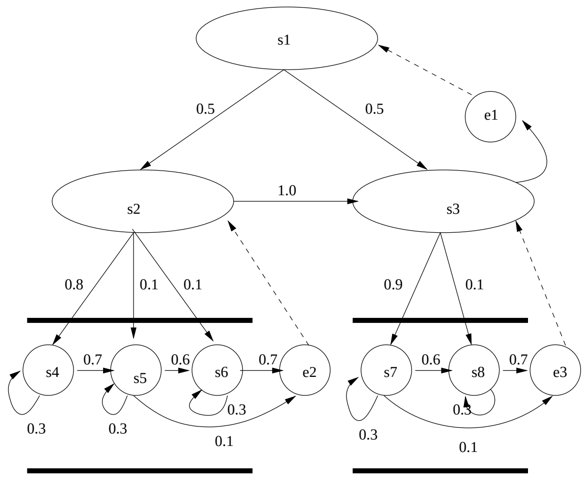


Fig. 1. An example hierarchical hidden Markov model (HHMM) which represents two adjacent corridors. Only leaf (production) states (s4, s5, s6, s7, and s8) produce observations. s2 and s3 represent corridors.

- S denotes the set of states. Unlike HHMMs, we have an additional type of state called entry states.
  – *Product states* which produce observations.
  – *Exit-states* which return control to their parent abstract state when entered.
  – *Entry-states* which belong to abstract states and when entered activate the children of the abstract state they are associated with.
  – *Abstract states* which group together *product*, *entry*, *exit*, and other *abstract* states. We say C^s to denote the product children, and entry states of other abstract states which are children of abstract state s. We say X^s to denote the set of exit states that belong to s and N^s to denote the set of entry-states that belong to abstract state s.
- A denotes the set of primitive actions. For example, in robot navigation the actions could be *go-forward* one meter, *turn-left* 90 degrees, and *turn-right* 90 degrees. Primitive actions are only taken at product states.
- $T(s'|s,a)$ denotes the horizontal transition probabilities, for primitive actions on *product* states, where s is the current product state and s' is the next state which can be either an *entry-state* or a *product-state.*
- $TE(s'|s_x,a)$ denotes the horizontal transition probabilities, for abstract states, where s is the current abstract state, and x is *the exit-state* that abstract state s was ex-

ited from under primitive action a, s' is the next state which can be either an *entry-state* or a *product-state*.

- $V(i|s_n)$ denotes the probability that entry state n which belongs to abstract state s will activate child state i. Child i could be a product state or an entry-state of some abstract state (other than s).
- Z denote the set of observations.
- $P(z|s,a)$ denotes the probability of observation z in product state s after action a has been taken.
- $R(s,a)$ denotes an immediate reward function defined on the product states.

IV. Robot Navigation in Corridor Environments

Figure 2 shows a section of the physical environment at the MSU engineering building. An immediate observation is that corridors are quite long (more than 1000 ft.) and locations within the corridors do not have distinguishing characteristics for a robot to localize. However, this problem can be alleviated if we treat whole corridors as single states and thus abstract away information that may be unnecessary for most navigation tasks. The reason is that for most navigation tasks a robot just needs to know in which corridor it is in, rather than the precise corridor location, and therefore it can often ignore location uncertainty that arises within corridors.

Fig. 2. The corridor on the left is a section of the MSU engineering building. Pavlov, the robot on the right, uses 16 sonar sensors to navigate the whole engineering building.

Figure 3 shows an HPOMDP model for robot navigation and figure 4 shows the equivalent flat model. Both models are used to represent corridor environments. For every two meters there are four product states (one for each direction). In the HPOMDP model, whole corridors are represented with a single abstract state, which in addition to the product states contains two entry-states and two exit-states for the entry and exit points of the corridor. Junction nodes are represented with four product states as shown in Figure 3. In the HPOMDP model for corridor environments, we use the term product level to denote the set of all product states, and abstract level denotes all abstract states plus the product states that belong to junctions.

A multi-resolution representation is intuitive for many reasons. First of all it allows us to scale up to larger environment since we do not need to represent all relationships among product states. Second, for many navigation tasks the robot does not need to know where it is in a corridor but rather that it is in some corridor which it has to traverse. Third, it offers us the capability of learning local policies for the different abstract states which we can reuse for any global plan at the abstract level. And finally, as shown in Figure 5, the robot's belief as to where it is at the abstract level is less uncertain from its belief as to where it is at a global product level. In fact, the entropy plot reveals a number of characteristics that motivate our planning algorithm:

1. It is obvious from the plot that uncertainty is less at an abstract level.
2. Uncertainty reduces dramatically at junctions both at the global and abstract levels.
3. At the global level uncertainty increases as the robot traverses corridors where all perceptions are usually the same, but at an abstract level it does not increase.
4. Overall uncertainty decreases as the robot traverses a series of corridors.

The above characteristics suggest that a good approximate algorithm could be one that treats the problem at the abstract level as completely observable. In addition, a successful algorithm would be one that always takes the

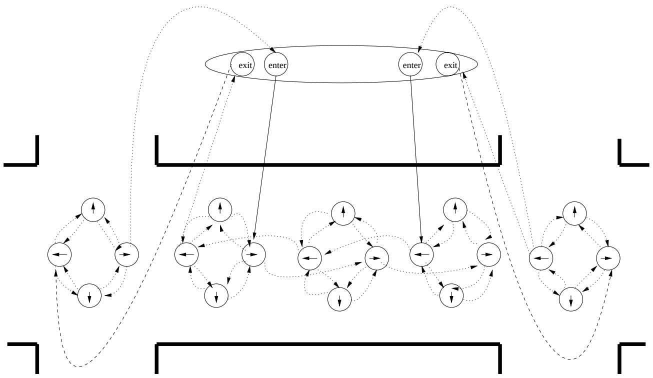


Fig. 3. This HPOMDP is used to model corridor environments. The dotted arrows from product states show non-zero transition probabilities for primitive actions. The black vertical arrows show non-zero vertical activations of product states from the entry states. The dashed arrows from exit states show non-zero transition probabilities from exit-states to adjacent states of the abstract state they are associated with.

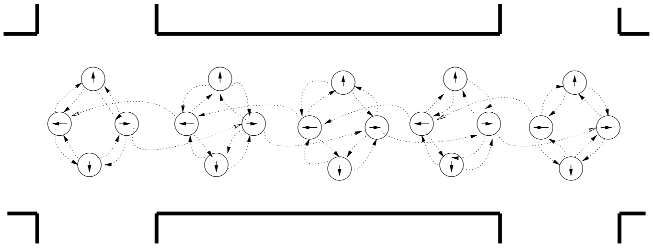


Fig. 4. This is the equivalent flat POMDP of the Hierarchical POMDP shown in Figure 3. In this model we only represent the product states of the hierarchical model. Global transition matrices for each action are automatically computed from the hierarchical model by considering all the possible paths between the product states under the primitive action.

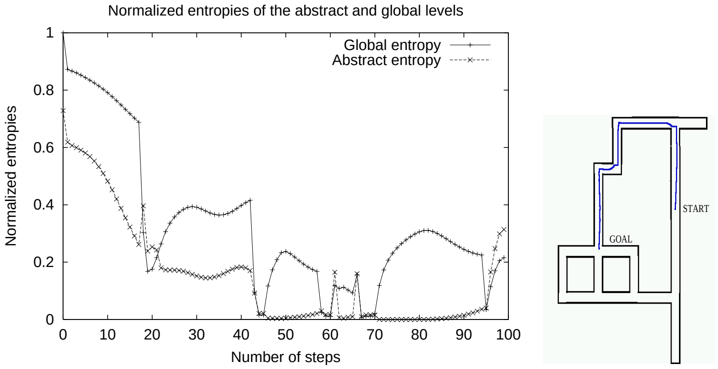


Fig. 5. The plot on the left shows the normalized entropies of the belief of the robot location at the global product level and at the abstract level. The entropy plot was created during a robot run on the Nomad 200 simulator whose trace is shown on the right.

robot from every entry-state of a corridor to the exit-state at the other exit point of the corridor so that overall uncertainty is gradually reduced. In the next section we formally present such an algorithm which is based on the theory of Markov decision processes (MDPs).

V. Planning and Execution Algorithm

A. Planning

To find a completely observable plan at an abstract level we treat the product states and the entry states (of the abstract states) at an abstract level as the states of an MDP. The set of actions is the set of primitive actions A applicable on product states and the set of macro-actions for each entry state n of every abstract state s, which we define to be M^{s_n}, and are responsible for taking the agent out of the abstract state s, or to some child state of the abstract state. For our corridor environment we defined at the most two macro-actions to be available at each entry state, the one that will take the robot to the opposite exit-state and an additional macro action if the goal belongs to the same abstract state as the entry state. The reason is that we always want the robot to traverse corridors such that the uncertainty is gradually reduced. We can design macro-actions for each corridor by either hard-coding them or by computing them using the MDP framework where we only give a goal reward. Figure 6 shows different macro-actions for corridors.

Given these macro-actions we need to evaluate them for each entry state s_n. Equation 1 gives us the transition probability from entry-state $s_n \in N^s$ to an adjacent state s' of the abstract state s under some macro-action μ. An exit from state s can occur from any exit state x under the primitive action $\pi^s_\mu(s_x)$, where π^s_μ defines the policy of macro action μ on all states i that are either in the C^s or X^s sets. If i is a product state then $\pi^s_\mu(i)$ is a primitive action. If i is an entry state then $\pi^s_\mu(i)$ is a macro action that belongs to $M^{s'_i}$, where s' is the abstract state associated with entry state i.

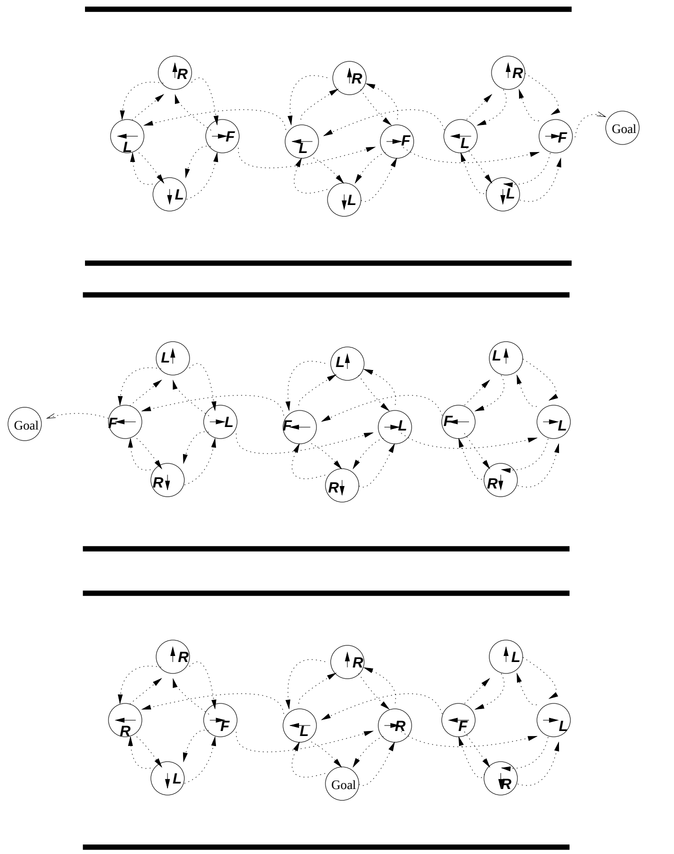


Fig. 6. F means go forward, R means turn right and L means turn left. The top figure is a policy for exiting the East side of the corridor with the go-forward action, the middle figure is a policy for exiting the West side of the corridor with the go-forward action, and the bottom figure is a policy for reaching a particular location within the corridor.

$$T(s'|s_n,\mu) = \sum_{\forall s_x \in X^s} \left[\sum_{\forall i \in C^s} V(i|s_n) TD(s_x|i,\mu) \right] TE(s'|s_x, \pi^s_\mu(s_x)) \tag{1}$$

The term $TD(x|i,\mu)$ is the expected discounted transition probability for exiting the parent of state i starting from i and can be calculated by solving a set of linear equations defined in Equation 2.

$$\begin{aligned} TD(s_x|i,\mu) =& T(s_x|i,\pi^s_\mu(i)) + \\ & \gamma \sum_{j \in C^s} T(j|i,\pi^s_\mu(i)) TD(s_x|j,\mu) \end{aligned} \tag{2}$$

In a similar manner we can calculate the reward R that will be gained when an abstract action μ is executed at an entry state s_n as shown in Equation 3.

$$R(s_n,\mu) = \sum_{i \in C^s} V(i|s_n) RD(i,\mu) \tag{3}$$

where the term $RD(i,\mu)$ can be calculated by solving a set of linear equations defined in Equation 4.

$$\begin{aligned} RD(i,\mu) =& R(i,\pi^s_\mu(i)) + \\ & \gamma \sum_{j \in C^s} T(j|i,\pi^s_\mu(i)) RD(j,\mu) \end{aligned} \tag{4}$$

Given any arbitrary HPOMDP and all the macro-actions M^{s_n} of every entry state of every abstract state we can calculate all the reward and transition models at the highest level by starting from the lowest levels and and moving toward the top. Discussion of macro-action generation and usage can be found in [7] and also in the more general theory of the *options* framework [8]. The major differences in this paper is that first of all applicability of each macro-action is defined by the abstract states and exit-states which can be though of as special cases of initiation and termination conditions in the *options* framework. And second, when it comes to the construction of global plans we compute them over a selected number of states (states at the abstract level).

Once the transition and reward models are calculated we can construct an abstract plan among product and entry-states at the abstract level. To find a plan, we first compute the optimal values of states by solving a set of linear equations defined by the Bellman equation (as shown below).

$$\begin{aligned} V^*(s_1) = & \max_{a \in A | a \in M^{s_1}} (R(s_1,a) + \\ & \gamma \sum_{s_2}^{C^{root}} T(s_2|s_1,a) V^*(s_2)) \end{aligned} \tag{5}$$

where a is either a primitive action or a macro action. We can then associate the best action for each state (or optimal policy π^*) greedily as shown in Equation 6.

$$\begin{aligned} \pi^*(s_1) =& \text{argmax}_{a \in A | a \in M^{s_1}} (R(s_1,a) + \\ & \gamma \sum_{s_2}^{C^{root}} T(s_2|s_1,a) V^*(s_2)) \end{aligned} \tag{6}$$

B. Execution

To execute the abstract plan we keep track of an "abstract" probability distribution over states at the abstract level. An efficient way to calculate the abstract belief state is to first calculate a global transition matrix among all product states under the primitive actions, and use that transition matrix to update a global belief state among product states after every action and observation as shown in Equation 7.

$$b'(s) = \frac{1}{P(z|a,b)} P(z|s,a) \sum_{s'=1}^{|S|} P(s|a,s') b(s') \tag{7}$$

where b' is the next belief state, $P(z|s,a)$ is the probability of perceiving observation z when action a was taken in state s. In the navigation case, we have 16 observations which indicate the probability of a wall and opening on the four sides of the robot. The observations are extracted from trained neural nets where the inputs are local occupancy grids constructed from sonar sensors and outputs are probabilities of walls and openings [9]. $P(s|a,s')$ is the probability of going from state s' to state s under action a and b is the previous belief state. Using the flat belief state b we can calculate the abstract belief state using Equation 8.

$$\begin{aligned} & B(s) = b(s) \text{ if s is a product state} \\ & B(s) = \sum_{c(s,i)}^{C^s} b(c(s,i)) \text{ if s is an abstract state} \end{aligned} \tag{8}$$

After every action and observation we update the abstract belief state. If the most likely state s is a product state, then we execute the action associated with s. If the most likely state s is an abstract state then we find the most likely entry state s_n by summing the beliefs of the children of the abstract state that have a non-zero vertical transition from the entry state s_n. We then execute the appropriate macro-action until abstract state s is no longer the most likely state. There are different ways to execute the macro-action based on heuristic POMDP solutions that assume that the underlying model is completely observable [1]. One way is to use the most likely state heuristic where we initialize the local probability distribution over the children of s according the vertical activation from entry-state s_n.

VI. Experimental Results

We applied the above algorithm to a large scale robot navigation environments shown in Figure 7. For the small environment the total number of states is 457 for POMDPs and 532 for HPOMDPs. For the large environment the total number of states is 1285 for POMDPs and 1385 for HPOMDPs. In both environments the task of the robot was to get to the four-way starting from a uniform initial global belief state. To formulate this as a goal planning problem we gave the robot a -1 reward for every action executed except for $+1$ reward for actions leading to the goal state. In this way the robot would learn to choose shorter routes so as to minimize the cost. For the simulated environments we tested the algorithm by starting the robot from 30 different random locations and observed the number of steps it took to reach the goal. We also ran the robot in the real environment starting it from 23 locations and using the HPOMDP model. We did not use the flat POMDP model in the real environment due to its large size, long planning times, and the limited computing power on the real robot. Table I shows the results for the different environments and the models used. It is clear that the HPOMDP models give 100 % success in simulation and high success rate in the real world. Unsuccessful trials in the real world were due to failures of the behavior-based layer of the navigation architecture that was responsible for executing the primitive actions (e.g., crash into a wall).

We also measured the time it took for the robot to construct a plan for the different environments and the differ-

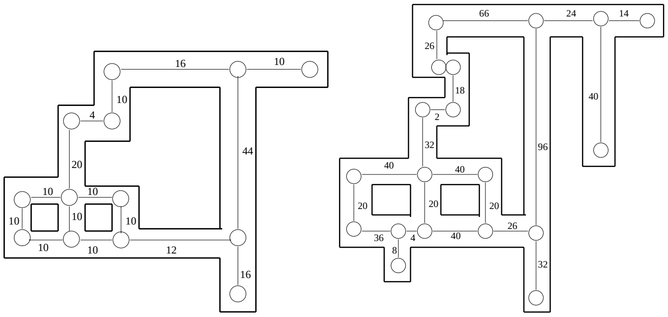


Fig. 7. The figure shows a small environment on the left that was used with the Nomad 200 simulator only and a large environment on the right that was used both in simulation and in the real world. The number next to the edges is the distance between the nodes in meters.

Envir.	Model	Goal %	Steps	Abstract Decisions
small sim.	HPOMDP	100	87.1	19.8
small sim.	POMDP	40	96.2	96.2
large sim.	HPOMDP	100	176.9	20.2
large sim.	POMDP	50	184	184
large real	HPOMDP	86.6	176.5	23.5

TABLE I

The table shows percentage of successful trials where the goal was reached, the average number of primitive steps, and the average number of decision at the abstract level starting from an initial maximum entropy belief state (uniform distribution).

ent models. The results are summarized in table II where for the HPOMDP model it takes significantly less time to compute a plan.

Environment	Model	Planning Time (sec)
small	HPOMDP	4.22
small	POMDP	25.05
large	HPOMDP	38.16
large	POMDP	409.92

TABLE II

The table shows the planning times for the different environments and models.

VII. Conclusions and Future Directions

Our hierarchical approach to robot navigation using the HPOMDP model seems to outperform previous flat POMDP based representations. The algorithm presented is highly successful in taking the robot to the goal starting from no positional knowledge. Also, the hierarchical approach allows us to scale up to larger domains both in terms of time and space. Abstract plans are constructed much faster than flat plan representations, and the amount of space needed to represent the model is a lot less since not all relationships between product states are represented.

The high success rate in taking the robot to its goal is due to two key reasons. First, the robot takes decisions at an abstract level where there are less number of states to consider and as a result less uncertainty. And second, when the robot enters a corridor it chooses the macro-action to move to the opposite exit of the corridor which results in reduction of the entropy. Even though going to the opposite end of the corridor may not be optimal if the environment was completely observable, in POMDPs it makes sense because it serves as an information gathering action for places where uncertainty is high (which we know to be inside long corridors).

In addition to the fact that this algorithm is successful for robot navigation in corridor environments it may have application to domains other than robotics. Currently, it is up to the designer to group together high entropy states. One future direction would be to infer the structure of the model as well, where the goal would be to group together states where uncertainty rises. Another future direction would be to apply the model and algorithm to deeper hierarchies with more than 2 levels. And finally, a future direction would be the study of this model with respect to exact POMDP solutions. A hierarchical approach for planning and execution in POMDPs that is closer to exact POMDP solutions is described in [10].

Acknowledgments

This research is supported in part by a DARPA MARS grant DABT63-99-1-0014, and an NSF Knowledge and Distributed Intelligence (KDI) Award ECS-9873531.

References

[1] Sven Koenig and Reid Simmons, "A Robot Navigation Architecture Based on Partially Observable Markov Decision Process Models," in *Artificial Intelligence Based Mobile Robotics:Case Studies of Successful Robot Systems*, D. Kortenkamp, R. Bonasso, and R. Murphy, Eds., pp. 91–122. MIT press, 1998.

[2] Hagit Shatkay and Leslie Kaelbling, "Learning Topological Maps with Weak Local Odeometric Information," in *Proc. of the International Joint Conference on Artificial Intelligence*, 1997.

[3] I. Nourbakhsh, R. Powers, and S. Birchfield, "Dervish: An office-navigation robot," in *AI Magazine 16(2):53-60.* 1995.

[4] Georgios Theocharous, Khashayar Rohanimanesh, and Sridhar Mahadevan, "Learning hierarchical partially observable markov decision processes for robot navigation," in *IEEE Conference on Robotics and Automation (ICRA)*, Seoul, Korea, 2001.

[5] Shai Fine, Yoram Singer, and Naftali Tishby, "The Hierarchical Hidden Markov Model: Analysis and Applications," *Machine Learning*, vol. 32, no. 1, pp. 41–62, July 1998.

[6] Lawrence Rabiner, "Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," in *Proceedings of the IEEE*, February 1989, vol. 77.

[7] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier, "Hierachical solution of Markov decision processes using macro-actions," in *UAI-98*, 1998.

[8] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, pp. 112:181–211, 1999.

[9] Sridhar Mahadevan, Georgios Theocharous, and Nikfar Khaleeli, "Fast concept learning for mobile robots," in *Autonomous Robots Journal, vol. 5, pp. 239-251.* 1998.

[10] J. Pineau, N. Roy, and S. Thrun, "A hierarchical approach to POMDP planning and execution," in *Workshop on Hierarchy and Memory in Reinforcement Learning (ICML 2001)*, Williams College, MA, June 2001.

Decision-Theoretic Planning with Concurrent Temporally Extended Actions

Khashayar Rohanimanesh
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1226
khash@cse.msu.edu

Sridhar Mahadevan
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1226
mahadeva@cse.msu.edu

Abstract

We investigate a model for planning under uncertainty with temporally extended actions, where multiple actions can be taken concurrently at each decision epoch. Our model is based on the *options* framework, and combines it with factored state space models, where the set of options can be partitioned into classes that affect disjoint state variables. We show that the set of decision epochs for concurrent options defines a semi-Markov decision process, if the underlying temporally extended actions being parallelized are restricted to Markov options. This property allows us to use SMDP algorithms for computing the value function over concurrent options. The concurrent options model allows overlapping execution of options in order to achieve higher performance or in order to perform a complex task. We describe a simple experiment using a navigation task which illustrates how concurrent options results in a faster plan when compared to the case when only one option is taken at a time.

1 Introduction

In our everyday life, our brain is constantly planning and executing concurrent (parallel) behaviors. For example, when we are driving, in parallel we visually search for road signs, while we may be talking to a passenger. Or when walking toward our car in a parking lot or our office, we may simultaneously reach for our keys, while continuing to talk on a cell-phone and navigating through the environment. Parallel execution of behaviors is sometimes useful in performing a task more quickly (e.g., the parking lot example). In other situations, the nature of the task requires that multiple behaviors run concurrently and cooperatively in order to perform the task (in the driving example, we have to both look at the road and navigate the car simultaneously). In this paper, we investigate a model for planning with concurrent behaviors. We adopt the theoretical framework of *options* (Sutton et al., 1999) to model temporally extended actions, since it is both a well-developed rigorous framework that addresses planning under uncertainty with temporally extended actions, and it allows *looking inside* behaviors to improve composition of temporally extended actions.

Previous work on decision-theoretic concurrent planning seems to be largely restricted to work focusing on combining *primitive* actions, ranging from planning in multi-dimensional vector action spaces (Cichosz, 1995) to planning with multiple simultaneous MDPs (Singh & Cohn, 1998), where the composite state space is the cross product of the state spaces of each individual MDP and the action set is a proper subset of a multi-dimensional primitive action space. In these models, each decision epoch is fixed and equal to single step execution. Our work differs in that we address planning with a set of parallel temporally extended actions that may not terminate at the same time, which makes the problem more challenging. We also exploit the fact that in many real world problems, the set of options can be factored into those that affect disjoint state variables. This factoring greatly reduces the complexity of planning with multi-dimensional composite state and action spaces (Boutilier et al., to appear).

In this paper, we address planning with a set of concurrent options, assuming that they do not compete for a shared resource (in the parking lot example, the option of reaching for the car key and the option of walking affect different portions of the composite state space). We present a navigation task involving moving through rooms using keys to open locked doors to illustrate how the concurrent options model facilitates faster planning. Our experiments show that the concurrent options model improves performance compared to the sequential case when only one behavior at a time can be executed.

The rest of the paper is organized as follows. In section 2, we will briefly overview the option framework. In section 3 we define the concurrent options model in detail. In section 4 we will present a computational problem and the performance results of planning using the concurrent options model. Section 5 outlines some problems for future research.

2 Options

Options are a generalization of primitive actions that include temporally extended courses of action in the context of reinforcement learning (Sutton et al., 1999). Options consist of three components: a policy $\pi : S \times A \rightarrow [0,1]$, a termination condition $\beta : S \rightarrow [0,1]$, and an initiation set $I \subseteq S$, where I denotes the set of states $s \in S$ in which the option can be initiated. Note that we can restrict the scope of application of a particular option by controlling the initiation set and the termination condition. For any state s, if option π is taken, then primitive actions are selected based on π until it terminates according to β. An option O is a *Markov option* if its policy, initiation set and termination condition depend stochastically only on the current state $s \in S$. An option O is a *semi-Markov option* if its policy, initiation set and termination condition are dependent on all prior history since the option was initiated. As an example, the option *exit-room* in which states are the different locations in the room is a Markov option, since for a given location, we know in what direction we should move independent of how we reached that location. Given a set of options O, let O_s denote the set of options in O that are available in each state $s \in S$ according to their initiation set. O_s resembles A_s in the standard reinforcement learning framework, in which A_s denotes the set of primitive (single step) actions. Similarly, we introduce *policies over options*. For a decision epoch d_t, the Markov policy over options $\mu : S \times O \rightarrow [0,1]$ selects an option $o_t \in O$, according to the probability distribution $\mu(s_t, .)$. The option o_t is then initiated in s_t until it terminates at a random time $t+k$ in some state s_{t+k} according to the termination condition, and the process repeats in s_{t+k}. For an option $o \in O$, and for any state $s \in S$, let $\varepsilon(o, s, t)$ denote the event of o being initiated in state s at time t. The total discounted reward accrued by executing option o in any state $s \in S$ is defined as:

$$r_s^o = E\{r_{t+1} + \gamma r_{t+2} + ... + \gamma^{k-1} r_{t+k} \mid \varepsilon(o,s,t)\} \quad (1)$$

where $t+k$ is the random time at which o terminates. Also let $p^o(s, s', k)$ denote the probability that the option o is initiated in state s and terminates in state s' after k steps. Then

$$p_{ss'}^o = \sum_{k=1}^{\infty} p^o(s, s', k)\gamma^k \quad (2)$$

Given the reward and state transition model of option o, we can write the Bellman equation for the value of a general policy μ as

$$V^\mu(s) = \sum_{o \in O_s} \mu(s,o) \left[r_s^o + \sum_{s'} p_{ss'}^o V^\mu(s') \right] \quad (3)$$

Similarly we can write the "option-value" Bellman equation for the value of an option o in state s as

$$Q^\mu(s,o) = r_s^o + \sum_{s'} p_{ss'}^o \sum_{o' \in O_{s'}} \mu(s',o') Q^\mu(s',o') \quad (4)$$

and the corresponding *optimal* Bellman equations are as follows:

$$V_O^*(s) = \max_{o \in O_s} \left[r_s^o + \sum_{s'} p_{ss'}^o V_O^*(s') \right] \quad (5)$$

$$Q_O^*(s,o) = r_s^o + \sum_{s'} p_{ss'}^o \max_{o' \in O_{s_{t+k}}} Q^*(s',o') \quad (6)$$

We can use *synchronous value iteration* (SVI) to compute $V_O^*(s)$ and $Q_O^*(s,o)$, which iterates the following step for every state $s \in S$:

$$V_t(s) = \max_{o \in O_s} \left[r_s^o + \sum_{s'} p_{ss'}^o V_{t-1}(s') \right] \quad (7)$$

$$Q_t(s,o) = r_s^o + \sum_{s'} p_{ss'}^o \max_{o' \in O_{s'}} Q_{t-1}(s',o') \quad (8)$$

Alternatively, if the option model is unknown, we can estimate $Q_O^*(s,o)$ using SMDP *Q-learning*, by doing sample backups after the termination of each option o, which transitions from state s to s' in k steps with cumulative discounted reward r:

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left[r + \gamma^k \max_{o' \in O_{s'}} Q(s',o') - Q(s,o) \right] \quad (9)$$

3 Concurrent Options

Definition: Let $o \equiv < I, \pi, \beta >$ be an option with state space S_o governed by the set of state variables $W_o = \{w_1^o, w_2^o, ..., w_{n_o}^o\}$. Let $\varphi_o \subset W_o$ denote the subset of state variables that evolve by some other processes (e.g. other options) and independent of o, and let $\Omega_o = W_o - \varphi_o$ denote the subset of state variables that evolve solely based on the option o. There is no explicit restriction on the initiation set, policy and termination condition with respect to the state space S_o. We refer to the class of options with this property as *partially-factored* options.

As an example, consider the task of delivering parts to a set of machines in a factory environment. The agent has to load up a part from the inventory load station and deliver it to a particular machine. We may define the options *deliver-part* and *load-part* with the set of state variables $W = W_{deliver\_part} = W_{load\_part} = \{position, part\_ready\}$ denoting the position of the agent and whether or not a part is available at the inventory load station, respectively. We may also define an *inventory* option with state variable $W_{inventory} = \{part\_ready\}$, which is set or reset whenever a part is ready to load, or not available. It is clear from this example that even though the initiation set, policy and the termination condition of the options *deliver-part* and *load-part* are defined over the whole state space spanned by W, the execution of these options has no effect on the state variable *part-ready* that is controlled by some other option (e.g. *inventory* option). In this example $\varphi_{deliver\_part} = \varphi_{load\_part} = \{part\text{-}ready\}$, and $\Omega_{deliver\_part} = \Omega_{load\_part} = \{position\}$. Also $\varphi_{inventory} = \emptyset$ and $\Omega_{inventory} = \{part\_ready\}$.

Two options o_i and o_j are called *coherent* if (1) they are both *partially-factored* options, and (2) $\Omega_{o_i} \cap \Omega_{o_j} = \emptyset$ (this condition is required to ensure that these two options will not affect the same portion of the state space so that they can safely run in parallel). In the above example, *deliver-part* and *inventory* options are coherent, but *deliver-part* and *load-part* options are not coherent, since the state variable *position* is controlled by both *deliver-part* and *load-part* options.

Now, assume O is a set of available options and $\{C_1, C_2, ..., C_n\}$ are n classes of options that partition O into n disjoint classes such that any two options belonging to different classes are coherent (can run in parallel), and any two options within the same class are not coherent (they control shared state variables and cannot run in parallel). Clearly any set of options generated by drawing each option from a separate class can safely be run in parallel (for the above example, we can define two classes of options with this property: $C_1 = \{deliver\_part, load\_part\}$ and $C_2 = \{inventory\}$).

Given the above definitions, we can define the *Concurrent Options* model as a 4-tuple (S, A, P, R):

State space: The state space represented by S is spanned by the set of state variables in the union of W_o sets ($o \in O$):

$$S = \bigcup_{o \in O} S_o \;\; and \;\; W = \bigcup_{o \in O} W_o \qquad (10)$$

It is also simple to verify that:

$$W = \bigcup_{o \in O} \Omega_o \qquad (11)$$

Let S_{Ω_o} denote the sub-space in S_o that is spanned by Ω_o. Note that for every option o, S_{Ω_o} is a sub-space of S_o and S_o is a sub-space of S. For every option $o \in O$ let $\theta_{\Omega_o}(s_o)$ return a vector whose elements are the current value of each state variable in Ω_o for a given state $s_o \in S_o$, and let $\theta_{\Omega_o}(s)$ return a vector whose elements are the current value of each state variable in Ω_o for a given state $s \in S$. Also, let $\theta_o(s)$ return the current $s_o \in S_o$. It is simple to verify that

$$\forall o \in O, \; \theta_{\Omega_o}(s_o) = \theta_{\Omega_o}(s) \qquad (12)$$

Based on equation 11, a sample state vector $s \in S$ can be represented by

$$s = (\theta_{\Omega_{o_1}}(s) : \theta_{\Omega_{o_2}}(s) : ... : \theta_{\Omega_{o_n}}(s)) \qquad (13)$$

where ':' is the concatenation operator. We will use this notation to explain the other components of the model.

Actions: For every state $s \in S$, a set of one or more options, each belonging to a different class can be initiated concurrently, therefore:

$$\forall s \in S, \vec{O}(s) \subseteq C_1 \times C_2 \times ... \times C_n \qquad (14)$$

A concurrent option $\vec{o} \in \vec{O}(s)$ consists of a set of m ($1 \leq m \leq n$) options that can be initiated in parallel. We represent a concurrent option by $\vec{o} = (o_1, o_2, ..., o_m)$ and call it a *multi-option* (to distinguish them from regular options), and also represent the set of available options in state s by $\vec{O}(s)$. We also need to define the event of termination of a multi-option $\vec{o}$. When multi-option $\vec{o}$ is executed in state s, a set of m options $o_i \in \vec{o}$ are initiated. Each option o_i will terminate at some random time t_{o_i}. We can define the event of termination for a multi-option based on either of the following events: (1) when any of the options $o_i \in \vec{o}$ terminates according to $\beta_i(s)$, multi-option $\vec{o}$ is declared terminated and the rest of the options that are not terminated at that point in time, are interrupted, or (2) when all of the options are terminated. In this paper, we restrict our discussion to a model based on the first definition. Based on this definition, the next decision epoch happens at a random time t according to $t = min\{t_{o_i} \mid 1 \leq i \leq m\}$.

Transition Probabilities: We now define the state transition probabilities based on the termination event explained above. Let $P^{\vec{o}}_{ss'}$ denote the probability that multi-option $\vec{o} = (o_1, o_2, ..., o_m)$ is initiated in state s, and terminates in state s'. Based on the first definition of the termination event, it denotes the probability that every option $o_i \in \vec{o}$ is initiated in state s, and when at least one of them terminates (according to its β_i function), the state of the system is s'. For each option o_i, the transition probability when only that option is initiated is well defined and is part of the option model. When more than one option is initiated simultaneously, the transition probability of the multi-option $\vec{o}$ is computed as follows. Let $P^{\vec{o}}(s, s', k)$ denote the the probability that multi-option $\vec{o}$ is initiated in state s, and terminates in state s' after k steps. Then:

$$P^{\vec{o}}_{ss'} = \sum_{k=1}^{\infty} P^{\vec{o}}(s, s', k)\gamma^k \qquad (15)$$

Now, let $\xi^{\vec{o}}(s, s', k)$ denote the probability that multi-option $\vec{o}$ is initiated in state s, and after k steps transitions to state s'. We can compute $P^{\vec{o}}(s, s', k)$ based on $\xi^{\vec{o}}(s, s', k)$, since it can be viewed as the probability of initiating the multi-option $\vec{o}$ in state s, running it for k steps without any of the options being terminated until step k, and at least one of the options terminates at step k:

$$\begin{aligned} &P^{\vec{o}}(s, s', k) = \\ &\xi^{\vec{o}}(s, s', k)\ (1 - \prod_{i=1}^{m}(1 - \beta_i(s'))) \end{aligned} \qquad (16)$$

The second term on the right hand side in equation 16 denotes the probability that at least one of the options terminates in state s' according to its termination condition β_i.

For every option o_i, $\xi^{o_i}(s_i, s'_i, 1)$ (where $s_i, s'_i \in S_i$) denotes the single step transition probability that the option o_i is executed in state s_i for only one step and the next state is s', assuming that no other option is initiated. Since each option o_i affects the state space through the set of state variables in Ω_{o_i} (and Ω_{o_i} sets are disjoint), we can define the $\xi^{o_i}(s_i, s'_i, 1)$ in terms of Ω_{o_i}. Using equation 12 we get:

$$\xi^{o_i}(s_i, s'_i, 1) = \xi^{o_i}(\theta_{\Omega_{o_i}}(s), \theta_{\Omega_{o_i}}(s'), 1) \qquad (17)$$

Using equation 13, we can rewrite the single step transition probability[1] of a multi-option $\vec{o}$:

$$\begin{aligned} \xi^{\vec{o}}(s, s', 1) = P^{\vec{o}}(&(\theta_{\Omega_{o_1}}(s) : \theta_{\Omega_{o_2}}(s) : ... : \theta_{\Omega_{o_m}}(s)), \\ &(\theta_{\Omega_{o_1}}(s') : \theta_{\Omega_{o_2}}(s') : ... : \theta_{\Omega_{o_m}}(s')), \\ &1) \end{aligned}$$

Thus, using equation 17 we get:

$$\begin{aligned} \xi^{\vec{o}}(s, s', 1) &= \prod_{i=1}^{m} \xi^{o_i}(\theta_{\Omega_{o_i}}(s), \theta_{\Omega_{o_i}}(s'), 1) \\ &= \prod_{i=1}^{m} \xi^{o_i}(s_i, s'_i, 1) \end{aligned} \qquad (18)$$

Therefore, the single step transition probability of a multi-option $\vec{o}$ from state s to state s' is the product of single step transition probabilities of each individual option $o \in \vec{o}$ from state s to state s', since through single step execution of every option $o \in \vec{o}$ in state s, each option will control a disjoint set of state variables that mutually contributes to the formation of state s' through their Ω sets, which is independent of the single-step execution of the other options that are running in parallel.

Having defined the single step transition probability (equation 18), we can recursively define the k-step transition probability:

$$\xi^{\vec{o}}(s, s', k) = \sum_{s_j \in S} \left(\begin{array}{c} \xi^{\vec{o}}(s, s_j, k-1) \\ \times \\ \prod_{i=1}^{m}(1 - \beta_i(s_j)) \\ \times \\ \xi^{\vec{o}}(s_j, s', 1) \end{array} \right) \qquad (19)$$

[1] Note that in this representation, for clarity, we have omitted those state variables that are not covered by the union of Ω_o sets ($\forall o \in \vec{o}$). In such cases, we can explicitly list those state variables, since they remain unchanged.

in which the first term on the right hand side denotes the probability of executing multi-option $\vec{o}$ (initiated in state s) for $k-1$ steps and ending up in state s_j, the second term denotes the probability that none of the options $o_i \in \vec{o}$ terminate at state s_j and the last term denotes the probability that the option is initiated in state s_j and executed for a single step, and ended up in state s'.

Based on equations 15, 16, 17, 18 and 19, we can compute $P^{\vec{o}}_{ss'}$, for all $s, s' \in S$ and for all $\vec{o} \in \vec{O}(s)$.

Reward function: For any state $s \in S$ and for any multi-option $\vec{o} \in \vec{O}(s)$:

$$R^{\vec{o}}_s = E\{r_{t+1} + \gamma r_{t+2} + ... + \gamma^{k-1} r_{t+k} \mid \varepsilon(\vec{o}, s, t)\} \quad (20)$$

where $t+k$ is the random time at which multi-option $\vec{o}$ terminates.

State value function: For any Markov policy μ, the state value function can be written

$$\begin{aligned} V^\mu(s) &= E\{r_{t+1} + \gamma r_{t+2} + ... + \gamma^{k-1} r_{t+k} + \\ &\quad \gamma^k V^\mu(s_{t+k}) \mid \varepsilon(\mu, s, t)\} \\ &= \sum_{\vec{o} \in \vec{O}(s)} \mu(\vec{o}, s)\ [R^{\vec{o}}_s + \sum_{s' \in S} P^{\vec{o}}_{ss'} V^\mu(s')] \end{aligned} \quad (21)$$

where k is the duration of multi-option $\vec{o}$ according to the termination event explained above.

Multi-option value of $\vec{o}$ under Markov policy μ:

$$\begin{aligned} Q^\mu(s, \vec{o}) &= E\{r_{t+1} + \gamma r_{t+2} + ... + \gamma^{k-1} r_{t+k} + \\ &\quad \gamma^k V^\mu(s_{t+k}) \mid \varepsilon(\vec{o}, s, t)\} \\ &= E\{r_{t+1} + \gamma r_{t+2} + ... + \gamma^{k-1} r_{t+k} + \\ &\quad \gamma^k \sum_{\vec{o}' \in \vec{O}(s_{t+k})} \mu(s_{t+k}, \vec{o}') Q^\mu(s_{t+k}, \vec{o}') \mid \varepsilon(\vec{o}, s, t)\} \\ &= R^{\vec{o}}_s + \sum_{s' \in S} P^{\vec{o}}_{ss'} \sum_{\vec{o}' \in \vec{O}(s')} \mu(s', \vec{o}') Q^\mu(s', \vec{o}') \end{aligned} \quad (22)$$

where k is the duration of multi-option $\vec{o}$ according to the termination condition explained above.

It has been shown earlier that the set of Markov options defines a semi-Markov decision process (SMDP) (Sutton et al., 1999). It is natural to conjecture whether this result carries over to multi-options. We show that this is indeed the case, with the assumptions discussed above.

Theorem (MDP + Concurrent Options = SMDP): For any MDP, and any set of *concurrent Markov options* defined on that MDP, the decision process that selects only among multi-options, and executes each one until its termination according to the multi-option termination condition, forms a semi-Markov decision process.

Proof: (Sketch) For a decision process to be a SMDP, it is required to define (1) set of states, (2) set of actions, (3) an expected cumulative discounted reward defined for every pair of state and action and (4) a well defined joint distribution of the next state and next decision epoch. In the concurrent options model, we have defined the set of states and the set of actions are the multi-options. The expected cumulative discounted reward and joint distributions of the next state and next decision epoch have been defined in terms of the underlying MDP. The policy and termination condition for every option that belongs to a multi-option, and the termination condition for a multi-option have also been defined.

4 Experimental Results

In this section we present a simple computational example that illustrates planning with concurrent options. We adopt the *rooms example* from (Sutton et al., 1999) and we add doors in each of the four hallways (Figure 1).

The agent cannot pass through locked (closed) doors unless it is holding the key. The state of the environment for this example consists of three state variables: position of the agent in the environment (represented by cells), state of the doors, and the state of the key. At any state, the agent can select actions from the set of navigation actions or the set of key related actions. Navigation actions comprises four stochastic primitive actions: *up*, *down*, *left* and *right* (Figure 1). Each navigation action with probability 9/10 causes the agent to move one cell in the corresponding direction, and with probability 1/10, moves the agent in one of the other three directions, each with probability 1/30. In either case, if the movement would take the agent into a wall, or a closed door when the agent is not holding the key, then the agent will remain in the same cell. We have also defined a *room-nop* primitive action that does not change the position of the agent (with probability 1). In each of the four rooms, we define two hallway Markov options (multi-step) that take the agent from anywhere within the room to one of the two hallway cells leading out of the room.

Figure 2 shows the policy for one of the hallway options. The termination condition $\beta(.)$ for hallway options is zero for states inside the room, except for the cell next to the target hallway cell (shaded cell in the Figure 2), in which the termination condition also de-

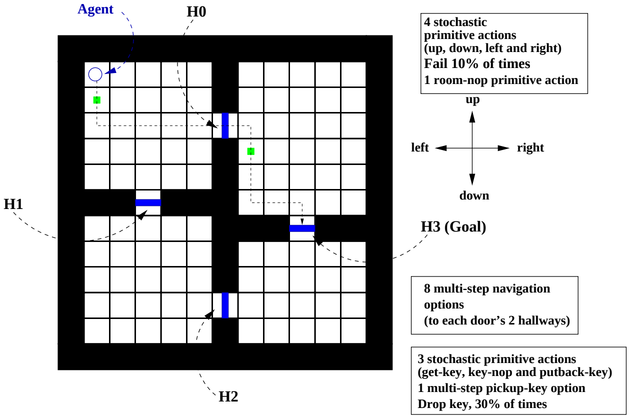

Agent
H0
H1
H2
H3 (Goal)
4 stochastic
primitive actions
(up, down, left and right)
Fail 10% of times
1 room-nop primitive action
up
left
right
down
8 multi-step navigation
options
(to each door's 2 hallways)
3 stochastic primitive actions
(get-key, key-nop and putback-key)
1 multi-step pickup-key option
Drop key, 30% of times

Figure 1: The rooms example is a grid world environment that includes locked doors in each hallway. The agent needs to pickup a key in order to unlock the doors and pass through the hallways. There are four stochastic primitive cell-to-cell navigation actions plus one *room-nop* primitive action and three stochastic primitive key actions defined for this environment. Eight multi-step hallway options (two for each room) and one multi-step key option are defined on top of these primitive actions respectively. The hallway options of each room take the agent from any cell in the room to one of the hallways connected to that room.

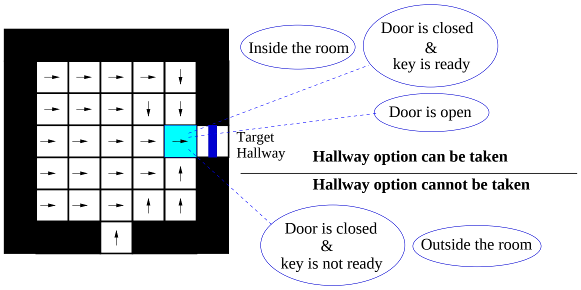


Figure 2: The policy associated with one of the hallway options. This figure also shows that the option can be taken at any cell within that room. In the shaded cell that is adjacent to the target hallway, the option can be taken if either the door is open, or the agent is holding the key at that cell. The option terminates in the target hallway and also in the shaded cell if the door is closed and the agent does is not holding the key.

pends on the state of the door in the target hallway, and also whether or not the agent is holding the key. In this cell, the termination condition is zero if either the door is open, or the door is closed and the agent is holding the key, otherwise the hallway option will terminate with probability 1. Assume that the agent is currently executing the hallway option and its current location is the cell adjacent to the target hallway, and also the door in the target hallway is locked. Then if the agent is holding the key, it continues executing the hallway option which will unlock the door and takes the agent to the target hallway, based on the stochastic process explained above. Once the agent exits the hallway, the door within that hallway changes its state to locked and closes again. The initiation set of the hallway option comprises all the states within the room plus the non-target hallway state leading into room. Note that for the cell next to the target hallway, the option can be initiated if either the door is open, or the door is closed and the agent is holding the key.

Figure 3 shows the states of the key process. Note that only at state six is the key ready to unlock doors (i.e. the agent is holding the key). There are seven states defined for the key process and the agent can select one of three primitive actions, *get-key* that is defined over states S_0 through S_5, *key-nop* that is defined on all key states and *putback-key* that is defined only at state S_6. Primitive action *key-nop* has a stochastic effect in that the agent may drop the key with probability $3/10$ once taken at state S_6 (dropping the key will reset the state of the key to S_0) and with probability $7/10$ will not change the state of the key process. If the *key-nop* action is taken at states S_0 through S_5, it will not change the state of the key process. The agent will advance the state of the process when action *get-key* is executed, or will reset the state of the process to S_0 if action *putback-key* is executed. We also provide a multi-step *pickup-key* Markov option (on top of the *get-key* primitive action). *Pickup-key* option's policy π advances key state (with probability 1) until the key is ready (state S_6). The termination condition $\beta(.)$ for *pickup-key* option is 1 for state S_6, and zero for rest of the states. Its initiation set comprises all of the key states except state S_6.

Figure 4 shows the evolution of the state of the environment when hallway options and key options run in parallel. Note that these options share the "key state" state variable, but they affect disjoint subspaces of the state space (e.g. hallway options will only control "position" and the "doors state" variables and key options will only affect the "key state" variable).

Using the notation developed in section 3, we define two classes of options: $C_1 = \{hallway_0, hallway_1, ..., hallway_7, room\_nop\}$[2]

[2] Note that even though hallway options of different rooms also control disjoint sub-spaces of the state space, we put them in the same class since based on their initiation

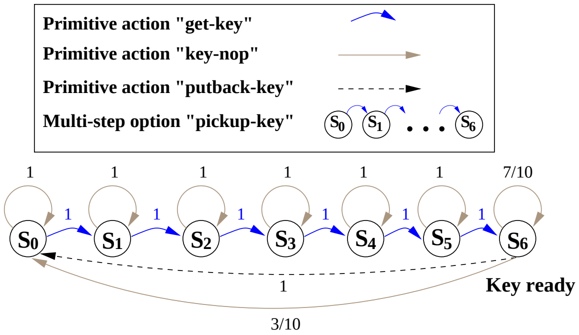

Primitive action "get-key"
Primitive action "key-nop"
Primitive action "putback-key"
Multi-step option "pickup-key"
S0
S1
S6
1
1
1
1
1
1
7/10
S0
S1
S2
S3
S4
S5
S6
1
Key ready
3/10

Figure 3: Representation of the key pickup option. At state s_6, agent is holding the key.

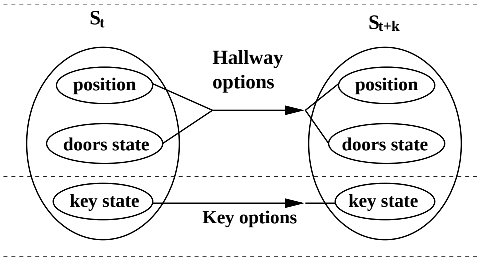


Figure 4: Evolution of states in the room navigation task. Initiation set, policy and termination probabilities of each option may share state variables with other parallel options (hallway options and pickup-key options share the "key state" variable), but each will mutually affect disjoint sets of state variables (hallway options control the position and the door state variables, and key option only controls the key state variable).

(eight hallway options and one *room-nop* option) and $C_2 = \{pickup\_key, key\_nop, putback\_key\}$[3]. For any hallway option, the set of state variables $W_{navigation} = \{position, doors\_state, key\_state\}$ (where *doors_state* specifies the state of each door) and for the key option $W_{key} = \{key\_state\}$. Note that $\Omega_{navigation} = \{position, doors\_state\}$ and $\Omega_{key} = \{key\_state\}$. Based on equation 10, the state space of the overall process is spanned by the set of state variables $W = W_{navigation} \cup W_{key} = \{position, doors\_state, key\_state\}$. The multi-options are members of the set $C_1 \times C_2$. Since only two hallway options can be taken in each room plus the *room-nop* option, and three key options can be taken in any cell within each room, a maximum of 9 multi-options can be defined in every state.

In order to evaluate the concurrent options framework, we compare its performance for the rooms example with the standard options framework. Note that in standard options framework, only one option can be taken at a time. Based on the concurrent options theorem in the previous section, we can apply SMDP-based Q-learning. Each multi-option is viewed as an indivisible, opaque unit of action (although intra-multi-option methods could also be developed). When multi-option $\vec{o}$ is initiated in state s, it transitions to state s' in which multi-option $\vec{o}$ terminates according to the termination condition defined for multi-options. We can use SMDP Q-learning method (Bradtke & Duff, 1995; Sutton et al., 1999) to update the multi-option-value function $Q(s, \vec{o})$ after each decision epoch where the multi-option $\vec{o}$ is taken in some state s and terminates in s':

$$
\begin{aligned}
Q(s,\vec{o}) \leftarrow & Q(s,\vec{o}) + \\
& \alpha \left[r + \gamma^k \max_{\vec{o'} \in \vec{O}(s')} Q(s', \vec{o'}) - Q(s,\vec{o}) \right] \qquad (23)
\end{aligned}
$$

where k denotes the number of time steps since initiation of the multi-option $\vec{o}$ at state s and its termination at state s', and r denotes the cumulative discounted reward over this period.

For a fixed position as the starting point (upper left corner cell) and a fixed goal (hallway $H3$) in Figure 1, we used both frameworks in order to learn the policy to navigate from starting position to the target. For any primitive action, a reward of -1 is provided as the single step reward in order to learn a policy that optimizes the time for performing the task. Figure 5 shows the median of time (in terms of number of primitive actions taken until success) where for trial n, it is the median of all trials from 1 to n. This figure shows that the standard options framework learns faster than the concurrent options framework, but eventually, the concurrent options framework learns a better policy. Also in Figure 1, the small solid rectangles within cells represent the approximate location that the "pickup-key" option is initiated as the agent moves toward hallway $H0$ and goal. With the policy learned using sequential execution of options, the agent navigates to the cell adjacent to the target hallway with the locked door using the corresponding hallway option, then it initiates the pickup-key option and waits in that cell for 7 steps until the key state advances to state S_6 at which point the key is ready to use. By overlapping execution of the hallway option and pick-up key option, the agent minimizes the time in order to reach the goal.

set, they can not run in parallel.

[3] Note that *room-nop*, *key-nop* and *putback-key* are primitive actions, but they are special case of single-step options.

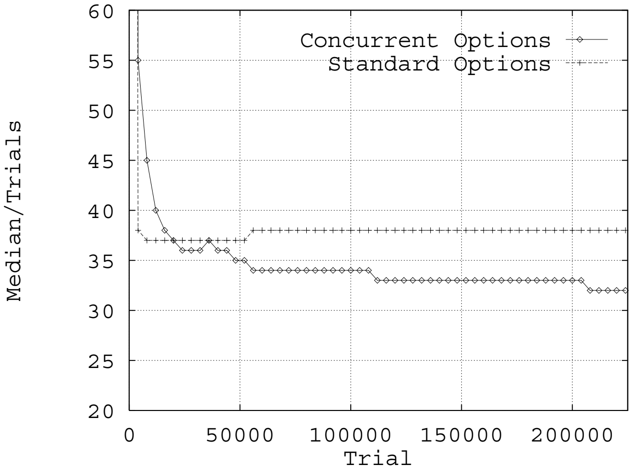


Figure 5: Performance of standard options and concurrent options framework using SMDP Q-learning in rooms example. The horizontal axis represents the number of trials, and the vertical axis shows the median of number of steps until success across trials.

5 Discussion and Future Work

In this paper we introduced the concurrent options model, which formalizes planning under uncertainty with parallel temporally extended actions where each action affects mutually disjoint subspaces of the environment state space. The key assumption that is required for a set of options to be safely run concurrently is that they are Markov. This restriction is necessary in order to have well defined termination condition and state prediction (transition probabilities). Consider the counter-example of a semi-Markov navigation option where the agent moves around the perimeter of a room twice before deciding to exit the room. For the key option to be invoked in parallel with this semi-Markov option, the agent would have to know how long the exit-room option had been running (information not available in the current decision epoch). We provided a simple computational experiment which shows that planning with concurrent options is more effective than when only one option is executed at a time.

There is a clear connection between the model of concurrent options proposed in this paper to work on factored MDPs (Boutilier & Goldszmidt, 1995; Boutilier et al., to appear; Dean & Givan, 1997). Our approach relies on factoring the set of state variables using sets of behaviors that do not conflict. However, we do not use compact models of actions, such as dynamic Bayesian nets. One immediate problem for future research is to investigate how to represent options using DBNs, which would then facilitate compact representations of multi-options as well. Compact representations of options would also provide for a form of value function approximation, an issue that we have ignored in this paper.

There are many interesting directions for further research: (1) Other forms of termination condition could be explored (terminate a multi-option when all of the options terminate, or terminate a multi-option when any option terminates but do not interrupt the rest of them and let them continue until their natural termination). (2) Planning and learning with concurrent options when semi-Markov options are also included (in addition to Markov options). (3) We could also learn the model of a multi-option (reward function and transition probabilities of a multi-option) (4) Finally, we should investigate the termination of a multi-option when interrupting a multi-option has higher value than continuing the multi-option (Sutton et al., 1999).

References

Boutilier, C., Dearden, R., & Goldszmidt, M. (to appear). Stochastic dynamic programming with factored representations. *To appear in Artificial Intelligence.*

Boutilier, C., & Goldszmidt, M. (1995). Exploiting structure in policy construction. *Proceedings of IJCAI.*

Bradtke, S., & Duff, M. (1995). Reinforcement learning methods for continuous-time markov decision problems. *Advances in Neural Information Processing Systems*, *7*, 393–400.

Cichosz, P. (1995). Learning multidimensional control actions from delayed reinforcements. *Eighth International Symposium on System-Modelling-Control (SMC-8).* Zakopane, Poland.

Dean, T., & Givan, R. (1997). Model minimization in markov decision processes. *Proceedings of AAAI.*

Parr, R. E. (1998). *Hierarchical control and learning for markov decision processes.* Doctoral dissertation, Department of Computer Science, University of California, Berkeley.

Singh, S., & Cohn, D. (1998). How to dynamically merge markov decision processes. *Proceedings of NIPS 11.*

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 181–211.

Reinforcement Learning from Limited Observations

Lyle Ungar and Greg Grudic
(ungar@central.cis.upenn.edu)

Abstract

Using reinforcement learning (RL) to optimize control of physical systems such as robots or telecommunications systems requires learning from limited observations. Classic RL successes such as playing backgammon and controlling elevators have relied on simulators to generate millions of observations. Unfortunately, such simulators are often not available, or are very slow. Experiments and measurements in physical systems tend to be expensive and noisy; one cannot do a million experimental runs on a robot. We claim that in order to use RL with limited observations one must use both extensive prior knowledge and careful sequential experimental design. One promising way to use prior knowledge is Boundary Localized Reinforcement Learning (BLRL), in which one starts with a mode-switching controller in which the state space is divided into different regions (modes), each using a different controller. Policy gradient methods are used to shift the mode boundaries in directions which improve performance. Localizing learning to decision boundaries vastly reduces the need for randomized search, at a cost of only finding locally optimal policies.

Localizing Search in Reinforcement Learning

Greg Grudic
Institute for Research in Cognitive Science
University of Pennsylvania
Philadelphia, PA, USA
grudic@linc.cis.upenn.edu

Lyle Ungar
Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
ungar@cis.upenn.edu

Abstract

Reinforcement learning (RL) can be impractical for many high dimensional problems because of the computational cost of doing stochastic search in large state spaces. We propose a new RL method, Boundary Localized Reinforcement Learning (BLRL), which maps RL into a mode switching problem where an agent deterministically chooses an action based on its state, and limits stochastic search to small areas around mode boundaries, drastically reducing computational cost. BLRL starts with an initial set of parameterized boundaries that partition the state space into distinct control modes. Reinforcement reward is used to update the boundary parameters using the policy gradient formulation of Sutton et al. (2000). We demonstrate that stochastic search can be limited to regions near mode boundaries, thus greatly reducing search, while still guaranteeing convergence to a locally optimal deterministic mode switching policy. Further, we give conditions under which the policy gradient can be arbitrarily well approximated without the use of any stochastic search. These theoretical results are supported experimentally via simulation.

Introduction

A cornerstone of all Reinforcement Learning (RL) is the concept that an agent uses a trial and error strategy to explore its environment and thus learns to maximize its reward. This trial and error process is usually implemented via stochastic search, which is governed by a probability distribution of actions taken during exploration. Such a stochastic search strategy has proven effective in many RL applications with low dimensional state spaces (Kaelbling, Littman, & Moore 1996).

The difficulty inherent in applying a stochastic search strategy (or any search strategy) to higher dimensional problems is that, in general, the search space grows exponentially with the number of state variables. As a consequence, the computational cost of reinforcement learning quickly becomes impractical as the dimension of the problem increases. The use of function approximation techniques to learn generalizations across large state spaces, and then the use of these generalizations to direct the search process, has been suggested as one possible solution to this curse of dimensionality problem in RL. However, even when function approximation techniques successfully generalize, the dimension of the search remains unchanged, and its computational cost can still be impractical.

We propose to reduce the computational cost of search in high dimensional spaces by searching only limited regions of the state space. The size of the search region bounds the computational cost of RL. Intuitively, the smaller the search region, the lower the computational cost of learning, making it possible to apply RL to very high dimensional problems.

To limit the search, we consider the class of deterministic mode switching controllers, where the action executed by an agent is deterministically defined by its location in state space. (See Figure 1.) Mode switching controllers are commonly used in many control applications in order to allow relatively simple controllers to be used in different operating regimes, such as aircraft climbing steeply vs. cruising at constant elevation (Lainiotis 1976). Mode switching has additional benefit for RL in applications such as robotics, where random actions may result in unsafe outcomes, and therefore actions must be deterministically chosen based on prior knowledge of which actions are both safe and beneficial.

Representing the agent's policy as a deterministic mode switching controller allows us to create a new type of reinforcement learning, Boundary Localized Reinforcement Learning (BLRL), in which the trial and error is limited to regions near mode boundaries. As BLRL is concerned solely with updating the boundary locations between modes, we parameterize these boundaries directly and perform RL on this parameterization using the Policy Gradient formulation of (Sutton *et al.* 2000). In effect, the learning shifts the mode boundaries to increase reward.

This paper presents three new theoretical results. The first result states that any stochastic policy (i.e. stochastic control strategy) can be transformed into a mode switching policy, which localizes search to near mode boundaries. The practical consequence of this result is that an RL problem can be converted to a BLRL problem, thus taking advantage of the convergence properties of BLRL in high dimensional state spaces. The second theoretical result states that convergence to a locally optimal mode switching policy is still obtained when stochastic search is limited to near mode boundaries. This means that most of the agent's state space can be ignored, while still guaranteeing convergence to a locally op-

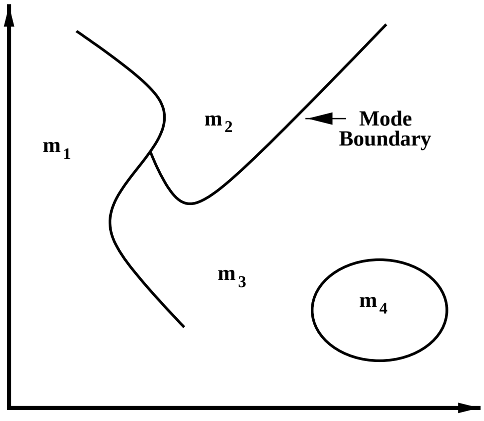


Figure 1: A Mode Switching Controller consists of a finite number of modes $m_1, m_2, ...$ or actions, which are deterministically applied in specific regions of the workspace. The state space is therefore divided into regions specified by Mode Boundaries.

timal solution. The final theoretical result gives a bound on the error in the policy gradient formulation if an agent uses a deterministic search strategy instead of a stochastic one. Surprisingly, convergence to approximately locally optimal deterministic policies does not require the execution of more than one type of action in each region of the state space associated with a single mode. This contrasts with typical RL search where different actions are executed in the same state in order to calculate a gradient in the direction of maximal reward. Avoiding executing multiple modes in each region allows us to limit the use of potentially dangerous or expensive random actions because we search only by making small adjustments in boundary locations. These theoretical results are supported experimentally via simulation.

RL Problem Formulation

Reinforcement Learning as a MDP

The typical formulation of RL is as a Markov Decision Process (MDP) (Kaelbling, Littman, & Moore 1996). The agent has a set of states S (usually discrete), a set of actions A, a reward function $R : S \times A \rightarrow \Re$, and a transition function $T : S \times A \rightarrow \pi(S)$ where $\pi(S)$ is a probability distribution of actions over the states S. The transition function is written as $T(s, a, s')$ and defines the probability of making a transition from state s to state s' using action a. The goal of reinforcement learning is to find a policy π (i.e. a probability distribution of actions over states), such that the reward obtained is optimized. Optimal policies are typically learned by learning the value of taking a given action in a given state, and then choosing the action which gives the maximum expected reward. The process of finding this optimal policy is formulated as a stochastic search typically dictated by the current policy π.

The basic premise of this standard approach to RL is that a good estimate of the value function can be obtained everywhere in state space. In small state spaces this premise is typically true, however, obtaining such estimates in larger state spaces can require extreme amounts of search.

Policy Gradient RL

The policy gradient formulation of RL which we use differs from the typical RL formulation in that policies are defined by some parameterization vector θ and there is a performance metric ρ that is a function of the policy, and can therefore also be parameterized by θ. Policy Gradient RL is then formulated as a gradient based update of the parameters as follows:

$$\theta_{t+1} = \theta_t + \alpha \frac{\partial \rho}{\partial \theta} \qquad (1)$$

where $\partial\rho/\partial\theta$ is the *performance gradient* and α is a positive step size. This formulation relies on the assumption that if the estimate of $\partial\rho/\partial\theta$ is accurate and α is small, then the updated policy parameters θ will give better performance, and the policy will eventually converge to a local optimum.

The policy gradient formulation dates back to Williams' (1987, 1992) REINFORCE algorithm which is known to give an unbiased estimate of the performance gradient $\partial\rho/\partial\theta$. However, REINFORCE suffers from slow convergence resulting from the fact that it requires a good estimate of the actual value of each state (termed the baseline reward parameter) to get a low variance estimate of $\partial\rho/\partial\theta$. This baseline reward parameter is difficult to calculate in practice and therefore REINFORCE has not been widely applied on RL problems.

Recently a number of policy gradient algorithms have been proposed which use function approximation estimates of the state-action value function to give low variance estimates of the performance gradient $\partial\rho/\partial\theta$, and thereby improve rate of convergence (Baird & Moore 1999; Sutton *et al.* 2000; Konda & Tsitsiklis 2000; Baxter & Bartlett 1999). However, there is experimental evidence that direct but *selective* sampling of the value of executing actions in states can give low variance estimates of $\partial\rho/\partial\theta$ without using function approximation (Grudic & Ungar 2000).

In this paper we use the *Action Transition Policy Gradient* (ATPG) algorithm formulation presented in (Grudic & Ungar 2000). The ATPG algorithm selectively samples the state-action value function whenever the agent changes actions, and uses only these samples to obtain estimates of $\partial\rho/\partial\theta$. The performance gradient estimate is based on the relative difference between the values of two different actions which are executed within one time step of each other. This utilization of relative reward gives a low variance estimate of $\partial\rho/\partial\theta$, and allows ATPG to typically converge in many orders of magnitude fewer iterations than other policy gradient algorithms on a variety of RL problems (Grudic & Ungar 2000).

Boundary Localized Policy Gradients (BLPG)

Policy Gradient Formulation

Our formulation of BLRL is based on *Policy Gradient Theorem* of (Sutton *et al.* 2000), which we briefly review below. For each time step $t \in \{0, 1, ...\}$ there is an associated state $s_t \in S$, action $a_t \in A$, and reward $r_t \in \Re$.

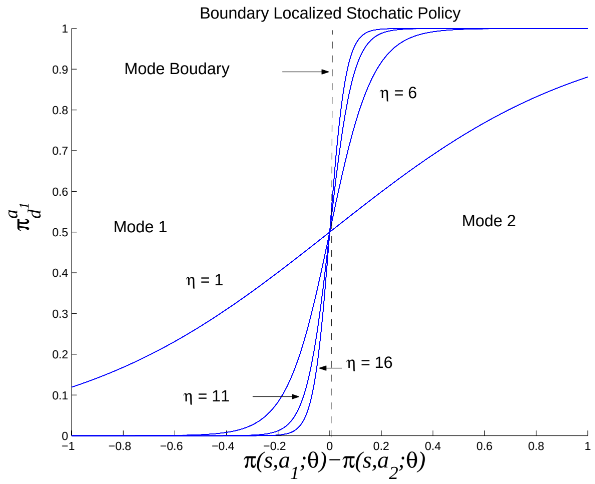


Figure 2: The η-transformation.

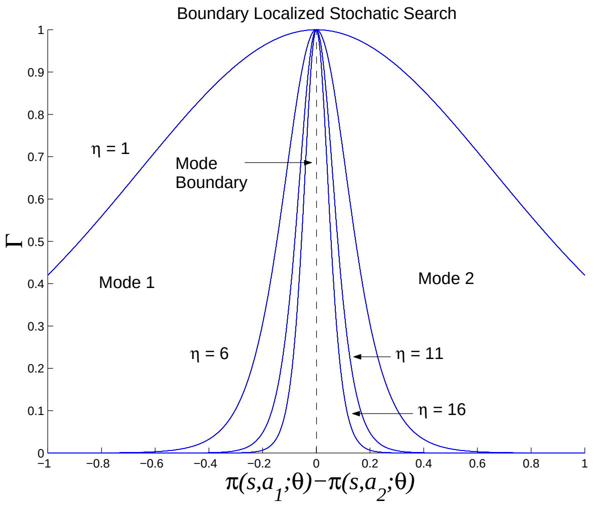


Figure 3: The magnitude of the policy gradient goes to zero everywhere except mode boundaries as $\eta \to \infty$.

Using the usual MDP formulation, the dynamics of the environment are characterized by state transition probabilities $P_{ss'}^{a} = \Pr\left\{s_{t+1} = s' \left| s_t = s, a_t = a\right.\right\}$, and expected rewards $R_s^a = E\left\{r_{t+1} \left| s_t = s, a_t = a\right.\right\}$, $\forall s, s' \in S, a \in A$. The agent is assumed to follow a probabilistic policy characterized by $\pi\left(s, a; \theta\right) = \Pr\left\{a_t = a \left| s_t = s; \theta\right.\right\}$, $\forall s \in S, a \in A$ and $\theta \in \Re^l$ is a l dimensional policy parameterization vector. The additional assumption made on the policy is that $\partial\pi/\partial\theta$ exists.

The Policy Gradient Theorem allows for both the average reward and discounted reward formulations for a performance metric. For brevity, we only state the discounted reward formulation here. The discount reward performance metric for an agent that starts at state s_0 is given by:

$$\rho\left(\pi\right) = E\left\{ \sum_{t=1}^{\infty} \gamma^t r_t \middle| s_0, \pi \right\} \tag{2}$$

where $\gamma \in [0, 1]$ is a discount reward factor. A *state-action value function* is defined as:

$$Q^{\pi}\left(s, a\right) = E\left\{ \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \middle| s_t = s, a_t = a, \pi \right\} \tag{3}$$

Finally, a discounted weighting of states encountered starting at state s_0 and then following π is defined by $d^{\pi} = \sum_{t=0}^{\infty} \gamma^t \Pr\left(s_t = s \left| s_0, \pi\right.\right)$.

Given the above definitions, the *Policy Gradient Theorem* states that the exact expression for the policy gradient is:

$$\frac{\partial\rho}{\partial\theta} = \sum_s d^{\pi}\left(s\right) \sum_a \frac{\partial\pi(s, a; \theta)}{\partial\theta} Q^{\pi}(s, a) \tag{4}$$

Boundary Localization: The η-Transform

The Policy Gradient Theorem assumes that policies are characterized by probability distributions: i.e. $\pi\left(s, a; \theta\right) = \Pr\left\{a_t = a \left| s_t = s; \theta\right.\right\}$. In this section we demonstrate that any policies thus formulated can be transformed into approximately deterministic policies, while still preserving the policy gradient convergence results. Consider a policy that consists of only two possible actions: $\pi(s, a_1; \theta)$ and $\pi(s, a_2; \theta)$. These policies can be mapped to boundary-localized stochastic policies, denoted by $\pi_d(s, a_1; \theta)$ and $\pi_d(s, a_2; \theta)$ respectively, using the following transformations:

$$\pi_d\left(s, a_1; \theta\right) = \frac{1}{2}\left[1 + \tanh\left(\eta\left(\pi\left(s, a_1; \theta\right) - \pi\left(s, a_2; \theta\right)\right)\right)\right] \tag{5}$$

and

$$\pi_d\left(s, a_2; \theta\right) = \frac{1}{2}\left[1 + \tanh\left(\eta\left(\pi\left(s, a_2; \theta\right) - \pi\left(s, a_1; \theta\right)\right)\right)\right] \tag{6}$$

where $\eta \to \infty$. We refer to these transformations as η-transformations. Figure 2 shows the effect of η on the probability distribution of the action a_1 (i.e. $\pi_c^{a_1} \equiv \pi_d\left(s, a_1; \theta\right)$). We can see that as $\eta \to \infty$ the probability of executing a_1 in regions of the state space where $\left(\pi\left(s, a_1; \theta\right) - \pi\left(s, a_2; \theta\right)\right) < 0$ becomes arbitrarily small. Similarly, in regions of the state space where $\left(\pi\left(s, a_1; \theta\right) - \pi\left(s, a_2; \theta\right)\right) > 0$ the probability of executing action a_1 is arbitrarily close to 1 as $\eta \to \infty$. Therefore the η-transformation transforms a policy $\pi(s, a_1; \theta)$ which is stochastic everywhere in state space, to a policy $\pi_d(s, a_1; \theta)$ which is stochastic only near the boundaries defined by $\left(\pi\left(s, a_1; \theta\right) - \pi\left(s, a_2; \theta\right)\right) = 0$. We refer to these regions in state space as *mode boundary* regions.

Boundary Localized Policy Gradient

The η-transformation makes the policy gradient become close to zero everywhere except at mode boundaries. To see this, differentiate the BL policy $\pi_d(s, a_1; \theta)$ with respect to the parameters θ as follows:

$$\begin{aligned} \frac{\partial\pi_d^{a_1}}{\partial\theta} &= \frac{\eta}{2}\left(\operatorname{sech}^2\left(\eta\left(\pi^{a_1} - \pi^{a_2}\right)\right)\right)\left(\frac{\partial\pi^{a_1}}{\partial\theta} - \frac{\partial\pi^{a_2}}{\partial\theta}\right) \\ &\triangleq \Gamma\left(\eta, \left(\pi^{a_1} - \pi^{a_2}\right)\right)\left(\frac{\partial\pi^{a_1}}{\partial\theta} - \frac{\partial\pi^{a_2}}{\partial\theta}\right) \end{aligned} \tag{7}$$

where, by definition, $\pi^{a_1} \equiv \pi(s, a_1; \theta)$, $\pi^{a_2} \equiv \pi(s, a_2; \theta)$, $\pi_d^{a_1} \equiv \pi_d(s, a_1; \theta)$ and $\pi_d^{a_2} \equiv \pi_d(s, a_2; \theta)$. Equation (7) indicates that the performance gradient has the following proportionality property:

$$\left|\frac{\partial \rho}{\partial \theta}\right| \propto \Gamma\left(\eta, \left(\pi^{a_1} - \pi^{a_2}\right)\right) \qquad (8)$$

This proportionality is plotted in Figure 3, where we see that as $\eta \to \infty$, the policy gradient approaches zero everywhere except near mode boundaries. This means that only regions in state space near mode boundaries need be stochastically searched when BL policies are used. The result is that BL policies have a significantly reduced search space than standard stochastic polices, making them computationally more viable for high dimensional RL problems.

The argument presented above for a policy of two actions can be extended to any finite number of actions. Therefore the η-transformation is valid for any finite set of policies, and one can transform any stochastic policy to a BL policy. Below we state the *Boundary Localized Policy Gradient Theorem*, which is a direct extension of the Policy Gradient theorem.

<u>Theorem</u>: Boundary Localized Policy Gradient *For any MDP, in either the average or discounted start-state formulations,*

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi_d(s, a; \theta)}{\partial \theta} Q^\pi(s, a) \qquad (9)$$

Proof Sketch: If $\partial\pi/\partial\theta$ exists then because the η-transformation is continuously differentiable, so does $\partial\pi_d/\partial\theta$. The rest of the proof follows that of (Sutton *et al.* 2000).

The significance of the BLPG theorem is that locally optimal BL polices can be learned using policy gradients. Therefore, even though search is localized to a very small region of the state space, a policy gradient algorithm (9) will still converge to a locally optimum policy.

Policy Gradients for Deterministic Policies

One of the problems with applying stochastic search-based RL to such applications as robotics is that random actions executed by a robot may result in unsafe or expensive outcomes. For example, if a robot is navigating a hallway and randomly decides to explore the result of the action *go towards a human "obstacle"*rather than try to avoid *"it"*, the result may be an injured human. Therefore, in this section we formulate an error bound on the policy gradient if the agent does not employ a stochastic search policy. Once again, consider a stochastic policy of two actions: $\pi(s, a_1; \theta)$ and $\pi(s, a_2; \theta)$. If an agent executes action a_1 that moves it a distance δ in state space, and thereafter executes action a_2, then the exact policy gradient is given by (4) and can be written as:

$$\begin{aligned}\frac{\partial \rho}{\partial \theta} = \; & d^\pi(s)\left[\frac{\partial \pi(s,a_1;\theta)}{\partial \theta} Q^\pi(s, a_1) + \right. \\ & \left. \frac{\partial \pi(s,a_2;\theta)}{\partial \theta} Q^\pi(s, a_2)\right] + \\ & d^\pi(s+\delta)\left[\frac{\partial \pi(s+\delta,a_1;\theta)}{\partial \theta} Q^\pi(s+\delta, a_1) + \right. \\ & \left. \frac{\partial \pi(s+\delta,a_2;\theta)}{\partial \theta} Q^\pi(s+\delta, a_2)\right]\end{aligned} \qquad (10)$$

Note that the exact expression for the policy gradient requires knowledge of the state-action value function for both actions at both locations in state space: i.e. $Q^\pi(s, a_1)$, $Q^\pi(s, a_2)$, $Q^\pi(s+\delta, a_1)$, and $Q^\pi(s+\delta, a_2)$ must all be known. If an agent is executing a deterministic policy, then under the current policy π, action a_2 has never been executed in state s, and action a_1 has never been executed in state $s+\delta$; this means that $Q^\pi(s, a_2)$ and $Q^\pi(s+\delta, a_1)$ are not known. Furthermore, if the agent is performing episodic learning and it is obtaining an estimate of the state-action value-function after each episode, then it also will not have estimates of $Q^\pi(s, a_2)$ and $Q^\pi(s+\delta, a_1)$. However, for both the episodic stochastic and deterministic cases, the agent does have estimates of $Q^\pi(s, a_1)$ and $Q^\pi(s+\delta, a_2)$; i.e. because a_1 is executed in s and a_2 is executed in $s+\delta$. Therefore, we propose the following approximation to the policy gradient approximation, which we term the *Boundary Localized Policy Gradient* (BLPG) Approximation:

$$\begin{aligned}\widehat{\frac{\partial \rho}{\partial \theta}} = \; & d^\pi(s)\left[\frac{\partial \pi(s,a_1;\theta)}{\partial \theta} Q^\pi(s, a_1) + \right. \\ & \left. \frac{\partial \pi(s,a_2;\theta)}{\partial \theta} Q^\pi(s+\delta, a_2)\right] + \\ & d^\pi(s+\delta)\left[\frac{\partial \pi(s+\delta,a_1;\theta)}{\partial \theta} Q^\pi(s, a_1) + \right. \\ & \left. \frac{\partial \pi(s+\delta,a_2;\theta)}{\partial \theta} Q^\pi(s+\delta, a_2)\right]\end{aligned} \qquad (11)$$

This approximation works if $Q^\pi(\cdot)$ is continuous. Formally, it must satisfy the Lipschitz smoothness condition:

$$\begin{gathered}\forall s \in S, S \subseteq \Re^N, a \in A, \delta \in \Re^N \\ \exists k > 0, k \in \Re \text{ s.t.} \\ |Q^\pi(s,a) - Q^\pi(s+\delta, a)| \leq k \|\delta\|\end{gathered} \qquad (12)$$

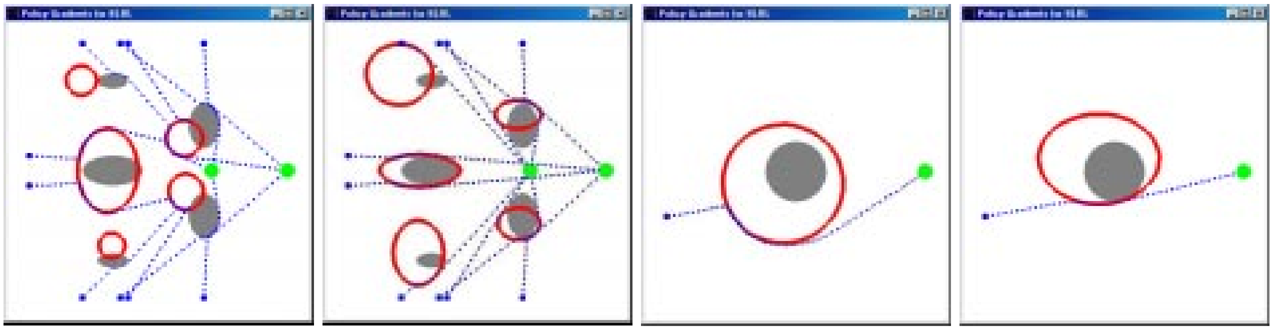
Note that this smoothness condition is satisfied in both the average and discounted reward formalization of RL. Given this formulation, we state the following lemma.

<u>Lemma</u>: BLPG Approximation *Assume that $Q^\pi(s,a)$ is Lipschitz smooth (12), and that the policy $\pi(s, a; \theta)$ has two actions (a_1 and a_2) and is differentiable with respect to θ. Assume also that the agent takes a step of size δ that takes it from a region where action a_1 is performed to a region where action a_2 is performed. Then if the policy gradient is approximated by (11), the error in the approximation is bounded by:*

$$\begin{aligned}\left|\frac{\partial \rho}{\partial \theta} - \widehat{\frac{\partial \rho}{\partial \theta}}\right| \leq \; & k\|\delta\|\left(\left|d^\pi(s)\,\frac{\partial \pi(s,a_2;\theta)}{\partial \theta}\right| + \right. \\ & \left. \left|d^\pi(s+\delta)\,\frac{\partial \pi(s+\delta,a_1;\theta)}{\partial \theta}\right|\right)\end{aligned} \qquad (13)$$

Proof: Subtracting (11) from (10) and taking the absolute value:

$$\begin{aligned}\left|\frac{\partial \rho}{\partial \theta} - \widehat{\frac{\partial \rho}{\partial \theta}}\right| & = \left|d^\pi(s)\,\frac{\partial \pi(s,a_2;\theta)}{\partial \theta}\left[Q(s,a_2) - \right.\right. \\ & \left. Q(s+\delta, a_2)\right] + \\ & d^\pi(s+\delta)\,\frac{\partial \pi(s+\delta,a_1;\theta)}{\partial \theta}\left[Q(s+\delta,a_1) - \right. \\ & \left.\left. Q(s, a_1)\right]\right| \\ & \leq k\|\delta\|\left(\left|d^\pi(s)\,\frac{\partial \pi(s,a_2;\theta)}{\partial \theta}\right| + \right. \\ & \left. \left|d^\pi(s+\delta)\,\frac{\partial \pi(s+\delta,a_1;\theta)}{\partial \theta}\right|\right)\end{aligned}$$



a) Initial b) Learned c) Initial d) Learned

Figure 4: Example of a simulated agent executing episodes in an environment. The agent begins at locations near the top, bottom, and/or left extremes of the environment and goes towards goal positions (small shaded circles) located at the right extreme or near the center. Dashed lines symbolize the agent's path and the obstacles are the larger gray areas. The agent can execute one of two possible actions: if it is executing a deterministic policy and if it is inside one of the regions delineated by a black ellipsoid, it moves away from the ellipsoid's center; otherwise it moves towards a goal position. If the agent is following a stochastic policy, then the ellipsoids indicate regions in state space where the "move away from" action is more probable.

□

Lemma I states that the BLPG approximation error approaches zero as the step size the agent takes approaches zero (i.e. as $||\delta|| \rightarrow 0$).

Simulation Results

We have simulated an agent interacting with its environment using one of two possible actions: the first is *"move towards a goal position"* and the second is *"move away from"* a location in state space. This second action is for obstacle avoidance. If an agent reaches a goal position it gets a reward of +1, if it hits an obstacle it gets a negative reward of -1, and if it is unable to reach its goal within a maximum allotted time, it receives a negative reward of -10.

The agent's state space is *continuously* defined as its Cartesian position, and the policies are parameterized by Gaussians. There are two parameters per dimension per Gaussian - one for position and one for width (i.e. variance). Thus each Guassian adds two parameters per dimension to the policy parameters θ in (4). Typical simulated environments are described in Figure 4. The agent's sensing of position in state space is noisy and is modeled by white noise, which is made proportional to 10% of how far an agent is able to move in one time step. The *Action Transition Policy Gradient* ATPG algorithm (Grudic & Ungar 2000) is used to learn locally optimal policy parameters θ. The ATPG algorithm assumes that the agent interacts with the environment in a series of episodes and the policy parameters θ are updated after each episode. Convergence is therefore measured in number of episodes.

2-D Simulation: Figures 4a and b show a 2-D scenario which has ten possible starting positions, two goal positions, five obstacles, and six Gaussians for defining policies (five for "move away form" which are shown as ellipsoids, and one for "move towards goal", which is most probable everywhere except inside the ellipsoids). Therefore there are a total of 24 policy parameters θ.

Figure 4a shows the initial policy and the resulting paths through the environment. Note that four paths end before a goal is reached and eight paths have collisions with obstacles. Figure 4b shows the paths after the policy parameters have converged to stable values. Note that the location and extent of the Gaussians has converged such that none of the paths now collide with obstacles, and the total distance traveled through state space is shorter.

	Stochastic RL	Stochastic BLRL	Deterministic BLRL
Episodes to converge	6900 (sd 400)	600 (sd 90)	260 (sd 40)

Table 1: 2-D Convergence results with standard deviations.

Table 1 shows the average number of episodes (over ten runs) required for convergence for the three types of polices studied: stochastic, boundary localized stochastic ($\eta = 16$), and deterministic. Note that the purely stochastic polices take the greatest number of episodes to converge, while the deterministic policies take the fewest.

N-D Simulation: We simulated 4, 8, 16, 32, 64, and 128 dimensional environments, with the number of parameters θ ranging from 14 to 512 (i.e. 2 parameters per Gaussian per dimension). The projection of these into the XY plane is shown in Figure 4c and d. Figure 4c shows the starting policies, while Figure 4d shows policies after convergence. In Figure 5, we summarize the convergence results (over ten runs with standard deviation bars) for the three types of policies studied: stochastic, boundary localized stochastic ($\eta = 16$), and deterministic. Note that for both the deterministic and boundary localized policies, convergence is essentially constant with dimension. However, for the stochastic policy, the convergence times explode with dimension. We only report convergence results up to 16 dimensions for stochastic policies - convergence on higher dimensions was still not achieved after 20,000 iterations at which time the simulation was stopped.

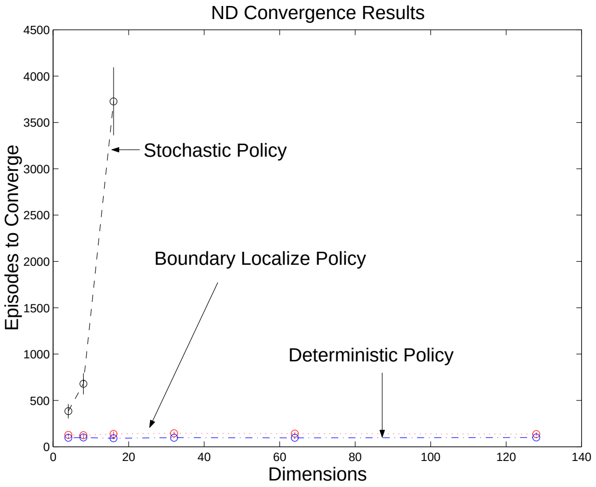


Figure 5: N-D convergence results over ten runs with standard deviation bars.

Discussion and Conclusion

Reinforcement learning (RL) suffers from the combinatorics of search in large state spaces. In this paper we have shown that the stochastic search region in RL can be reduced to mode boundaries by dividing the control policy into a set of state dependent modes. Such controllers are common in complicated control systems, two well-known examples being gain switching controllers (Narendra, Balakrishnan, & Ciliz 1995) and heterogeneous controllers (Kuipers & Åström 1994). The proposed *Boundary Localized Reinforcement Learning* (BLRL) method directly parameterizes the mode boundaries and then uses policy gradients to move the boundaries to give locally optimal policies. Further, we have proven that search can be made deterministic by assuming that the state-action value function is continuous across mode boundaries, a condition that is satisfied in both the average and discounted reward formalization of RL in continuous state spaces.

The policy gradient formulation guarantees that the policies learned are locally, but not necessarily globally, optimal. However, our proposed localization of search means that RL can be applied to high dimensional problems for which global solutions are intractable. Experimental results show that restricting search to boundary regions gives many orders of magnitude reduction in the number of episodes required for convergence. In addition, deterministic policies require slightly fewer episodes to converge than the boundary localized stochastic policies.

The BLRL method is ideally suited for continuous high dimensional RL problems where the agent executes many actions, and each action moves the agent a small distance in state space. One such problem domain is robotics, where actions are executed many times a second (often at 200 Hz or more) and each action moves the robot a small distance through its workspace. In addition, robot controllers can naturally be partitioned into modes, and prior knowledge can be used to define initial boundary parameterizations. Further, domain knowledge can be used to identify which mode transitions are dangerous, and boundaries can be explicitly defined to prohibit such transitions. We are currently applying BLRL to the problem of learning locally optimal policies via reinforcement reward for multiple autonomous robots as they interact with each other and the environment.

Acknowledgements

Thanks to Vijay Kumar and Jane Mulligan for discussing this work with us. This work was funded by the IRCS at the University of Pennsylvania, and by the DARPA ITO MARS grant no. DABT63-99-1-0017.

References

Baird, L., and Moore, A. W. 1999. Gradient descent for general reinforcement learning. In Jordan, M. I.; Kearns, M. J.; and Solla, S. A., eds., *Advances in Neural Information Processing Systems*, volume 11. Cambridge, MA: MIT Press.

Baxter, J., and Bartlett, P. L. 1999. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Computer Sciences Laboratory, Australian National University.

Grudic, G. Z., and Ungar, L. H. 2000. Localizing policy gradient estimates to action transitions. In *Forthcoming*.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Konda, V. R., and Tsitsiklis, J. N. 2000. Actor-critic algorithms. In Solla, S. A.; Leen, T. K.; and Mller, K.-R., eds., *Advances in Neural Information Processing Systems*, volume 12. Cambridge, MA: MIT Press.

Kuipers, B., and Åström, K. J. 1994. The composition and validation of heterogeneous control laws. *Automatica* 30(2):233–249.

Lainiotis, D. G. 1976. A unifying framework for adaptive systems, i: Estimation, ii. *Proceedings of the IEEE* 64(8):1126–1134, 1182–1197.

Narendra, K. S.; Balakrishnan, J.; and Ciliz, K. 1995. Adaptation and learning using multiple models, switching and tuning. *IEEE Control Systems Magazine* 15(3):37–51.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. In Solla, S. A.; Leen, T. K.; and Mller, K.-R., eds., *Advances in Neural Information Processing Systems*, volume 12. Cambridge, MA: MIT Press.

Williams, R. J. 1987. A class of gradient-estimating algorithms for reinforcement learning in neural networks. In *Proceedings of the IEEE First International Conference on Neural Networks*.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8(3):229–256.

The Linear Programming Approach to Approximate Dynamic Programming

Benjamin Van Roy
(bvr@stanford.edu)

Abstract

I will discuss an efficient method based on linear programming for approximating solutions to large scale dynamic programming problems. The approach ``fits" a linear combination of pre-selected basis functions to the dynamic programming cost-to-go function. I will present error bounds that offer performance guarantees and also guide the selection of both basis functions and ``state--relevance weights" that influence quality of the approximation. Experimental results in the domain of queueing network control provide empirical support for the methodology.

The Linear Programming Approach to Approximate Dynamic Programming

D.P. de Farias and B. Van Roy
Department of Management Science and Engineering
Stanford University, Stanford, CA 94305-4023
{pucci,bvr}@stanford.edu

Abstract

The curse of dimensionality gives rise to prohibitive computational requirements that render infeasible the exact solution of large-scale stochastic control problems. We study an efficient method based on linear programming for approximating solutions to such problems. The approach "fits" a linear combination of pre-selected basis functions to the dynamic programming cost-to-go function. We develop error bounds that offer performance guarantees and also guide the selection of both basis functions and "state-relevance weights" that influence quality of the approximation. Experimental results in the domain of queueing network control provide empirical support for the methodology.

1 Introduction

Dynamic programming offers a unified approach to solving problems of stochastic control. Central to the methodology is the cost-to-go function, which is obtained via solving Bellman's equation. The domain of the cost-to-go function is the state space of the system to be controlled, and dynamic programming algorithms compute and store a table consisting of one cost-to-go value per state. Unfortunately, the size of a state space typically grows exponentially in the number of state variables. Known as the *curse of dimensionality*, this phenomenon renders dynamic programming intractable in the face of problems of practical scale.

One approach to dealing with this difficulty is to generate an approximation within a parameterized class of functions, in a spirit similar to that of statistical regression. In particular, to approximate a cost-to-go function J^* mapping a state space $\mathcal{S}$ to reals, one would design a parameterized class of functions $\tilde{J} : \mathcal{S} \times \Re^K \mapsto \Re$, and then compute a parameter vector $r \in \Re^K$ to "fit" the cost-to-go function; i.e., so that

$$\tilde{J}(\cdot, r) \approx J^*.$$

Note that there are two important preconditions to the development of an effective approximation. First, we need to choose a parameterization $\tilde{J}$ that can closely approximate the desired cost-to-go function. In this respect, a suitable choice requires some practical experience or theoretical analysis that provides rough information on the shape of the function to be approximated. "Regularities" associated with the function, for example, can guide the choice of representation. Second, we need an efficient algorithm that computes appropriate parameter values.

The focus of this paper is on an algorithm for computing parameters for linearly parameterized function classes. Such a class can be represented by

$$\tilde{J}(\cdot, r) = \sum_{k=1}^{K} r_k \phi_k,$$

where each ϕ_k is a "basis function" mapping $\mathcal{S}$ to $\Re$ and the parameters $r_1, \ldots, r_K$ represent basis function weights. The algorithm we study is based on a linear programming formulation, originally proposed by Schweitzer and Seidman [20], that generalizes the linear programming approach to exact dynamic programming [4, 8, 9, 10, 13, 16]. Though the basic algorithm was proposed by Schweitzer and Seidman more than fifteen years ago, the analysis of errors associated with the algorithm remains an open issue. Also, there is no evidence pointing to its viability as a practical methodology for large scale dynamic programming. The only published experimental results to date involve low-dimensional problems [22, 18].

The main contribution of this paper is an error bound that characterizes the quality of approximations produced by the linear programming approach. The error is characterized in relative terms, compared against the "best possible" approximation of the optimal cost-to-go function given the selection of basis functions. This is the first such error bound not only for the linear programming approach but also for any algorithm that approximates cost-to-go functions of general stochastic control problems by computing weights for arbitrary collections of basis functions.

In addition to providing performance guarantees, the error bound and associated analysis offer new interpretations and insights pertaining to the linear programming approach. Among other things, this understanding guides selection of "state-relevance weights" that heavily influence the quality of the approximation and are critical to practical use of the methodology. In addition, insights from the analysis offer guidance in selection of basis functions.

The paper is organized as follows. We first formulate in Section 2 the stochastic control problem under consideration and discuss linear programming approaches to exact and approximate dynamic programming. In Section 3, we discuss the significance of "state-relevance weights." Section 4 contains the main results of the paper, which offer error bounds for the algorithm, as well as associated analyses. The error bounds involve problem-dependent terms, and in Section 5, we study characteristics of these terms in examples involving queueing networks. Presented in Section 6 are experimental results involving problems of queueing network control. A final section offers closing remarks, including a discussion of merits of the linear programming approach relative to other methods for approximate dynamic programming.

2 Stochastic Control and Linear Programming

We consider discrete-time stochastic control problems involving a finite state space $\mathcal{S}$ of cardinality $|\mathcal{S}| = N$. For each state $x \in \mathcal{S}$, there is a finite set of available actions $\mathcal{A}_x$. Taking action $a \in \mathcal{A}_x$ when the current state is x incurs cost $g_a(x)$. State transition probabilities $p_a(x, y)$ represent, for each pair (x, y) of states and each action $a \in \mathcal{A}_x$, the probability that the next state will be y given that the current state is x and the current action is $a \in \mathcal{A}_x$.

A *policy* u is a mapping from states to actions. Given a policy u, the dynamics of the system follow a Markov chain with transition probabilities $p_{u(x)}(x, y)$. For each policy u, we define a transition matrix P_u whose (x, y)th entry is $p_{u(x)}(x, y)$.

The problem of stochastic control amounts to selection of a policy that optimizes a given criterion. In this paper, we will employ as an optimality criterion infinite-horizon discounted cost of the form

$$J_u(x) = \mathrm{E}\left[\sum_{t=0}^{\infty} \alpha^t g_u(x_t) \middle| x_0 = x\right],$$

where $g_u(x)$ is used as shorthand for $g_{u(x)}(x)$ and the discount factor $\alpha \in (0, 1)$ reflects intertemporal preferences. It is well known that there exists a single policy u that minimizes $J_u(x)$ simultaneously for all x, and the goal is to identify that policy.

Let us define operators T_u and T by

$$T_u J = g_u + \alpha P_u J \quad \text{and} \quad TJ = \min_u \left(g_u + \alpha P_u J\right),$$

where the minimization is carried out component-wise. Dynamic programming involves solution of Bellman's equation

$$J = TJ.$$

The unique solution J^* of this equation is the optimal cost-to-go function

$$J^* = \min_u J_u,$$

and optimal control actions can be generated based on this function, according to

$$u(x) = \operatorname*{argmin}_{a \in \mathcal{A}_x} \left(g_a(x) + \alpha \sum_{y \in \mathcal{S}} p_a(x,y) J^*(y) \right).$$

Dynamic programming offers a number of approaches to solving Bellman's equation. One of particular relevance to our paper makes use of linear programming, as we will now discuss. Consider the problem

$$\begin{array}{ll} \max & c'J \\ \text{s.t.} & TJ \geq J, \end{array} \tag{1}$$

where c is a vector with positive components, which we will refer to as *state-relevance weights*. It can be shown that any feasible J satisfies $J \leq J^*$. It follows that, for any set of positive weights c, J^* is the unique solution to (1).

Note that T is a nonlinear operator, and therefore the constrained optimization problem written above is not a linear program. However, it is easy to reformulate the constraints to transform the problem into a linear program. In particular, noting that each constraint

$$(TJ)(x) \geq J(x)$$

is equivalent to a set of constraints

$$g_a(x) + \alpha \sum_{y \in \mathcal{S}} p_a(x,y) J(y) \geq J(x) \qquad \forall a \in \mathcal{A}_x,$$

we can rewrite the problem as

$$\begin{array}{ll} \max & c'J \\ \text{s.t.} & g_a(x) + \alpha \sum_{y \in \mathcal{S}} p_a(x,y) J(y) \geq J(x), \quad \forall x \in S, a \in \mathcal{A}_x. \end{array}$$

We will refer to this problem as the *exact LP*.

As mentioned in the introduction, state spaces for practical problems are enormous due to the curse of dimensionality. Consequently, the linear program of interest involves prohibitively large numbers of variables and constraints. The approximation algorithm we study reduces dramatically the number of variables.

Let us now introduce the linear programming approach to approximate dynamic programming. Given pre-selected basis functions $\phi_1, \ldots, \phi_K$, define a matrix

$$\Phi = \begin{bmatrix} | & & | \\ \phi_1 & \vdots & \phi_K \\ | & & | \end{bmatrix}.$$

With an aim of computing a weight vector $\tilde{r} \in \Re^K$ such that $\Phi\tilde{r}$ is a close approximation to J^*, one might pose the following optimization problem

$$\begin{array}{ll} \max & c'\Phi r \\ \text{s.t.} & T\Phi r \geq \Phi r. \end{array} \qquad (2)$$

Given a solution $\tilde{r}$, one might then hope to generate near-optimal decisions according to

$$u(x) = \operatorname*{argmin}_{a \in \mathcal{A}_x} \left(g_a(x) + \alpha \sum_{y \in \mathcal{S}} p_a(x,y)(\Phi\tilde{r})(y) \right).$$

We will call such a policy a *greedy* policy with respect to $\Phi\tilde{r}$. More generally, a greedy policy u with respect to a function J is one that satisfies

$$u(x) = \operatorname*{argmin}_{a \in \mathcal{A}_x} \left(g_a(x) + \alpha \sum_{y \in \mathcal{S}} p_a(x,y)J(y) \right).$$

As with the case of exact dynamic programming, the optimization problem (2) can be recast as a linear program

$$\begin{array}{ll} \max & c'\Phi r \\ \text{s.t.} & g_a(x) + \alpha \displaystyle\sum_{y \in \mathcal{S}} p_a(x,y)(\Phi r)(y) \geq (\Phi r)(x), \quad \forall x \in S, a \in \mathcal{A}_x. \end{array}$$

We will refer to this problem as the *approximate LP*. Note that, though the number of variables is reduced to K, the number of constraints remains as large as in the exact LP. Fortunately, most of the constraints become inactive, and solutions to the linear program can be approximated efficiently. In numerical studies presented in Section 6, for example, we sample and use only a relatively small subset of the constraints. We expect that subsampling in this way suffices for most practical problems, and we are developing in current work sample-complexity bounds that qualify this expectation. There are also alternative approaches studied in the literature for alleviating the need to consider all constraints. Examples include heuristics presented in [22] and problem-specific approaches making use of cutting-planes methods (e.g., [12]).

In the next three sections, we assume that the approximate LP can be solved, and we study the quality of the solution as an approximation to the cost-to-go function.

3 The Importance of State-Relevance Weights

In the exact LP, for any vector c with positive components, maximizing $c'J$ yields J^*. In other words, the choice of state-relevance weights does not influence the solution. The same statement does not hold for the approximate LP. In fact, as we will demonstrate later in this section, the choice of state-relevance weights bears a significant impact on the quality of the resulting approximation.

To motivate the role of state-relevance weights, let us start with a lemma that offers an interpretation of their function in the approximate LP. This lemma makes use of a norm $\|\cdot\|_{1,c}$, defined by

$$\|J\|_{1,c} = \sum_{x \in \mathcal{S}} c(x)|J(x)|.$$

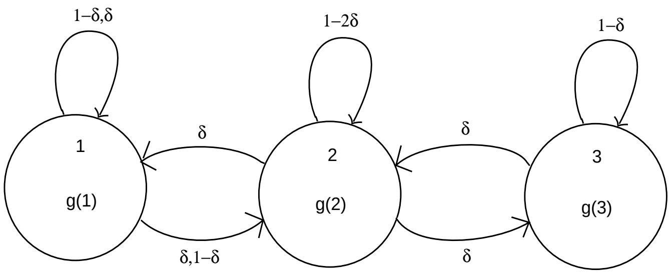


Figure 1: Markov decision process for Example 3.1.

Lemma 3.1 *A vector $\tilde{r}$ solves*

$$\begin{array}{ll} \max & c'\Phi r \\ \text{s.t.} & T\Phi r \geq \Phi r, \end{array}$$

if and only if it solves

$$\begin{array}{ll} \min & \|J^* - \Phi r\|_{1,c} \\ \text{s.t.} & T\Phi r \geq \Phi r. \end{array}$$

Proof: It is well known that the dynamic programming operator T is monotonic. From this and the fact that T is a contraction with fixed point J^*, it follows that, for any J with $J \leq TJ$, we have

$$J \leq TJ \leq T^2 J \leq ... \leq J^*.$$

Hence, any r that is a feasible solution to the optimization problems of interest satisfies $\Phi r \leq J^*$. It follows that

$$\|J^* - \Phi r\|_{1,c} = \sum_{x \in \mathcal{S}} c(x)|J^*(x) - (\Phi r)(x)| = c'J^* - c'\Phi r,$$

and maximizing $c'\Phi r$ is therefore equivalent to minimizing $\|J^* - \Phi r\|_{1,c}$. □

The preceding lemma points to an interpretation of the approximate LP as the minimization of a certain weighted norm, with weights equal to the state-relevance weights. This suggests that c imposes a tradeoff in the quality of the approximation across different states, and we can lead the algorithm to generate better approximations in a region of the state space by assigning relatively larger weight to that region. In the next examples, we identify states that should be weighted heavily to improve performance of the policy generated by the approximate LP. An understanding of the characteristics of these states can guide the choice of state-relevance weights.

The first example provides some insight into the behavior of the approximate LP when cost-to-go values are much higher in some regions of the state space than in others. Such a situation is likely to arise when the state space is large.

3.1 Example: States with Very Large Costs

Consider the problem illustrated in Figure 1. The node labels indicate state indices and associated costs. There are three states, 1, 2 and 3, with costs $g(1) < g(2) \ll g(3)$ which do not

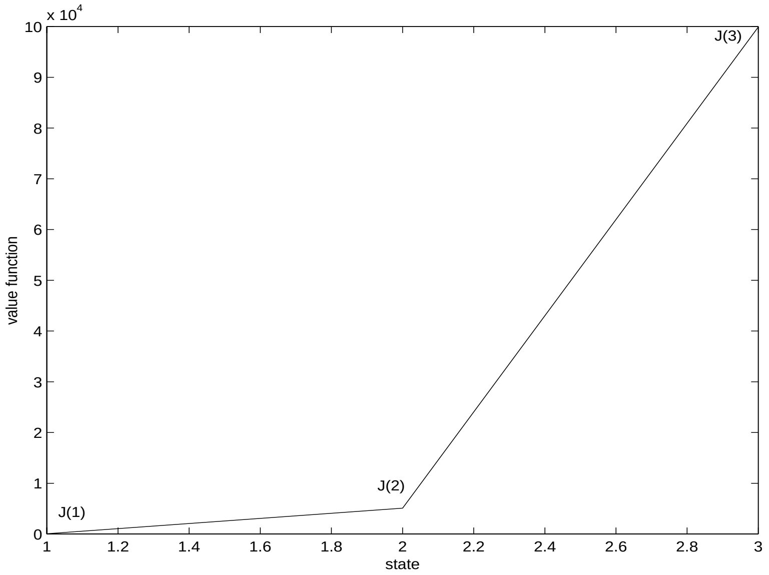


Figure 2: Typical cost-to-go function for $g(1) < g(2) \ll g(3)$.

depend on the action taken. There is one available action for states 2 and 3, and two actions for state 1. We assume that the transition probability δ is small, for instance,

$$\delta = \frac{1-\alpha}{g(3)}.$$

Clearly, it is optimal to take the first action in state 1, which makes the probability of remaining in that state equal to $1-\delta$ and the probability of going to state 2 equal to δ. We can calculate the cost-to-go function:

$$J^* = \left[\begin{array}{c} \frac{g(1)}{(1-\alpha)(1+\alpha/g(3))} + \frac{\alpha}{g(3)+\alpha} J_2^* \\ \frac{(1+\alpha/g(3))g(2)}{(1+3\alpha/g(3))(1-\alpha)} + \frac{\alpha(g(1)+g(3))}{(1-\alpha)(g(3)+3\alpha)} \\ \frac{g(3)}{(1-\alpha)(1+\alpha/g(3))} + \frac{\alpha}{g(3)+\alpha} J_2^* \end{array} \right] \approx \left[\begin{array}{c} \frac{g(1)}{1-\alpha} \\ \frac{g(2)+\alpha}{1-\alpha} \\ \frac{g(3)}{1-\alpha} \end{array} \right].$$

The cost-to-go function J^* corresponding to $g(1) = 0.5$, $g(2) = 50$, $g(3) = 1000$ and $\alpha = 0.99$ is plotted in Figure 2.

Let

$$\Phi = \left[\begin{array}{cc} g(1) & 1 \\ g(2) & 1 \\ 2g(2) - g(3) & 1 \end{array} \right].$$

The constraints of the approximate LP are given by

$$(1-\alpha) \left[\begin{array}{cc} g(1) + \frac{\alpha(g(1)-g(2))}{g(3)} & 1 \\ \frac{g(1)-\alpha g(2)}{1-\alpha} + \frac{\alpha(g(2)-g(1))}{g(3)} & 1 \\ g(2) + \alpha - \frac{\alpha g(1)}{g(3)} & 1 \\ 2g(2) - g(3) - \alpha + \frac{\alpha g(2)}{g(3)} & 1 \end{array} \right] \left[\begin{array}{c} r_1 \\ r_2 \end{array} \right] \leq \left[\begin{array}{c} g(1) \\ g(1) \\ g(2) \\ g(3) \end{array} \right].$$

The constraint boundaries intersect the axes at

$$\left(0, \frac{g(1)}{1-\alpha}\right) \text{ and } \left(\frac{g(1)}{(1-\alpha)\left(g(1)+\frac{\alpha(g(1)-g(2))}{g(3)}\right)}\right) \approx \left(\frac{1}{1-\alpha}, 0\right);$$

$$\left(0, \frac{g(1)}{1-\alpha}\right) \text{ and } \left(\frac{g(1)}{g(1)-\alpha g(2)+(1-\alpha)\frac{\alpha(g(2)-g(1))}{g(3)}}, 0\right) \approx (-\epsilon, 0);$$

$$\left(0, \frac{g(2)}{1-\alpha}\right) \text{ and } \left(\frac{1}{(1-\alpha)\left(1+\frac{\alpha}{g(2)}-\frac{\alpha g(1)}{g(2)g(3)}\right)}, 0\right) \approx \left(\frac{1-\epsilon}{1-\alpha}, 0\right);$$

$$\left(0, \frac{g(3)}{1-\alpha}\right) \text{ and } \left(\frac{g(3)}{(1-\alpha)\left(2g(2)-g(3)-\alpha+\frac{\alpha g(2)}{g(3)}\right)}, 0\right) \approx (-M, 0),$$

where ϵ represents a relatively small positive number and M represents a relatively large positive number. The feasible region for a particular set of parameter values is illustrated in Figure 3.

Let us make some observations that are based on the figure but hold true more generally so long as $g(1) < g(2) \ll g(3)$ and δ is small. First, constraint 4 is never active in the positive quadrant, and the only relevant extreme points are at the intersection of constraints 1 and 2 (which is $\eta_1 = (0, \frac{g(1)}{1-\alpha})$) and the intersection of constraints 1 and 3 (which is $\eta_2 \approx (\frac{1}{1-\alpha}, 0)$). Note that the solution is bounded for all positive state-relevance weights c, since $\Phi r \leq J^*$ for all feasible r. Hence, η_1 and η_2 are the only possible solutions. The solution η_2 leads to the best policy. The solution η_1, on the other hand, does not.

It turns out that the choice of state-relevance weights can lead us to either η_1 or η_2. In particular, since

$$c'\Phi r = (c_1 g(1) + g(2)(c_2 + 2c_3) - c_3 g(3))r_1 + r_2,$$

when c_3 is relatively large, η_1 is the optimal solution, while η_2 is the optimal solution when c_3 is small. The shape of the cost-to-go function in Figure 2 explains this behavior: the cost-to-go at state 3 is much higher than that at states 1 and 2, hence there is potentially more opportunity for increasing the objective function of the approximate LP by increasing Φr at state 3 than at states 1 or 2. Therefore it appears that, unless we discount state 3 more heavily than the others, the approximate LP will approximate $J^*(3)$ very closely at the expense of large errors in approximating $J^*(1)$ and $J^*(2)$.

3.2 Example: Weights Induced by Steady-State Probabilities

The preceding example illustrates how the approximate LP is likely to yield poor approximations for states with relatively low cost-to-go unless these states are emphasized by state-relevance weights. The next example further develops an understanding of state-relevance weights through a numerical study that points towards choosing state-relevance weights that reflect steady-state probabilities that would be induced if the system were to be controlled by a "good" policy. Indeed, in approximating the solution to a given stochastic control problem, it seems sensible to weight more heavily portions of the state space that are visited frequently, so that accuracy will be emphasized in such regions. This seems particularly motived in large-scale problems, in which desirable policies often exhibit some form of "stability," guiding the system to a limited region of the state space and allowing only infrequent excursions from this region.

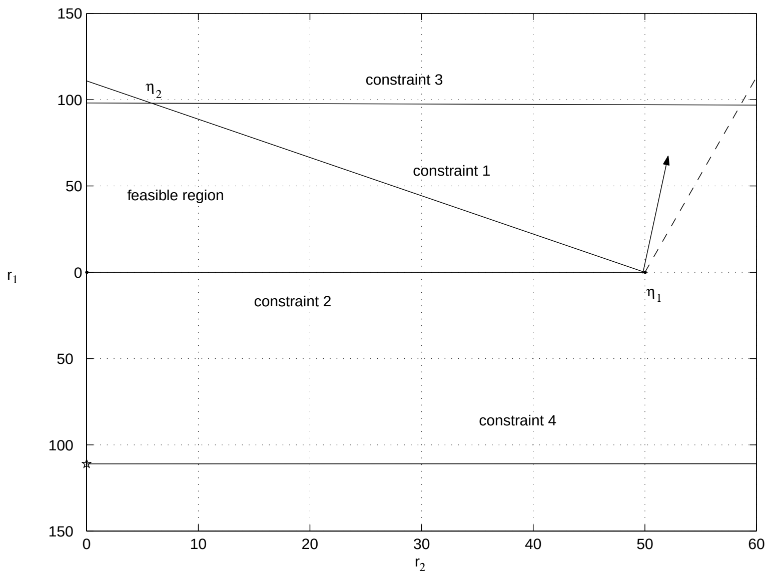


Figure 3: Typical feasible region for $g(1) < g(2) << g(3)$. The arrow represents $c'\Phi$ for a state-relevance weight c which yields a "good" solution for the approximate LP.

Consider the problem illustrated in Figure 4. The node labels indicate state indices and associated costs. There are four states and costs for each state do not depend on the action taken. There is one available action for states 1 and 3, and two available actions for states 2 and 4. The arc labels indicate transition probabilities. Clearly, it is optimal to take the first action in both states 2 and 4. Let

$$\Phi = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & -15 \\ 1 & 100 \end{bmatrix}$$

and $\alpha = 0.99$. One can solve the approximate LP with $c = e$ (e is the vector with every component equal to 1) to obtain an approximation to the cost-to-go function Φr^e, and then generate a policy u^e that is greedy with respect to Φr^e. This policy takes the right action in state 4 but the wrong action in state 2. The resulting costs-to-go are given by

$$J_{u^e} = \begin{bmatrix} 1560.9 \\ 2935.5 \\ 2978.3 \\ 3565.6 \end{bmatrix}.$$

Let us now discuss how more appropriate state-relevance weights might be chosen and how they can influence the outcome. One possible measure of relevance is given by the stationary distribution induced by an optimal policy. In this simple example, such a distribution is easily calculated and is given by [0.98895 0.00999 0.00100 0.00006]. In general, however, it is difficult to compute such a distribution because the computation typically requires knowledge of an optimal policy.

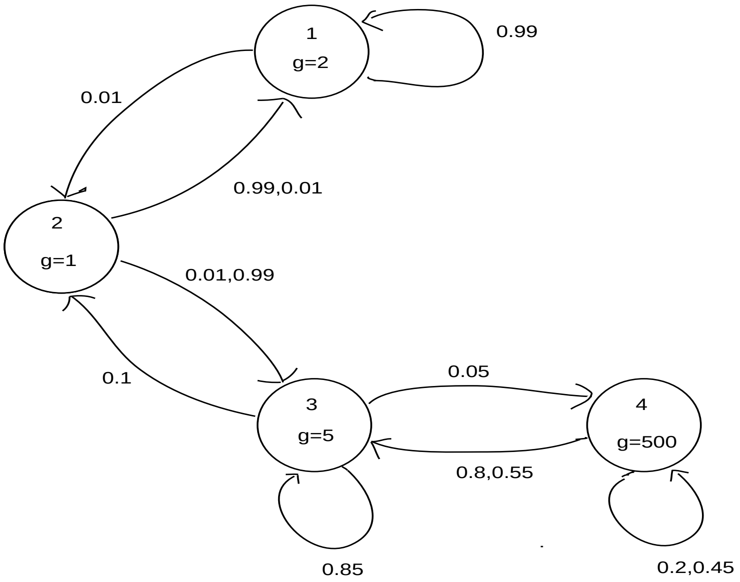


Figure 4: Markov decision process for Example 3.2.

Is there an alternative approach to generating a good set of weights c? Note that state 4 has a much higher cost than the other states, and we might therefore argue that an optimal policy would try to avoid this state. In addition, based on the pattern of transition probabilities, we can identify the region comprised of states 1, 2 and 3 as "stable," whereas state 4 is "unstable" under any policy; in particular, all actions in state 4 involve a transition with relatively high probability to state 3. This motivates choosing a lower weight for state 4 than for other states. For example, one might try $c = [1\ 1\ 1\ 0.6]$. Denote the solution to the approximate LP by r^c. It turns out that the greedy policy u^c with respect to Φr^c takes the right action in state 2 and the wrong action in state 4, and the cost-to-go for this policy is

$$J_{u^c} = \begin{bmatrix} 203.2 \\ 206.5 \\ 637.7 \\ 1527.9 \end{bmatrix}$$

The policy u^e bears much higher average cost (28.9066) than u^c (2.0382).

We have seen that the state-relevance weights can influence the quality of an approximation. It is not clear how to find good weights in an efficient manner for larger problems. However, our examples do point to two heuristics for selecting weights. First, it seems important to assign low weights to states with high costs, as suggested by Example 3.1. Second, if the system when operated by a good policy spends most of its time in a certain subset of the state space, it seems that this subset should be weighted heavily, as illustrated by Example 3.2. We expect that these heuristics will not generally conflict. In particular, in realistic large-scale problems, states with high costs should be avoided, and the system should spend most of its time in a low-cost region of the state space.

4 Error Bounds for the Approximate LP

When the optimal cost-to-go function lies within the span of the basis functions, solution of the approximate LP yields the exact optimal cost-to-go function. Unfortunately, it is difficult in practice to select a set of basis functions that contains the optimal cost-to-go function within its span. Instead, basis functions must be based on heuristics and simplified analyses. One can only hope that the span comes close to the desired cost-to-go function.

For the approximate LP to be useful, it should deliver good approximations when the cost-to-go function is near the span of selected basis functions. In this section, we develop bounds that ensure desirable results of this kind. We begin in Section 4.1 with a simple bound capturing the fact that, if e is within the span of the basis functions, the error in the result of the approximate LP is proportional to the minimal error given the selected basis functions. Though this result is interesting in its own right, the bound is very loose – perhaps too much so to be useful in practical contexts. In Section 4.2, however, we remedy this situation by providing a tighter bound, which constitutes the main result of the paper.

4.1 A Simple Bound

Let $\|\cdot\|_\infty$ denote the maximum norm, defined by $\|J\|_\infty = \max_{x\in\mathcal{S}} |J(x)|$, and recall that e denotes the vector with every component equal to 1. Our first bound is given by the following theorem.

Theorem 4.1 *Let e be in the span of the columns of Φ and c be a probability distribution. Then, if $\tilde{r}$ is an optimal solution to the approximate LP,*

$$\|J^* - \Phi\tilde{r}\|_{1,c} \leq \frac{2}{1-\alpha} \min_r \|J^* - \Phi r\|_\infty.$$

This bound establishes that when the optimal cost-to-go function lies close to the span of the basis functions, the approximate LP generates a good approximation. In particular, if the error $\min_r \|J^* - \Phi r\|_\infty$ goes to zero (e.g., as we make use of more and more basis functions) the error resulting from the approximate LP also goes to zero.

Though the above bound offers some support for the linear programming approach, there are some significant weaknesses:

1. The bound calls for an element of the span of the basis functions to exhibit uniformly low error over all states. In practice, however, $\min_r \|J^* - \Phi r\|_\infty$ is typically huge, especially for large-scale problems.
2. The bound does not take into account the choice of state-relevance weights. As demonstrated in the previous section, these weights can significantly impact the approximation error. A sharp bound should take them into account.

In Section 4.2, we will state and prove the main result of this paper, which provides an improved bound that aims to alleviate the shortcomings listed above. First, we prove Theorem 4.1.

Proof of Theorem 4.1

Let r^* be one of the vectors minimizing $\|J^* - \Phi r\|_\infty$ and define $\epsilon = \|J^* - \Phi r^*\|_\infty$. The first step is to find a feasible point $\bar{r}$ such that $\Phi\bar{r}$ is within distance $O(\epsilon)$ of J^*. Since

$$\|T\Phi r^* - J^*\|_\infty \leq \alpha\|\Phi r^* - J^*\|_\infty,$$

we have

$$T\Phi r^* \geq J^* - \alpha\epsilon e. \tag{3}$$

We also recall that for any vector J and any scalar k,

$$\begin{aligned} T(J - ke) &= \min_u \{g_u + \alpha P_u(J - ke)\} \\ &= \min_u \{g_u + \alpha P_u J - \alpha k e\} \\ &= \min_u \{g_u + \alpha P_u J\} - \alpha k e \\ &= TJ - \alpha k e. \end{aligned} \tag{4}$$

Combining (3) and (4), we have

$$\begin{aligned} T(\Phi r^* - ke) &= T\Phi r^* - \alpha k e \\ &\geq J^* - \alpha\epsilon e - \alpha k e \\ &\geq \Phi r^* - (1+\alpha)\epsilon e - \alpha k e \\ &= \Phi r^* - ke + [(1-\alpha)k - (1+\alpha)\epsilon]\, e. \end{aligned}$$

Since e is within the span of the columns of Φ, there exists a vector $\bar{r}$ such that

$$\Phi\bar{r} = \Phi r^* - \frac{(1+\alpha)\epsilon}{1-\alpha} e,$$

and $\bar{r}$ is a feasible solution to the approximate LP. By the triangular inequality,

$$\|\Phi\bar{r} - J^*\|_\infty \leq \|J^* - \Phi r^*\|_\infty + \|\Phi r^* - \Phi\bar{r}\|_\infty \leq \epsilon\left(1 + \frac{1+\alpha}{1-\alpha}\right) = \frac{2\epsilon}{1-\alpha}.$$

If $\tilde{r}$ is an optimal solution to the approximate LP, by Lemma 3.1, we have

$$\begin{aligned} \|J^* - \Phi\tilde{r}\|_{1,c} &\leq \|J^* - \Phi\bar{r}\|_{1,c} \\ &\leq \|J^* - \Phi\bar{r}\|_\infty \\ &\leq \frac{2\epsilon}{1-\alpha} \end{aligned}$$

where the second inequality holds because c is a probability distribution. The result follows. □

4.2 An Improved Bound

To set the stage for development of an improved bound, let us establish some notation. First, we introduce a weighted maximum norm, defined by

$$\|J\|_{\infty,\gamma} = \max_{x \in \mathcal{S}} \gamma(x)|J(x)|, \tag{5}$$

for any $\gamma : \mathcal{S} \mapsto \Re^+$. As opposed to the maximum norm employed in Theorem 4.1, this norm allows for uneven weighting of errors across the state space.

We also introduce an operator H, defined by

$$(HV)(x) = \max_{a \in \mathcal{A}_x} \sum_y P_a(x,y)V(y),$$

for all $V : \mathcal{S} \mapsto \Re$. For any V, $(HV)(x)$ represents the maximum expected value of $V(y)$ if the current state is x and y is a random variable representing the next state. Based on this operator, we define a scalar

$$k_V = \max_x \frac{V(x)}{V(x) - \alpha(HV)(x)}, \tag{6}$$

for each $V : \mathcal{S} \mapsto \Re$.

We interpret the argument V of H as a "Lyapunov function," while we view k_V as a "Lyapunov stability factor," in a sense that we will now explain. In the upcoming theorem, we will only be concerned with functions V that are positive and that make k_V nonnegative. Also, our error bound for the approximate LP will grow proportionately with k_V, and we therefore want k_V to be small. At a minimum, k_V should be finite, which translates to a condition

$$\alpha(HV)(x) < V(x), \qquad \forall x \in \mathcal{S}. \tag{7}$$

If α were equal to 1, this would look like a Lyapunov stability condition: the maximum expected value $(HV)(x)$ at the next time step must be less than the current value $V(x)$. In general, α is less than 1, and this introduces some slack in the condition. Note also that k_V becomes smaller as the $(HV)(x)$'s become small relative to the $V(x)$'s. Hence, k_V conveys a degree of "stability," with smaller values representing stronger stability.

We are now ready to state our main result. For any given function V mapping $\mathcal{S}$ to positive reals, we use $1/V$ as shorthand for a function $x \mapsto 1/V(x)$.

Theorem 4.2 *Let $\tilde{r}$ be a solution of the approximate LP. Then, for any $v \in \Re^K$ such that $(\Phi v)(x) > 0$ for all $x \in \mathcal{S}$ and $\alpha H \Phi v < \Phi v$,*

$$\|J^* - \Phi\tilde{r}\|_{1,c} \leq 2k_{\Phi v}(c'\Phi v) \min_r \|J^* - \Phi r\|_{\infty, 1/\Phi v}. \tag{8}$$

Let us now discuss how this new theorem addresses the shortcomings of Theorem 4.1 listed in the previous section. We treat in turn the two items from the aforementioned list.

1. The norm $\|\cdot\|_\infty$ appearing in Theorem 4.1 is undesirable largely because it does not scale well with problem size. In particular, for large problems, the cost-to-go function can take on huge values over some (possibly infrequently visited) regions of the state space, and so can approximation errors in such regions.

   Observe that the maximum norm of Theorem 4.1 has been replaced in Theorem 4.2 by $\|\cdot\|_{\infty,1/\Phi v}$. Hence, the error at each state is now weighted by the reciprocal of the Lyapunov function value. This should to some extent alleviate difficulties arising in large problems. In particular, the Lyapunov function should take on large values in undesirable regions of the state space – regions where J^* is large. Hence, division by the Lyapunov function acts as a normalizing procedure that scales down errors in such regions.

2. As opposed to the bound of Theorem 4.1, the state-relevance weights do appear in our new bound. In particular, there is a coefficient $c'\Phi v$ scaling the right-hand-side. In general, if the state-relevance weights are chosen appropriately, we expect that this factor of $c'\Phi v$ will be reasonably small and independent of problem size. We defer to Section 5 further qualification of this statement and a discussion of approaches to choosing c in contexts posed by concrete examples.

Proof of Theorem 4.2

The remainder of this section is dedicated to a proof of Theorem 4.2. We begin with a preliminary lemma bounding the effects of applying the dynamic programming operator to two different functions.

Lemma 4.1 *For any J and $\overline{J}$,*

$$|TJ - T\bar{J}| \leq \alpha \max_u P_u |J - \bar{J}|.$$

Proof: Note that, for any J and $\bar{J}$,

$$\begin{aligned}
TJ - T\bar{J} &= \min_u \{g_u + \alpha P_u J\} - \min_u \left\{g_u + \alpha P_u \bar{J}\right\} \\
&= g_{u^J} + \alpha P_{u^J} J - g_{u^{\bar{J}}} - \alpha P_{u^{\bar{J}}} \bar{J} \\
&\leq g_{u^{\bar{J}}} + \alpha P_{u^{\bar{J}}} J - g_{u^{\bar{J}}} - \alpha P_{u^{\bar{J}}} \bar{J} \\
&\leq \alpha \max_u P_u (J - \bar{J}) \\
&\leq \alpha \max_u P_u |J - \bar{J}|,
\end{aligned}$$

where u^J and $u^{\bar{J}}$ denote greedy policies with respect to J and $\bar{J}$, respectively. An entirely analogous argument gives us

$$T\bar{J} - TJ \leq \alpha \max_u P_u |J - \bar{J}|,$$

and the result follows. □

Based on the above lemma, we can place the following bound on constraint violations in the approximate LP.

Lemma 4.2 *For any vector V with positive components and any vector J,*

$$TJ \geq J - (\alpha HV + V) \|J^* - J\|_{\infty,1/V}. \tag{9}$$

Proof: Note that

$$|J^*(x) - J(x)| \leq \|J^* - J\|_{\infty,1/V} V(x).$$

By Lemma 4.1,

$$\begin{aligned}
|(TJ^*)(x) - (TJ)(x)| &\leq \alpha \max_a \sum_y P_a(x,y) |J^*(y) - J(y)| \\
&\leq \alpha \|J^* - J\|_{\infty,1/V} \max_{a \in \mathcal{A}_x} \sum_y P_a(x,y) V(y) \\
&= \alpha \|J^* - J\|_{\infty,1/V} (HV)(x).
\end{aligned}$$

Letting $\epsilon = \|J^* - J\|_{\infty,1/V}$, it follows that

$$\begin{aligned}
(TJ)(x) &\geq J^*(x) - \alpha\epsilon(HV)(x) \\
&\geq J(x) - \epsilon V(x) - \alpha\epsilon(HV)(x).
\end{aligned}$$

The result follows. □

The next lemma establishes that subtracting an appropriately scaled version of a Lyapunov function from Φr^* leads us to the feasible region of the approximate LP.

Lemma 4.3 *Let v be a vector satisfying $(\Phi v)(x) > 0$ for all $x \in \mathcal{S}$ and $\alpha H \Phi v < \Phi v$, let r^* be a vector minimizing $\|J^* - \Phi r\|_{\infty,1/\Phi v}$, and let*

$$\overline{r} = r^* - \|J^* - \Phi r^*\|_{\infty,1/\Phi v}(2k_{\Phi v} - 1)v.$$

Then,

$$T\Phi\overline{r} \geq \Phi\overline{r}.$$

Proof: Let $\epsilon = \|J^* - \Phi r^*\|_{\infty,1/\Phi v}$. By Lemma 4.1,

$$\begin{aligned} |(T\Phi r^*)(x) - (T\Phi\overline{r})(x)| &= |(T\Phi r^*)(x) - (T(\Phi r^* - \epsilon(2k_{\Phi v} - 1)\Phi v))(x)| \\ &\leq \alpha \max_a \sum_y P_a(x,y)\epsilon(2k_{\Phi v} - 1)(\Phi v)(y) \\ &= \alpha\epsilon(2k_{\Phi v} - 1)(H\Phi v)(x), \end{aligned}$$

since $k_V > 1$ for all $V > 0$ such that $\alpha H V < V$, hence in particular $2k_{\Phi v} - 1 > 0$. It follows that

$$T\Phi\overline{r} \geq T\Phi r^* - \alpha\epsilon(2k_{\Phi v} - 1)H\Phi v.$$

By Lemma 4.2,

$$T\Phi r^* \geq \Phi r^* - \epsilon\left(\alpha H\Phi v + \Phi v\right),$$

and therefore

$$\begin{aligned} T\Phi\overline{r} &\geq \Phi r^* - \epsilon\left(\alpha H\Phi v + \Phi v\right) - \alpha\epsilon(2k_{\Phi v} - 1)H\Phi v \\ &= \Phi\overline{r} - \epsilon\left(\alpha H\Phi v + \Phi v\right) + \epsilon(2k_{\Phi v} - 1)(\Phi v - \alpha H\Phi v) \\ &\geq \Phi\overline{r} - \epsilon\left(\alpha H\Phi v + \Phi v\right) + \epsilon(\Phi v + \alpha H\Phi v) \\ &= \Phi\overline{r}, \end{aligned}$$

where the last inequality follows from the fact that $\Phi v - \alpha H\Phi v > 0$ and

$$2k_{\Phi v} - 1 = \max_x \frac{(\Phi v)(x) + \alpha(H\Phi v)(x)}{(\Phi v)(x) - \alpha(H\Phi v)(x)}.$$

□

Given the preceding lemmas, we are poised to prove Theorem 4.2.

Proof of Theorem 4.2: From Lemma 4.3, we know that $\overline{r} = r^* - \|J^* - \Phi r^*\|_{\infty,1/\Phi v}k_{\Phi v}v$ is a feasible solution for the approximate LP. It follows that

$$\|\Phi\overline{r} - \Phi r^*\|_{\infty,1/\Phi v} \leq \|J^* - \Phi r^*\|_{\infty,1/\Phi v}k_{\Phi v}\|\Phi v\|_{\infty,1/\Phi v}.$$

From Lemma 3.1, we have

$$\begin{aligned} \|J^* - \Phi\tilde{r}\|_{1,c} &\leq \|J^* - \Phi\overline{r}\|_{1,c} \\ &= \sum_x c(x)(\Phi v)(x)\frac{|J^*(x) - (\Phi\overline{r})(x)|}{(\Phi v)(x)} \\ &\leq \left(\sum_x c(x)(\Phi v)(x)\right)\max_x \frac{|J^*(x) - (\Phi\overline{r})(x)|}{(\Phi v)(x)} \\ &= c'\Phi v\|J^* - \Phi\overline{r}\|_{\infty,1/\Phi v} \\ &\leq c'\Phi v\left(\|J^* - \Phi r^*\|_{\infty,1/\Phi v} + \|\Phi\overline{r} - \Phi r^*\|_{\infty,1/\Phi v}\right) \\ &\leq c'\Phi v\left(\|J^* - \Phi r^*\|_{\infty,1/\Phi v} + \|J^* - \Phi r^*\|_{\infty,1/\Phi v}(2k_{\Phi v} - 1)\|\Phi v\|_{\infty,1/\Phi v}\right) \\ &\leq 2k_{\Phi v}c'\Phi v\|J^* - \Phi r^*\|_{\infty,1/\Phi v}. \end{aligned}$$

□

5 On the Choice of Lyapunov Function

The Lyapunov function Φv plays a central role in the bound of Theorem 4.2. Its choice influences three terms on the right-hand-side of the bound:

1. the error $\min_r \|J^* - \Phi r\|_{\infty,1/\Phi v}$;
2. the Lyapunov stability factor $k_{\Phi v}$;
3. the inner product $c'\Phi v$ with the state-relevance weights.

An appropriately chosen Lyapunov function should make all three of these terms relatively small. Furthermore, for the bound to be useful in practical contexts, these terms should not grow much with problem size.

In the following subsections, we present three examples involving choices of Lyapunov functions in queueing problems. The intention is to illustrate more concretely how Lyapunov functions might be chosen and that reasonable choices lead to practical error bounds that are independent of the number of states, as well as the number of state variables. The first example involves a single autonomous queue. A second generalizes this to a context with controls. A final example treats a network of queues. In each case, we study the three terms enumerated above and how they scale with the number of states and/or state variables.

5.1 Example: An Autonomous Queue

Our first example involves a model of an autonomous (i.e., uncontrolled) queueing system. We consider a Markov process with states $0, 1, ..., N-1$, each representing a possible number of jobs in a queue. The system state x_t evolves according to

$$x_{t+1} = \begin{cases} \min(x_t + 1, N-1), & \text{with probability } p, \\ \max(x_t - 1, 0), & \text{otherwise,} \end{cases}$$

and it is easy to verify that the steady-state probabilities $\pi(0), \ldots, \pi(N-1)$ satisfy

$$\pi(x) = \pi(0)\left(\frac{p}{1-p}\right)^x.$$

If the state satisfies $0 < x < N-1$, a cost $g(x) = x^2$ is incurred. For the sake of simplicity, we assume that costs at the boundary states 0 and $N-1$ are chosen[1] to ensure that the cost-to-go function takes the form

$$J^*(x) = \rho_2 x^2 + \rho_1 x + \rho_0,$$

[1] It is easy to verify that such a choice of boundary conditions is possible. In particular, given the desired functional form for J^*, we can solve for ρ_0, ρ_1, and ρ_2, based on Bellman's equation for states $1, \ldots, N-2$:

$$J^*(x) = x^2 + \alpha(pJ^*(x+1) + (1-p)J^*(x-1), \qquad \forall x = 1, \ldots, N-2.$$

Note that the solution is unique as long as $N > 5$. We can then set $g(0) \equiv J^*(0) - \alpha(pJ^*(1) + (1-p)J^*(0))$ and $g(N-1) \equiv J^*(N-1) - \alpha(pJ^*(N-1) + (1-p)J^*(N-2))$ so that Bellman's equation is also satisfied for states 0 and $N-1$.

for some scalars ρ_0, ρ_1, ρ_2 with $\rho_0 > 0$ and $\rho_2 > 0$. We assume that $p < 1/2$ so that the system is "stable." Stability here is taken in a loose sense indicating that the steady-state probabilities are decreasing for all sufficiently large states.

Suppose that we wish to generate an approximation to the value function using the linear programming approach. Further suppose that we have chosen the state-relevance weights c to be the vector π of steady-state probabilities and the basis functions to be $\phi_1(x) = 1$ and $\phi_2(x) = x^2$.

How good can we expect the approximate cost-to-go function $\Phi\tilde{r}$ to be as we increase the number of states N? First note that

$$\begin{aligned}
\min_r \|J^* - \Phi r\|_{1,c} &\leq \|J^* - (\rho_0\phi_1 + \rho_2\phi_2)\|_{1,c} \\
&= \sum_{x=0}^{N-1} \pi(x)|\rho_1|x \\
&= |\rho_1| \sum_{x=0}^{N-1} \pi(0) \left(\frac{p}{1-p}\right)^x x \\
&\leq |\rho_1| \frac{\frac{p}{1-p}}{1 - \frac{p}{1-p}},
\end{aligned}$$

for all N. The last inequality follows from the fact that the summation in the third line corresponds to the expected value of a geometric random variable conditioned on its being less than N. Hence, $\min_r \|J^* - \Phi r\|_{1,c}$ is uniformly bounded over N. One would hope that $\|J^* - \Phi\tilde{r}\|_{1,c}$, with $\tilde{r}$ being an outcome of the approximate LP, is similarly uniformly bounded over N. It is clear that Theorem 4.1 does not offer a uniform bound of this sort. In particular, the term $\min_r \|J^* - \Phi r\|_\infty$ on the right-hand-side grows proportionately with N and is unbounded as N increases. Fortunately, this situation is remedied by Theorem 4.2, which does provide a uniform bound. In particular, as we will show in the remainder of this section, for an appropriate Lyapunov function $V = \Phi v$, the values of $\min_r \|J^* - \Phi r\|_{\infty,1/V}$, k_V and $c'V$ are all uniformly bounded over N, and together, these values offer a bound on $\|J^* - \Phi\tilde{r}\|_{1,c}$ that is uniform over N.

We will make use of a Lyapunov function

$$V(x) = x^2 + \frac{2}{1-\alpha},$$

which is clearly within the span of our basis functions ϕ_1 and ϕ_2. Given this choice, we have

$$\begin{aligned}
\min_r \|J^* - \Phi r\|_{\infty,1/V} &\leq \max_{x\geq 0} \frac{|\rho_2 x^2 + \rho_1 x + \rho_0 - \rho_2 x^2 - \rho_0|}{x^2 + 2/(1-\alpha)} \\
&= \max_{x\geq 0} \frac{|\rho_1| x}{x^2 + 2/(1-\alpha)} \\
&\leq \frac{|\rho_1|}{2\sqrt{2/(1-\alpha)}}.
\end{aligned}$$

Hence, $\min_r \|J^* - \Phi r\|_{\infty,1/V}$ is uniformly bounded over N.

We next show that k_V is uniformly bounded over N. This amounts to showing that the difference $V - \alpha HV$ is positive and not too small as compared to V. In order to do that, we first find bounds on HV in terms of V. For $0 < x < N-1$, we have

$$\alpha(HV)(x) = \alpha\left[p\left(x^2 + 2x + 1 + \frac{2}{1-\alpha}\right) + (1-p)\left(x^2 - 2x + 1 + \frac{2}{1-\alpha}\right)\right]$$

$$\begin{aligned}
&= \alpha\left[x^2 + \frac{2}{1-\alpha} + 1 + 2x(2p-1)\right] \\
&\le \alpha\left(x^2 + \frac{2}{1-\alpha} + 1\right) \\
&= V(x)\left(\alpha + \frac{\alpha}{V(x)}\right) \\
&\le V(x)\left(\alpha + \frac{1}{V(0)}\right) \\
&= V(x)\frac{1+\alpha}{2}.
\end{aligned}$$

For $x = 0$, we have

$$\begin{aligned}
\alpha(HV)(0) &= \alpha\left[p\left(1 + \frac{2}{1-\alpha}\right) + (1-p)\frac{2}{1-\alpha}\right] \\
&= \alpha p + \alpha\frac{2}{1-\alpha} \\
&\le V(0)\left(\alpha + \frac{1-\alpha}{2}\right) \\
&= V(0)\frac{1+\alpha}{2}.
\end{aligned}$$

Finally, we clearly have

$$\alpha(HV)(N-1) \le \alpha V(N-1) \le V(N-1)\frac{1+\alpha}{2},$$

since the only possible transitions from state $N-1$ are to states $x \le N-1$ and V is a nondecreasing function. Therefore,

$$\begin{aligned}
k_V &= \max_x \frac{V(x)}{V(x) - \alpha(HV)(x)} \\
&\le \max_x \frac{V(x)}{V(x) - V(x)\frac{1+\alpha}{2}} \\
&\le \frac{2}{1-\alpha},
\end{aligned}$$

and k_V is uniformly bounded for all N.

We now treat $c'V$. Note that for $N \ge 1$,

$$\begin{aligned}
c'V &= \sum_{x=0}^{N-1} \pi(0)\left(\frac{p}{1-p}\right)^x \left(x^2 + \frac{2}{1-\alpha}\right) \\
&= \frac{1 - p/(1-p)}{1 - [p/(1-p)]^N} \sum_{x=0}^{N-1} \left(\frac{p}{1-p}\right)^x \left(x^2 + \frac{2}{1-\alpha}\right) \\
&\le \frac{1 - p/(1-p)}{1 - p/(1-p)} \sum_{x=0}^{\infty} \left(\frac{p}{1-p}\right)^x \left(x^2 + \frac{2}{1-\alpha}\right) \\
&= \frac{1-p}{1-2p}\left(\frac{2}{1-\alpha} + 2\frac{p^2}{(1-2p)^2} + \frac{p}{1-2p}\right),
\end{aligned}$$

so $c'V$ is uniformly bounded for all N.

5.2 Example: A Controlled Queue

In the previous example, we treated the case of an autonomous queue and showed how the terms involved in the error bound of Theorem 4.2 were uniformly bounded on the number of states N. We now address a more general case in which we can control the queue service rate. For any time t and state $0 < x_t < N-1$, the next state is given by

$$x_{t+1} = \begin{cases} x_t - 1, & \text{with probability } q(x_t), \\ x_t + 1, & \text{with probability } p, \\ x_t, & \text{otherwise.} \end{cases}$$

From state 0, a transition to state 1 or 0 occurs with probabilities p or $1-p$, respectively. From state $N-1$, a transition to state $N-2$ or $N-1$ occurs with probabilities $q(N-2)$ or $1-q(N-2)$, respectively. The arrival probability p is the same for all states and we assume that $p < 1/2$. The action to be chosen in each state x is the departure probability or service rate $q(x)$, which takes values in a finite set $\{q_i, i = 1, ..., A\}$. We assume that $q_A = 1 - p > p$, therefore the queue is "stabilizable". The cost incurred at state x if action q is taken is given by

$$g(x,q) = x^2 + m(q),$$

where m is a nonnegative and increasing function.

As discussed before, our objective is to show that the terms involved in the error bound of Theorem 4.2 are uniformly bounded over N. We start by finding a suitable Lyapunov function based on our knowledge of the problem structure. In the autonomous case, the choice of the Lyapunov function was motivated by the fact that the cost-to-go function was a quadratic. We now proceed to show that in the controlled case, J^* can be bounded above by a quadratic

$$J^*(x) \leq \rho_2 x^2 + \rho_1 x + \rho_0$$

for some $\rho_0 > 0$, ρ_1 and $\rho_2 > 0$ that are constant independent of the queue buffer size $N-1$. Note that J^* is bounded above by the cost-to-go of a policy $\bar{\mu}$ that takes action $q(x) = 1-p$ for all x, hence it suffices to find a quadratic upper bound for the cost-to-go of this policy. We will do so by making use of the fact that for any policy μ and any vector J, $T_\mu J \leq J$ implies $J \geq J_\mu$. Take

$$\begin{aligned} \rho_2 &= \frac{1}{1-\alpha}, \\ \rho_1 &= \frac{\alpha\left[2\rho_2(2p-1)\right]}{1-\alpha}, \\ \rho_0 &= \max\left(\frac{\alpha p(\rho_2+\rho_1)}{1-\alpha}, \frac{m(1-p)+\alpha\left[\rho_2+\rho_1(2p-1)\right]}{1-\alpha}\right). \end{aligned}$$

For any state x such that $0 < x < N-1$, we can verify that

$$\begin{aligned} J(x) - (T_{\bar{\mu}}J)(x) &= \rho_0(1-\alpha) - m(1-p) - \alpha\left[\rho_2 + \rho_1(2p-1)\right] \\ &\geq \frac{m(1-p)+\alpha\left[\rho_2+\rho_1(2p-1)\right]}{1-\alpha}(1-\alpha) - m(1-p) - \alpha\left[\rho_2+\rho_1(2p-1)\right] \\ &= 0. \end{aligned}$$

For state $x = N-1$, note that if $N > 1 - \rho_1/2\rho_2$ we have $J(N) > J(N-1)$ and

$$\begin{aligned} J(N-1) - (T_{\bar{\mu}}J)(N-1) &= J(N-1) - (N-1)^2 - m(1-p) - \alpha\left[(1-p)J(N-2) + pJ(N-1)\right] \\ &\geq J(N-1) - (N-1)^2 - m(1-p) - \alpha\left[(1-p)J(N-2) + pJ(N)\right] \\ &= \rho_0(1-\alpha) - m(1-p) - \alpha\left[\rho_2 + \rho_1(2p-1)\right] \\ &\geq 0. \end{aligned}$$

Finally, for state $x = 0$ we have

$$\begin{aligned} J(0) - (T_{\bar{\mu}}J)(0) &= (1-\alpha)\rho_0 - \alpha p(\rho_2 + \rho_1) \\ &\geq (1-\alpha)\frac{\alpha p(\rho_2+\rho_1)}{1-\alpha} - \alpha p(\rho_2+\rho_1) \\ &= 0. \end{aligned}$$

It follows that $J \geq T_\mu J$, and for all $N > 1 - \rho_1/2\rho_2$,

$$0 \leq J^* \leq J_{\bar{\mu}} \leq J = \rho_2 x^2 + \rho_1 x + \rho_0.$$

A natural choice of Lyapunov function is, as in the previous example, $V(x) = x^2 + C$ for some $C > 0$. It follows that

$$\begin{aligned} \min_r \|J^* - \Phi r\|_{\infty,1/V} &\leq \|J^*\|_{\infty,1/V} \\ &\leq \max_{x\geq 0} \frac{\rho_2 x^2 + \rho_1 x + \rho_0}{x^2 + C} \\ &< \rho_2 + \frac{\rho_1}{2\sqrt{C}} + \frac{\rho_0}{C}. \end{aligned}$$

Now note that

$$\begin{aligned} \alpha(HV)(x) &\leq \alpha\left[p(x^2 + 2x + 1 + C) + (1-p)(x^2 + C)\right] \\ &= V(x)\left(\alpha + \frac{\alpha p(2x+1)}{x^2 + C}\right) \end{aligned}$$

and for C sufficiently large and independent of problem size, there is a $\beta < 1$ such that $\alpha HV \leq \beta V$ and $k_V \leq \frac{1}{1-\beta}$.

It remains to be shown that $c'V$ is finite. For that, we need to specify the state-relevance vector c. As in the case of the autonomous queue, we might want it to be close to the steady-state distribution of the states under the optimal policy. Clearly, it is not easy to choose state-relevant weights in that way since we do not know the optimal policy. Alternatively, we will use the general shape of the steady-state distribution to generate sensible state-relevance weights. To that end, let us analyze the infinite buffer case and show that, under some stability assumptions, there should be a geometric upper bound for the tail of steady-state distribution; we expect that results for finite (large) buffers should be similar if the system is stable, since in this case most of the steady-state distribution will be concentrated on relatively small states. Let us assume that the system under the optimal policy is indeed stable – that should generally be the case if the discount factor is large. For a queue with infinite buffer the optimal service rate $q(x)$ is nondecreasing in x [1], and stability therefore implies that

$$q(x) \geq q(x_0) > p$$

for all $x \geq x_0$ and some sufficiently large x_0. It is easy then to verify that the tail of the steady-state distribution has an upper bound with geometric decay since it should satisfy

$$\pi(x)p = \pi(x+1)q(x+1),$$

and therefore

$$\frac{\pi(x+1)}{\pi(x)} \leq \frac{p}{q(x_0)} < 1,$$

for all $x \geq x_0$. Thus a reasonable choice of state-relevance weights is $c(x) = \pi(0)\xi^x$, where $\pi(0) = \frac{1-\xi}{1-\xi^N}$ is a normalizing constant making c a probability distribution. In this case,

$$\begin{aligned} c'V &= \mathrm{E}\left[X^2 + C \mid X < N\right] \\ &\leq 2\frac{\xi^2}{(1-\xi)^2} + \frac{\xi}{1-\xi} + C, \end{aligned}$$

where X represents a geometric random variable with parameter $1-\xi$. We conclude that $c'V$ is uniformly bounded on N.

5.3 Example: A Queueing Network

Both previous examples were one-dimensional and showed that terms of interest are uniformly bounded over the number of states. We now consider a single reentrant line with d queues and finite buffers of size B to determine the impact of dimensionality on the terms involved in the error bound of Theorem 4.2.

We assume that exogenous arrivals occur at queue 1 in any time step with probability $p < 1/2$. The state $x \in \Re^d$ indicates the number of jobs in each queue. The cost per stage incurred at state x is given by

$$g(x) = \frac{|x|}{d} = \frac{1}{d}\sum_{i=1}^{d} x_i,$$

the average number of jobs per queue.

Let us first consider the cost-to-go function J^* and its dependency on the number of state variables d. Our goal is to establish bounds on J^* that will offer some guidance on the choice of a Lyapunov function V that keeps the error $\min_r \|J^* - \Phi r\|_{\infty,1/V}$ small. Since $J^* \geq 0$, we will only derive upper bounds. Instead of carrying the buffer size B throughout the calculations, we will consider the infinite buffer case. The cost-to-go of the finite buffer case should be bounded above by that of the infinite buffer case as having finite buffers corresponds to having jobs arriving at a full queue discarded at no additional cost.

As in [17], we have

$$\mathrm{E}_x\left[|x_t|\right] \leq |x| + pt,$$

since the expected total number of jobs at time t cannot exceed the total number of jobs at time 0 plus the expected number of arrivals between 0 and t, which is equal to pt. Therefore we have

$$\begin{aligned} \mathrm{E}_x\left[\sum_{t=0}^{\infty} \alpha^t |x_t|\right] &= \sum_{t=0}^{\infty} \alpha^t \mathrm{E}_x\left[|x_t|\right] \\ &\leq \sum_{t=0}^{\infty} \alpha^t (|x| + pt) \\ &= \frac{|x|}{1-\alpha} + \frac{p}{(1-\alpha)^2}. \end{aligned} \tag{10}$$

The first equality holds because $|x_t| \geq 0$ for all t; by the monotone convergence theorem, we can interchange the expectation and the summation. We conclude from (10) that the discounted cost-to-go in the infinite buffer case should be bounded above by a linear function of the state; in particular,

$$0 \leq J^*(x) \leq \frac{\rho_1}{d}|x| + \rho_0,$$

for some positive scalars ρ_0 and ρ_1 independent of the number of queues d. Note that since exogenous arrivals are restricted to queue 1, we can have a sharper bound with $\rho_0 = O(1/d)$. However, in the general case of arrivals occurring in many or all of the queues, we would still have ρ_0 independent of d.

As discussed before, the discounted cost-to-go in the infinite buffer case provides an upper bound for the cost-to-go in the case of finite buffers of size B. Therefore, the costs-to-go in the finite buffer case should be bounded above by the same linear function regardless of the value of B.

As in the previous examples, we will establish bounds on the terms involved in the error bound of Theorem 4.2. We consider a Lyapunov function $V(x) = \frac{1}{d}|x| + C$ for some constant $C > 0$, which implies

$$\begin{aligned}
\min_r \|J^* - \Phi r\|_{\infty,1/V} &\leq \|J^*\|_{\infty,1/V} \\
&\leq \max_{x\geq 0} \frac{\rho_1|x| + d\rho_0}{|x| + dC} \\
&\leq \rho_1 + \frac{\rho_0}{C},
\end{aligned}$$

and the above bound is independent of the number of queues in the system.

Now let us study k_V. We have

$$\begin{aligned}
\alpha(HV)(x) &\leq \alpha\left[p\left(\frac{1}{d}|x| + \frac{1}{d} + C\right) + (1-p)\left(\frac{1}{d}|x| + C\right)\right] \\
&\leq V(x)\left(\alpha + \alpha p \frac{\frac{1}{d}}{\frac{|x|}{d} + C}\right) \\
&\leq V(x)\left(\alpha + \frac{\alpha p}{dC}\right),
\end{aligned}$$

and it is clear that, for C sufficiently large and independent of d, there is a $\beta < 1$ independent of d such that $\alpha HV \leq \beta V$, and therefore $k_V \leq \frac{1}{1-\beta}$.

Finally, let us consider $c'V$. We expect that under some stability assumptions, the tail of the steady-state distribution will have an upper bound with geometric decay [3] and we take $c(x) = \left(\frac{1-\xi}{1-\xi^{B+1}}\right)^d \xi^{|x|}$. The state-relevance weights c are equivalent to the conditional joint distribution of d independent and identically distributed geometric random variables conditioned on the event that they are less than $B+1$. Therefore,

$$\begin{aligned}
c'V &= E\left[\frac{1}{d}\sum_{i=1}^{d} X_i + C \;\middle|\; X_i < B+1, i = 1, ..., d\right] \\
&< E[X_1] + C \\
&= \frac{\xi}{1-\xi} + C,
\end{aligned}$$

where $X_i, i = 1, ..., d$ are identically distributed geometric random variables with parameter $1-\xi$. It follows that $c'V$ is uniformly bounded over the number of queues.

6 Application to Controlled Queueing Networks

In this section, we discusss numerical experiments involving application of the linear programming approach to controlled queueing problems. In all examples, we assume that at most one

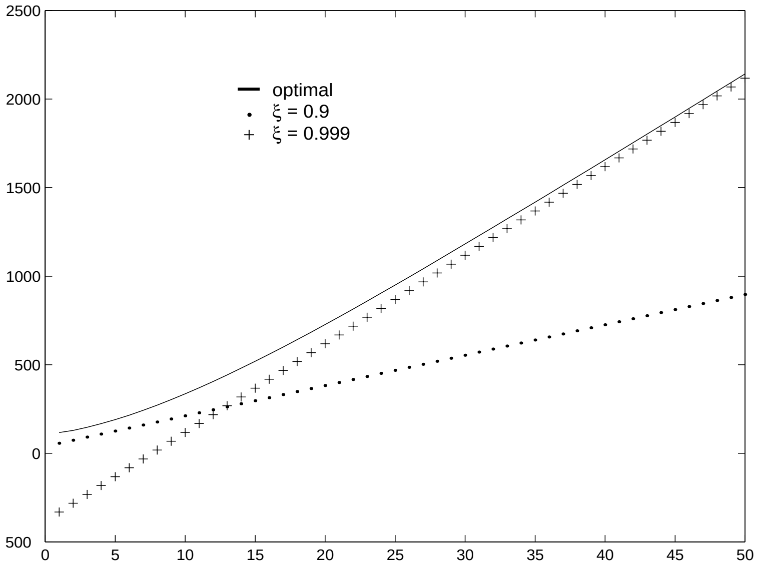


Figure 5: Approximate cost-to-go function for Example 6.1.

event (arrival/departure) occurs at each time step. The first example illustrates how state-relevance weights influence the solution of the approximate LP.

6.1 Single Queue with Controlled Service Rate

In Section 5.2, we studied a queue with a controlled service rate and determined that the bounds on the error of the approximate LP were uniform over the number of states. That example provided some guidance on the choice of basis functions; in particular, we now know that including a quadratic and a constant function guarantees that an appropriate Lyapunov function is in the span of the columns of Φ. Furthermore, our analysis of the (unknown) steady-state distribution revealed that state-relevance weights of the form $c(x) = (1-\xi)\xi^x$ are a sensible choice. However, how to choose an appropriate value of ξ was not discussed there. In this section, we present results of experiments with different values of ξ for a particular instance of the model described in Section 5.2. The values of ξ chosen for experimentation are motivated by ideas developed in Section 3.

We assume that jobs arrive at a queue with probability $p = 0.2$ in any unit of time. Service rates/probabilities $q(x)$ are chosen from the set $\{0.2, 0.4, 0.6, 0.8\}$. The cost incurred at any time for being in state x and taking action q is given by

$$g(x, q) = x + 60q^3.$$

We take the buffer size to be 49999 and the discount factor to be $\alpha = 0.98$. We select basis functions $\phi_1(x) = 1$, $\phi_2(x) = x$, $\phi_3(x) = x^2$, $\phi_4(x) = x^3$ and state-relevance weights $c(x) = (1-\xi)\xi^x$. The approximate LP is solved for $\xi = 0.9$ and $\xi = 0.999$ and we denote the solution of the approximate LP by r^{ξ}. The numerical results are presented in Figures 5, 6, 7 and 8.

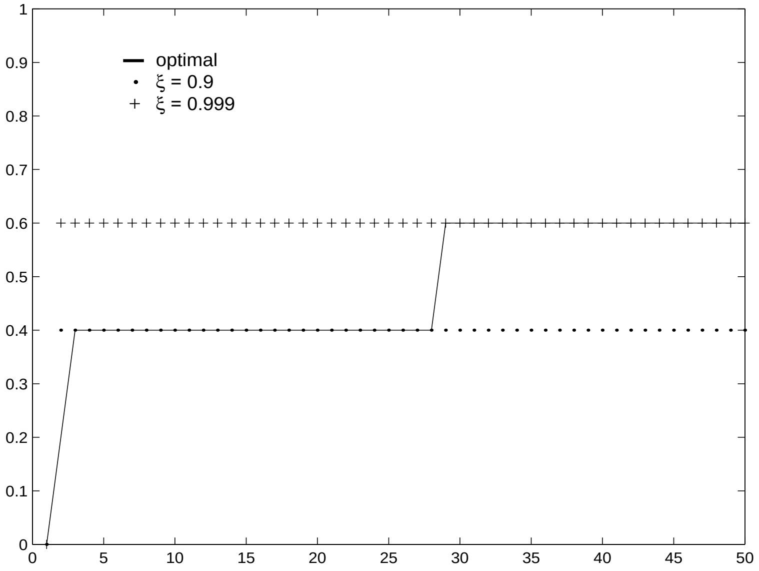


Figure 6: Greedy action for Example 6.1.

Figure 5 shows the approximations Φr^{ξ} to the cost-to-go function generated by the approximate LP. Note that the results agree with the analysis developed in Section 3; small states are approximated better when $\xi = 0.9$ whereas large states are approximated almost exactly when $\xi = 0.999$.

In Figure 6 we see the greedy action with respect to Φr^{ξ}. We get the right action for almost all "small" states with $\xi = 0.9$. On the other hand, $\xi = 0.999$ yields optimal actions for all relatively large states in the relevant range.

The most important result is illustrated in Figure 7, which depicts the cost-to-go functions associated with the greedy policies. Note that despite taking wrong actions for all relatively large states, the policy induced by $\xi = 0.9$ performs better than that generated with $\xi = 0.999$ in the range of relevant states, and it is close in value to the optimal policy even in those states for which it does not take the optimal action. Indeed, the average cost incurred by the greedy policy with respect to $\xi = 0.9$ is 2.92, relatively close to the average cost incurred by the optimal (discounted cost) policy, which is 2.72. The average cost incurred when $\xi = 0.999$ is 4.82, which is significantly higher.

Steady-state probabilities for each of the different greedy policies, as well as the corresponding (rescaled) state-relevance weights are shown in Figure 8. Note that setting ξ to 0.9 captures the relative frequencies of states, whereas setting ξ to 0.999 weights all states in the relevant range almost equally.

6.2 A Four-Dimensional Queueing Network

In this section we study the performance of the approximate LP algorithm when applied to a queueing network with two servers and four queues. The system is depicted in Figure 9 and it is the same as one studied in [5, 14, 19]. Arrival (λ) and departure ($\mu_i, i = 1, ..., 4$) probabilities

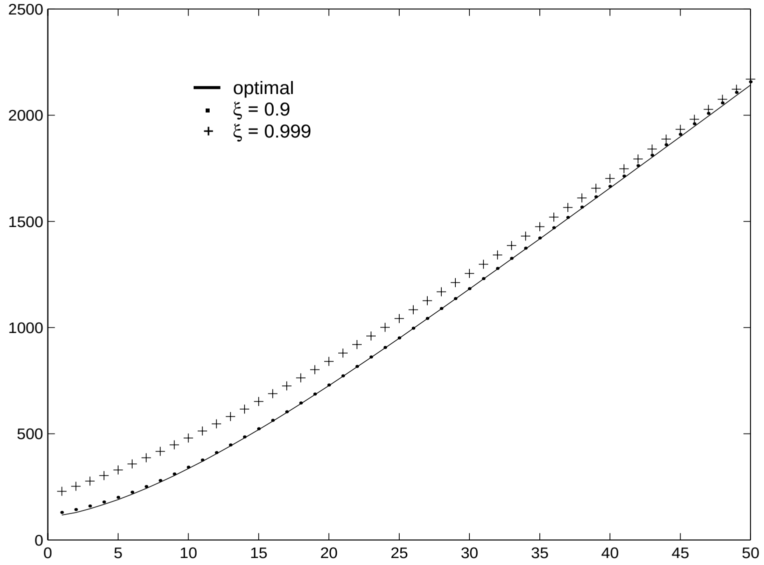


Figure 7: Cost-to-go function for Example 6.1.

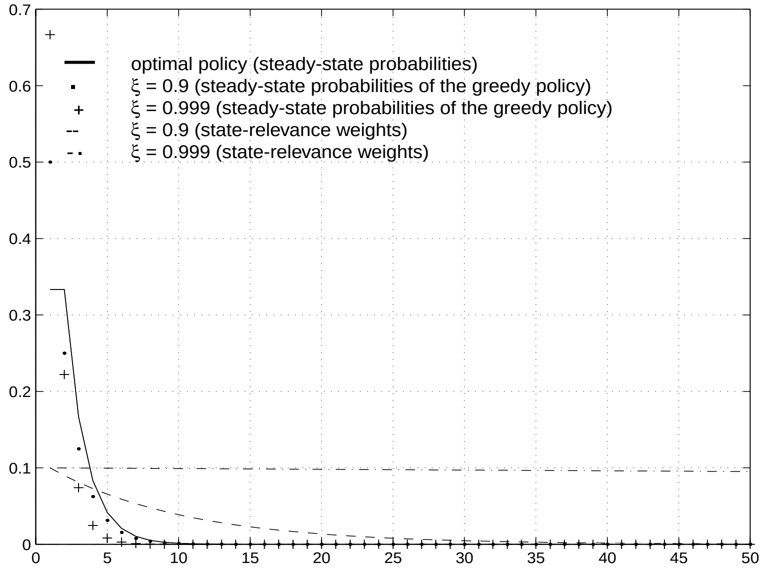


Figure 8: Steady-state probabilities for Example 6.1.

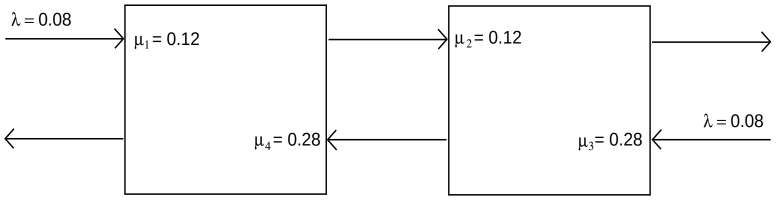


Figure 9: System for Example 6.2.

Policy	Average cost
$\xi = 0.95$	33.37
LONGEST	45.04
FIFO	45.71
LBFS	144.1

Table 1: Performance of different policies for Example 6.2. Average cost estimated by simulation after 50000000 iterations, starting with empty system.

are indicated. We assume a discount factor $\alpha = 0.99$. The state $x \in \Re^4$ indicates the number of jobs in each queue and the cost incurred in any period is $g(x) = |x|$, the total number of jobs in the system. Actions $a \in \{0,1\}^4$ satisfy $a_1 + a_4 \leq 1$, $a_2 + a_3 \leq 1$ and the non-idling assumption, i.e., a server must be working if any of its queues is nonempty. We have $a_i = 1$ iff queue i is being served.

Constraints for the approximate LP are generated by sampling 40000 states according to the distribution given by the state-relevance weights c. We choose the basis functions to span all of the polynomials in x of degree 3; therefore, there are

$$\begin{pmatrix} 4 \\ 0 \end{pmatrix} + \begin{pmatrix} 4 \\ 1 \end{pmatrix} + \left[\begin{pmatrix} 4 \\ 1 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix}\right] + \left[\begin{pmatrix} 4 \\ 1 \end{pmatrix} + 2\begin{pmatrix} 4 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 3 \end{pmatrix}\right] = 35$$

basis functions. The terms in the above expression denote the number of basis functions of degree 0, 1, 2, and 3, respectively.

We choose the state-relevance weights to be $c(x) = (1-\xi)^4 \xi^{|x|}$. Experiments were performed for a range of values of ξ. The best results were generated when $0.95 \leq \xi \leq 0.99$. The average cost was estimated by simulation with 50,000,000 iterations, starting with an empty system.

We compared the average cost obtained by the greedy policy with respect to the solution of the approximate LP with that of several different heuristics, namely, first-in-first-out (FIFO), last-buffer-first-served (LBFS), and a policy that always serves the longest queue (LONGEST). Results are summarized in Table 6.2 and we can see that the approximate LP yields significantly better performance than all of the other heuristics.

6.3 An Eight-Dimensional Queueing Network

In our last example, we consider a queueing network with eight queues. The system is depicted in Figure 10, with arrival ($\lambda_i, i = 1,2$) and departure ($\mu_i, i = 1, ..., 8$) probabilities indicated.

The state $x \in \Re^8$ represents the number of jobs in each queue. The cost-per-state is $g(x) = |x|$, and the discount factor α is 0.995. Actions $a \in \{0,1\}^8$ indicate which queues are being

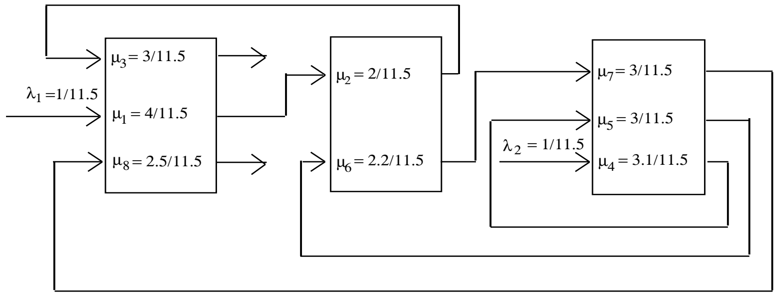


Figure 10: System for Example 6.3.

served; $a_i = 1$ iff a job from queue i is being processed. We consider only non-idling policies and, at each time step, a server processes jobs from one of its queues exclusively.

We choose state-relevance weights are of the form $c(x) = (1-\xi)^8 \xi^{|x|}$. The basis functions are chosen to span all polynomials in x of degree at most 2; therefore, the approximate LP has 47 variables. Due to the relatively large number of actions per state (up to 18), we choose to sample a relatively small number of states. Constraints for the approximate LP are generated by sampling 5000 states according to the distribution associated with the state-relevance weights c. Experiments were performed for $\xi = 0.85, 0.9$ and 0.95, and $\xi = 0.9$ yielded the policy with smallest average cost.

To evaluate the performance of the policy generated by the approximate LP, we compared it with first-in-first-out (FIFO), last-buffer-first-serve (LBFS) and a policy that serves the longest queue in each server (LONGEST). LBFS serves the job that is closest to leaving the system; for example, if there are jobs in queue 2 and in queue 6, a job from queue 2 is processed since it will leave the system after going through only one more queue, whereas the job from queue 6 will still have to go through two more queues. We also choose to assign higher priority to queue 8 than to queue 3 since queue 8 has higher departure probability.

We estimated the average cost of each policy with 50,000,000 simulation steps, starting with an empty system. Results appear in Figure 11. The policy generated by the approximate LP performs significantly better than each of the heuristics, yielding more than 10% improvement over LBFS, the second best policy. We expect that even better results could be obtained by refining the choice of basis functions and state-relevance weights.

The constraint generation step took 74.9 seconds and the resulting LP was solved in approximately 3.5 minutes of CPU time with CPLEX 7.0 running on a Sun Ultra Enterprise 5500 machine with Solaris 7 operating system and a 400 MHz processor.

7 Closing Remarks and Open Issues

In this paper we studied the linear programming approach to approximate dynamic programming for stochastic control problems as a means of alleviating the curse of dimensionality. We provided an error bound based on certain assumptions on the basis functions. The bounds were shown to be uniform in the number of states and state variables in certain queueing problems. Our analysis also led to some guidelines in the choice of the so-called "state-relevance weights" for the approximate LP.

An alternative to the approximate LP are temporal-difference learning (TD) methods [2, 6, 7, 21, 23, 24, 25]. In such methods, one tries to find a fixed point for an "approximate dynamic

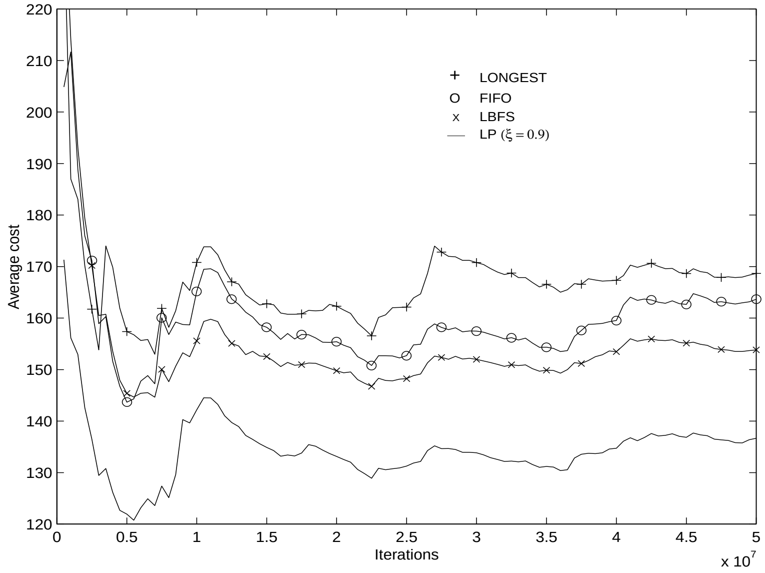


Figure 11: Average number of jobs in the system for Example 6.3.

programming operator" by simulating the system and learning from the observed costs and state transitions. Experimentation is necessary to determine when TD can offer better results than the approximate LP. However, it is worth mentioning that due to its complexity, much of TD's behavior is still to be understood; there are no convergence proofs or effective error bounds for general stochastic control problems. Such poor understanding leads to implementation difficulties; a fair amount of trial and error is necessary in order to get the method to perform well or even to converge. The approximate LP, on the other hand, benefits from the inherent simplicity of linear programming: its analysis is simpler, and error bounds such as those provided here provide guidelines on how to set the algorithm's parameters most efficiently. Packages for large-scale linear programming developed in the recent past also make the approximate LP relatively easy to implement.

In future work, as mentioned before, we will address the problem of constraint sampling. We expect that the number of constraints required for good performance will grow linearly in the number of basis functions employed, independently of the number of states or the number of state variables. It is also worth noting that although our bounds were developed under the assumption that all constraints are satisfied, this might not be necessary. Indeed, as pointed out in [11], when using the approximate LP method, one might actually benefit from allowing some of the constraints to be violated; in particular, note that if the constraints for a given state x are violated for a given vector J, that means $(TJ)(x) < J(x)$, in which case J is possibly assigning a high cost-to-go to state x. If we sample the constraints based on the relative "importance" of each state, it could be the case that the states corresponding to overestimated costs-to-go are states with actual high costs and the "error" introduced by constraint sampling could lead to better decisions as compared to the lower bound given by the approximate LP when all constraints are satisfied.

We have motivated many of the ideas and guidelines for choice of parameters through examples in queueing problems. In future work, we intend to explore how these ideas would be interpreted in other contexts, such as portfolio management and inventory control.

Several other questions remain open and are the object of future investigation: Can the state-relevance weights in the objective function be chosen in some adaptive way? Can we add robustness to the approximate LP algorithm to account for errors in the estimation of costs and transition probabilities, i.e., design an alternative LP with meaningful performance bounds when problem parameters are just known to be in a certain range? How do our results extend to the average cost case? How do our results extend to the infinite-state case? How does the quality of the approximate cost-to-go function, measure by the weighted L_1 norm, translate into actual performance of the associated greedy policy?

Acknowledgements

The authors thank John Tsitsiklis and Sean Meyn for useful discussions related to topics in the paper. This research was supported by NSF CAREER Grant ECS-9985229, by the ONR under Grant MURI N00014-00-1-0637, and by an IBM Research Fellowship.

References

[1] Bertsekas, D., *Dynamic Programming and Optimal Control,* Athena Scientific, 1995.

[2] Bertsekas, D. & Tsitsiklis, J.N., *Neuro-Dynamic Programming,* Athena Scientific, 1996.

[3] Bertsimas, D., Gamarnik, D. & Tsitsiklis, J., "Performance of Multiclass Markovian Queueing Networks via Piecewise Linear Lyapunov Functions," submitted to *Annals of Applied Probability,* 2000.

[4] Borkar, V., "A Convex Analytic Approach to Markov Decision Processes," *Probability Theory and Related Fields 78,* pp. 583-602, 1988.

[5] Chen, R-R & Meyn, S., "Value Iteration and Optimization of Multiclass Queueing Networks," *Queueing Systems 32,* pp. 65-97, 1999.

[6] Dayan, P., "The Convergence of TD(λ) for General λ," *Machine Learning 8*, pp. 341-362, 1992.

[7] de Farias, D.P. & Van Roy, B.,"On the Existence of Fixed Points for Appproximate Value Iteration and Temporal-Difference Learning," *Journal of Optimization Theory and Applications 105,* No. 3, June, 2000.

[8] de Ghellinck, G., "Les Problèmes de Décisions Séquentielles," *Cahiers du Centre d'Etudes de Recherche Opérationnelle 2*, pp.161-179, 1960.

[9] Denardo, E.V., "On Linear Programming in a Markov Decision Problem," *Management Science 16*, Vol. 5, pp. 282-288, 1970.

[10] D'Epenoux, F., "A Probabilistic Production and Inventory Problem," *Management Science 10*, Vol. 1, pp.98-108, 1963.

[11] Gordon, G., *Approximate Solutions to Markov Decision Processess,* Ph.D. Thesis, Carnegie Mellon University, 1999.

[12] Grötschel, M. & Holland, O., " Solution of Large-Scale Symmetric Travelling Salesman Problems," *Mathematical Programming 51*, 141-202, 1991.

[13] Hordijk, A., & Kallenberg, L.C.M., "Linear Programming and Markov Decision Chains," *Management Science 25*, pp. 352-362, 1979.

[14] Kumar, P.R. & Seidman, T.I., "Dynamic Instabilities and Stabilization Methods in Distributed Real-Time Scheduling of Manufacturing Systems," *IEEE Transactions on Automatic Control 35*, No. 3, pp. 289-298, 1990.

[15] Kumar, P.R. & Meyn, S., "Duality and Linear Programs for Stability and Performance Analysis of Queueing Networks and Scheduling Policies," *IEEE Transactions on Automatic Control 41*, No. 1, pp. 4-17, January 1996.

[16] Manne, A.S., "Linear Programming and Sequential Decisions," *Management Science 6*, No. 3, pp. 259-267, 1960.

[17] Meyn, S.P., "Sequencing and Routing in Multiclass Queueing Networks, Part I: Feedback Regulation," to appear, *SIAM Journal of Control and Optimization*, 2001.

[18] I. Ch. Paschalidis and J. N. Tsitsiklis, "Congestion-Dependent Pricing of Network Services," *IEEE/ACM Transactions on Networking*, Vol. 8, No. 2, pp. 171-184, 2000.

[19] Rybko, A.N. & Stolyar, A.L., "On the Ergodicity of Stochastic Processes Describing the Operation of Open Queueing Networks," *Problemy Peredachi Informatsii 28*, pp. 3-26, 1992.

[20] Schweitzer, P. & Seidmann, A., "Generalized Polynomial Approximations in Markovian Decision Processes," *Journal of Mathematical Analysis and Applications 110*, pp. 568-582, 1985.

[21] Sutton, R.S., "Learning to Predict by the Methods of Temporal Differences," *Machine Learning 3*, pp 9-44, 1988.

[22] Trick, M. & Zin, S., "A Linear Programming Approach to Solving Dynamic Programs," unpublished manuscript, 1993.

[23] Tsitsiklis, J.N. & Van Roy, B., "An Analysis of Temporal-Difference Learning with Function Approximation," *IEEE Transactions on Automatic Control 42*, No. 5, pp. 674-690, 1997.

[24] Van Roy, B., "Neuro-Dynamic Programming: Overview and Recent Trends," in *Markov Decision Processes: Models, Methods, Directions, and Open Problems,* edited by E. Feinberg and A. Schwartz, Kluwer, 2000.

[25] Van Roy, B., *Learning and Value Function Approximation in Complex Decision Processes,* Ph.D. Thesis, Massachusetts Institute of Technology, May 1998.

Robust Reinforcement Learning Control with Static and Dynamic Stability

Chuck Anderson, with R. M. Kretchmar, P. M. Young, D. C. Hittle
(International Journal of Robust and Nonlinear Control, vol. 11, 2001)
(anderson@cs.colostate.edu)

Abstract

Robust control theory is used to design stable controllers in the presence of uncertainties. This provides powerful closed-loop robustness guarantees, but can result in controllers that are conservative with regard to performance. Here we present an approach to learning a better controller through observing actual controlled behavior. A neural network is placed in parallel with the robust controller and is trained through reinforcement learning to optimize performance over time. By analyzing nonlinear and time-varying aspects of a neural network via uncertainty models, a robust reinforcement learning procedure results that is guaranteed to remain stable even as the neural network is being trained. The behavior of this procedure is demonstrated and analyzed on two control tasks. Results show that at intermediate stages the system without robust constraints goes through a period of unstable behavior that is avoided when the robust constraints are included. (Partially supported by the National Science Foundation through grants CMS-9804757 and CMS-9732986.)

Approximating a Policy Can be Easier Than Approximating a Value Function

Chuck Anderson
Technical Report CS-00-101, Colorado State University

Abstract

Value functions can speed the learning of a solution to Markov Decision Problems by providing a prediction of reinforcement against which received reinforcement is compared. Once the learned values relatively reflect the optimal ordering of actions, further learning is not necessary. In fact, further learning can lead to the disruption of the optimal policy if the value function is implemented with a function approximator of limited complexity. This is illustrated here by comparing Q-learning and a policy-only algorithm (Baxter and Bartlett, 1999), both using a simple neural network as the function approximator. A Markov Decision Problem is shown for which Q-learning oscillates between the optimal policy and a sub-optimal one, while the direct-policy algorithm converges on the optimal policy.

Robust Solutions to Markov Decision Problems

Laurent Elghaoui, Arnab Nilim
(elghaoui@eecs.berkeley.edu, nilim@eecs.berkeley.edu)

Abstract

Optimal solutions to finite-state Markov Decision Problems (MDPs) are often sensitive with respect to the state transition probabilities. In many practical problems, the estimates of the transition probabilities are not accurate. Hence, estimation errors are, together with the curse of dimensionality, a limiting factor in applying MDPs to real-life problems. We propose an algorithm for solving MDPs such that the solution is guaranteed to be robust with respect to the estimation errors of the state transition probabilities. Our algorithm is based on a robust version to exact linear programming formulations of MDPs, involving a statistically accurate yet numerically efficient representation of uncertainty via lower bounds on likelihood functions. Our robust MDP approach involves very moderate additional computational cost accrued by the new robustness requirements, while offering a non-conservative description of uncertainty.

Algebraic and Adaptive Learning for Heuristic Neural Control

NSF Workshop on Learning and Approximate Dynamic Programming
Playacar, Mexico, April 8-10, 2002

Silvia Ferrari and Robert F. Stengel
(sferrari@phoenix.princeton.edu)

Summary

A nonlinear control system comprising a network of networks is taught using a two-phase learning procedure realized through novel techniques for initialization, on-line training, and adaptive critic design. A critical observation is that the gradients of the networks must equal corresponding linear gain matrices at chosen operating points. On-line learning is based on a dual heuristic adaptive critic architecture that improves control for large, coupled motions by accounting for actual plant dynamics and nonlinear effects. An action network computes the optimal control law; a critic network predicts the derivative of the cost-to-go with respect to the state. Both networks are algebraically initialized based on a-priori linear control knowledge and continue to adapt on line during full scale simulations of the plant. On-line training takes place sequentially over discrete periods of time and involves several numerical procedures. A backpropagating algorithm called Resilient Backpropagation is modified and successfully implemented to meet these objectives, without excessive computational expense. This adaptive controller is as conservative as the linear designs and as effective as the global nonlinear controller. The method is successfully implemented for the full-envelope control of a six-degree-of-freedom aircraft simulation. The results show that the on-line adaptation brings about improved performance with respect to the initialization phase during large-angle, coupled aircraft maneuvers.

Bibliography:

Ferrari, S., Stengel, R. F. "An Adaptive Critic Global Controller," to be presented at the American Control Conference, Anchorage, AK, May 2002.

Ferrari, S., Stengel, R. F. "Classical/Neural Synthesis of Nonlinear Control Systems," to appear in the Journal of Guidance, Control, and Dynamics.

Ferrari, S., Stengel, R. F., "Algebraic Training of a Neural Network," Proceedings of the American Control Conference, Arlington, VA, June 2001.

Ferrari, S., Stengel, R. F., "Classical/Neural Synthesis of Nonlinear Control Systems," AIAA-2000-4552, Proceedings of the AIAA Guidance, Navigation, and Control Conference, Denver, CO, August 2000.

An Adaptive Critic Global Controller

Silvia Ferrari* and Robert F. Stengel†

Princeton University
Department of Mechanical and Aerospace Engineering
Princeton, NJ 08544

Abstract

A nonlinear control system comprising a network of networks is taught using a two-phase learning procedure realized through novel techniques for initialization, on-line training, and adaptive critic design. The neural networks are initialized algebraically by observing that the gradients of the networks must equal corresponding linear gain matrices at chosen operating points. On-line learning is based on a dual heuristic adaptive critic architecture that improves control for large, coupled motions by accounting for plant dynamics and nonlinear effects. The result is an adaptive controller that is as conservative as the linear designs and as effective as the global controller. The design method is implemented to control the full six-degree-of-freedom simulation of a business jet aircraft.

1. Introduction

The problem of optimizing a desired metric over time lies at the basis of virtually all robust and fault-tolerant control and identification schemes. Dynamic programming uses the principle of optimality to find an optimal strategy of action in a nonlinear environment. *Backwards* or *discrete* dynamic programming methods discretize the state space and make a direct comparison of the cost associated with all feasible trajectories, guaranteeing solution of the optimal control problem [1]. This approach leads to a number of computations that grows exponentially with the number of state variables ("curse of dimensionality") [2]. Adaptive critic designs constitute a class of *approximate* dynamic programming methods [3] that uses incremental optimization combined with a parametric structure to efficiently approximate the optimal cost and control. They optimize a short-term cost metric that ensures optimization of the cost over all future times. Neural networks are the parametric structures of choice, because they easily handle large-dimensional input and output spaces and can learn in an incremental fashion.

The simplest adaptive critic architectures are based on heuristic dynamic programming (HDP). They implement a critic network to approximate the cost-to-go in the Bellman equation [2] and an action network to approximate the optimal control law. This paper presents a design approach based on a modification of HDP, referred to as dual heuristic programming (DHP), where the critic network approximates the derivatives of the cost-to-go with respect to the state. DHP is more promising than its earlier counterpart because it learns more quickly and alleviates persistence of excitation problems by computing the correlation between the cost and the individual state elements [4].

The advantages brought about by using prior knowledge in conjunction with on-line training are widely recognized in the neurocontrol literature [5]. In the present approach, the nonlinear control system, comprising a network of networks, is taught using a two-phase learning procedure. During the first phase, referred to as *initialization*, the network size and parameters are determined from well-established linear control theory solely by solving algebraic equations that identify the exact matching of gain matrices at chosen operating points. During a second phase, on-line learning by a DHP approach improves control response for large, coupled motions, based on the actual state of the plant. This on-line phase accounts for differences between actual and assumed dynamic models and for nonlinear effects not captured by the linear designs. Classical control theory provides a unifying framework for the two training phases. The algebraic initialization is based on the linear quadratic regulator; the DHP approach is based on approximate dynamic programming.

2. Foundations

The goal of the adaptive critic design is to approximate the optimal control law for an infinite horizon problem subject to the real-time dynamics of a continuous plant or simulation. The neural controller adapts on line, with the plant operating over the entire range of state and command-input elements, $\{\mathbf{x}(\mathbf{y}_c), \mathbf{y}_c\}$, or some suitably dense set in the space denoted by *OR*. The plant state, $\mathbf{x}$, and the command input, $\mathbf{y}_c$, are fed to the controller on-line and are unknown prior to operation. It is assumed that linearized time-invariant plant models are known *a priori* for a subset of operating points, $OP \subset OR$. Corresponding linear control data are used to initialize the action and critic neural networks. These networks are further adjusted over time through the DHP architecture sketched in Fig. 1.

[1] Graduate Student.

† Professor. Fellow IEEE, AIAA.

Presented at the 2002 American Control Conference, Anchorage, AK, May 2002.

2.1. Problem Statement

Consider the deterministic minimization of a scalar integral function of the $n \times 1$ plant state, $\mathbf{x}$, and of the $m \times 1$ control, $\mathbf{u}$, and a scalar terminal cost:

$$J = \phi[\mathbf{x}(t_f)] + \int_{t_0}^{t_f} L[\mathbf{x}(\tau), \mathbf{u}(\tau)]\, d\tau \qquad (1)$$

The objective is to determine the control law for which this cost function is stationary, subject to the dynamic equation:

$$\dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t)], \ \ \mathbf{x}(t_0) \text{ given} \qquad (2)$$

Plant motions and controls are sensed in the $e_s \times 1$ output vector $\mathbf{y}_s$,

$$\mathbf{y}_s(t) = \mathbf{h}_s[\mathbf{x}(t), \mathbf{u}(t)] \qquad (3)$$

It is assumed that perfect measurements are available and that the output views all elements of the state. The mission goals are expressed by the $e_c \times 1$ command input, $\mathbf{y}_c$, which can be viewed as some desirable combination of state and control elements with $e_c \le m$.

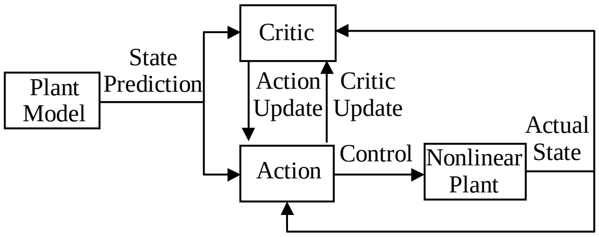


Figure 1. Dual heuristic programming adaptive critic.

The *action network* models the control law, which is assumed to be a function of the state. It can be written as the sum of a nominal and a perturbed effect,

$$\mathbf{u}^*[\mathbf{x}^*(t)] = \mathbf{u}_0^*[\mathbf{x}_0^*(t)] + \Delta\mathbf{u}^*[\mathbf{x}_0^*(t), \Delta\mathbf{x}^*(t)] \qquad (4)$$

where, $\mathbf{x}^*(t) = \mathbf{x}_0^*(t) + \Delta\mathbf{x}^*(t)$, and $(\bullet)^*$ denotes the optimal solution. When the control law depends on parameters and command inputs as well as the state [6], an augmented state can be defined to include these additional elements, as described in later sections. At any moment in time, $t_0 \le t \le t_f$, the minimized value function or cost-to-go, $V^*(t)$, corresponding to eq. (1) can be expressed as:

$$V^*[\mathbf{x}^*(t)] = \min_{\mathbf{u}(t)} \left\{ \phi[\mathbf{x}^*(t_f)] - \int_{t_f}^{t} L[\mathbf{x}^*(\tau), \mathbf{u}(\tau)]\, d\tau \right\} \qquad (5)$$

The *critic network* evaluates the action network performance by approximating the following derivative of the corresponding cost-to-go with respect to the state:

$$\boldsymbol{\lambda}^*[\mathbf{x}^*(t)] \equiv \frac{\partial V^*[\mathbf{x}^*(t)]}{\partial \mathbf{x}^*(t)} \qquad (6)$$

Single-hidden-layer sigmoidal neural networks of the type shown in Fig. 2 are chosen to model the action and critic functionals. They have input $\mathbf{p}(t) = [\mathbf{x}(t)^T\ \mathbf{a}(t)^T]^T$, where $\mathbf{a}$ is a *scheduling vector* of auxiliary inputs that informs the neural networks of the dynamically significant variables in the system. The network adjustable parameters consist of the input weights, $\mathbf{W}$, of the output weights, $\mathbf{V}$, and of the input and output biases, $\mathbf{d}$ and $\mathbf{b}$. The output of the network is computed as the nonlinear transformation of the weighted sum of the input and the input bias:

$$\mathbf{z}[\mathbf{p}(t)] = \mathbf{V}^T \boldsymbol{\sigma}[\mathbf{W}\mathbf{p}(t) + \mathbf{d}] + \mathbf{b} \qquad (7)$$

$\boldsymbol{\sigma}[\bullet]$ is a vector-valued function composed of individual sigmoidal functions of the form $\sigma(n) \equiv (e^n - 1)/(e^n + 1)$. This architecture can approximate any nonlinear function on a compact space arbitrarily well [7].

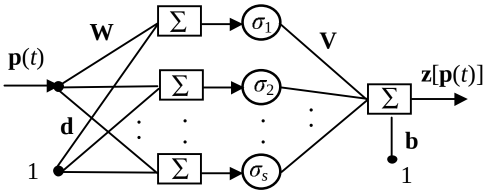


Figure 2. Sample vector-input vector-output sigmoidal network with *s* nodes in the hidden layer.

2.2. Initialization Phase

The goal of the initialization phase is to incorporate linear control knowledge in the nonlinear control system. The procedure is based on the observation that the network gradients must equal corresponding linear control matrices at selected operating points, *OP*, indexed by $\kappa = 1, 2, \ldots, p$. Linearized models of the plant can be obtained from eq. 2 for the subset *OP* by assuming small perturbations about corresponding equilibria, and ignoring time-varying effects:

$$\Delta\dot{\mathbf{x}}(t) = \mathbf{F}\Delta\mathbf{x}(t) + \mathbf{G}\Delta\mathbf{u}(t), \ \ \Delta\mathbf{x}(t_0) \text{ given} \qquad (8)$$

The optimization goals are expressed as a quadratic function of the state and control

$$J = \frac{1}{2}\int_0^{t_f} \left[\Delta\mathbf{x}^T(\tau)\mathbf{Q}\Delta\mathbf{x}(\tau) + 2\Delta\mathbf{x}^T(\tau)\mathbf{M}\Delta\mathbf{u}(\tau) + \Delta\mathbf{u}^T(\tau)\mathbf{R}\Delta\mathbf{u}(\tau)\right] d\tau \qquad (9)$$

When the plant is subject to continuing disturbance inputs and t_f becomes infinite in the limit, the value of J may still be bounded by defining an average cost,

$$J_A = \lim_{t_f \to \infty} \frac{J}{t_f} \qquad (10)$$

that has the same optimality conditions as J [6]. As t_f approaches infinity, it is reasonable to let the terminal cost, $\phi[\mathbf{x}(t_f)]$, equal zero. Furthermore, it can be shown [8] that the value function,

$$V^*[\Delta\mathbf{x}^*(t)] = \frac{1}{2}\Delta\mathbf{x}^{*T}(t)\mathbf{P}(t)\Delta\mathbf{x}^*(t) \qquad (11)$$

is optimal for eq. 8 and 9, and that $\mathbf{P}(t)$ approaches its steady-state value $\mathbf{P}$. The following closed-form linear-optimal control law can be derived [6]:

$$\Delta\mathbf{u}^*(t) = -\mathbf{R}^{-1}\left[\mathbf{G}^{\mathrm{T}}\mathbf{P} + \mathbf{M}^{\mathrm{T}}\right]\Delta\mathbf{x}^*(t) = -\mathbf{C}\Delta\mathbf{x}^*(t) \qquad (12)$$

LTI control laws that satisfy desired engineering criteria [9] can be designed for *OP* to provide a set of locally optimal gains and Riccati matrices $\{\mathbf{C}, \mathbf{P}\}_\kappa$. The gradient of the action network at the κ^{th} operating point, which has value in initializing the network, is found by differentiating eq. 4 with respect to $\mathbf{x}^*(t)$. Using the result in eq. 12:

$$\left.\frac{\partial\mathbf{u}^*\left[\mathbf{x}^*(t)\right]}{\partial\mathbf{x}^*(t)}\right|_{\mathbf{x}_0^*,\,\mathbf{a}_\kappa} = \left.\frac{\partial\Delta\mathbf{u}^*(t)}{\partial\Delta\mathbf{x}^*(t)}\right|_{\Delta\mathbf{x}^*=\mathbf{0},\,\mathbf{a}_\kappa} = -\mathbf{C}_\kappa \qquad (13)$$

$\mathbf{C}_\kappa$ is known from the LQ optimal gain matrices, and $\mathbf{a}_\kappa$ is the scheduling vector evaluated at the κ^{th} operating conditions. In infinite horizon problems, the structure of the value function is independent of time; therefore, a single time-invariant critic network can be used to approximate $\lambda^*[\mathbf{x}^*(t)]$ or simply $\lambda^*(t)$ (eq. 6). The LQ optimal value function, eq. 11, can be differentiated twice with respect to the state to seek the following derivative,

$$\left.\frac{\partial\boldsymbol{\lambda}^*\left[\mathbf{x}^*(t)\right]}{\partial\mathbf{x}^*(t)}\right|_{\mathbf{x}_0^*,\,\mathbf{a}_\kappa} = \left.\frac{\partial^2 V^*\left[\Delta\mathbf{x}^*(t)\right]}{\partial\Delta\mathbf{x}^*(t)^2}\right|_{\Delta\mathbf{x}^*=\mathbf{0},\,\mathbf{a}_\kappa} = \mathbf{P}_\kappa \qquad (14)$$

where, $\mathbf{P}_\kappa$ is known and is used to initialize the critic.

Thus, under the stated assumptions, the network gradient $\partial\mathbf{z}[\mathbf{p}(t)]/\partial\mathbf{x}(t)$ is known for both the critic and the action network. In addition, the following condition applies to their input/output relationship:

$$\mathbf{z}\left[\mathbf{x}(t), \mathbf{a}(t)\right]\Big|_{\mathbf{x}_0^*,\,\mathbf{a}_\kappa} = \mathbf{0} \qquad (15)$$

The network architecture, number of nodes, and parameters that match these requirements exactly are determined in one step by solving sets of linear algebraic equations.

2.3. Dual Heuristic Programming Adaptive Critic

The on-line logic is implemented in discrete time through an incremental optimization scheme based on dual heuristic dynamic programming. During each time interval $\Delta t = t_{k+1} - t_k$, the action and critic networks are adapted to more closely approximate the optimal control law and value function derivatives, respectively. Adaptation criteria are derived from the recurrence relation by discretizing the optimal control problem [1]. Howard's form of the recurrence relation [3] can be used to approximate the value function over time,

$$V\left[\mathbf{x}(t_k)\right] = L\left[\mathbf{x}(t_k), \mathbf{u}(t_k)\right] + V\left[\mathbf{x}(t_{k+1})\right] \qquad (16)$$

where $V[\mathbf{x}(t_{k+1})]$ is necessarily a predicted value. The control $\mathbf{u}(t_k)$ is defined as the function of $\mathbf{x}(t_k)$ that minimizes the right-hand side of eq. 16. When the function $V[\mathbf{x}(t_k)]$ is calculated from eq. 16 based on the current control, and $\mathbf{u}(t_k)$ is adjusted to minimize this optimal value function approximation, the method iteratively converges to an optimal strategy [3]. For simplicity, the asterisks will be omitted in the remainder of the paper.

At time t_k, the control strategy for which the value function is stationary satisfies the optimality condition:

$$\frac{\partial V\left[\mathbf{x}(t_k)\right]}{\partial\mathbf{u}(t_k)} = \frac{\partial L\left[\mathbf{x}(t_k), \mathbf{u}(t_k)\right]}{\partial\mathbf{u}(t_k)} + \boldsymbol{\lambda}\left[\mathbf{x}(t_{k+1})\right]\frac{\partial\mathbf{x}(t_{k+1})}{\partial\mathbf{u}(t_k)} = 0 \qquad (17)$$

Equation 16 is differentiated with respect to the state to obtain a recurrence relation for the DHP critic, which approximates the functional $\lambda[\mathbf{x}(t)]$:

$$\begin{aligned}\boldsymbol{\lambda}\left[\mathbf{x}(t_k)\right] \equiv \frac{\partial V\left[\mathbf{x}(t_k)\right]}{\partial\mathbf{x}(t_k)} = \frac{\partial L\left[\mathbf{x}(t_k), \mathbf{u}(t_k)\right]}{\partial\mathbf{x}(t_k)} + \\ \frac{\partial L\left[\mathbf{x}(t_k), \mathbf{u}(t_k)\right]}{\partial\mathbf{u}(t_k)}\frac{\partial\mathbf{u}\left[\mathbf{x}(t_k)\right]}{\partial\mathbf{x}(t_k)} + \boldsymbol{\lambda}\left[\mathbf{x}(t_{k+1})\right]\frac{\partial\mathbf{x}(t_{k+1})}{\partial\mathbf{x}(t_k)} + \qquad (18) \\ \boldsymbol{\lambda}\left[\mathbf{x}(t_{k+1})\right]\frac{\partial\mathbf{x}(t_{k+1})}{\partial\mathbf{u}(t_k)}\frac{\partial\mathbf{u}\left[\mathbf{x}(t_k)\right]}{\partial\mathbf{x}(t_k)}\end{aligned}$$

The critic is then used to compute $\lambda[\mathbf{x}(t_{k+1})]$ in eq. 17, once the prediction of the state, $\mathbf{x}(t_{k+1})$, is known from the model of the plant (eq. 2).

3. On-line Phase Implementation

The DHP criteria are implemented for the networks' adaptation, based on $\mathbf{x}(t_k)$, as shown by the schematic in Figs. 3 and 4. The adjustable parameters (or weights) of each network are updated to minimize the mean-squared error between a desired output or target and the network's actual output, $\mathbf{z}[\mathbf{p}(t_k)]$, for the input $\mathbf{p}(t_k)$. Equations 17 and 18 are used to generate the action and the critic desired outputs corresponding to $\mathbf{p}(t_k)$, $\mathbf{u}_D(t_k)$ and $\lambda_D(t_k)$, respectively. During the first time interval $(t_1 - t_0)$, the initialization weights are used prior to the network update. Later, the weights obtained during $(t_k - t_{k-1})$ are used as prior weights for the interval $(t_{k+1} - t_k)$.

3.1. Action and Critic Network Target Generation

The action network target, $\mathbf{u}_D(t_k)$, is obtained by solving the optimality condition, eq. 17, which consists of a set of nonlinear equations. A guess to the solution, $\mathbf{u}_D(t_k)^G$, initially is provided by the action network (using the prior weights). Subsequently, it is perturbed by an established algorithm (e.g., Newton-Raphson) until the stopping condition is met. $\lambda(t_{k+1})$ is computed by the critic network based on the prediction of $\mathbf{x}(t_{k+1})$, as shown in Fig. 3. Once the action network has been updated, the critic's desired output is computed from eq. 18 based on the exact values of $\mathbf{u}(t_k)$ and $\partial\mathbf{u}(t_k)/\partial\mathbf{x}(t_k)$. In the case of the critic (Fig. 4), no iteration is needed to compute its target $\lambda_D(t_k)$, which is at best a prediction of $\lambda(t_k)$. The derivatives $\partial L[\bullet]/\partial\mathbf{x}(t_k)$ and $\partial L[\bullet]/\partial\mathbf{u}(t_k)$ are computed analytically from $L[\mathbf{x}(t_k),\mathbf{u}(t_k)]$. The transition matrices, $\partial\mathbf{x}(t_{k+1})/\partial\mathbf{u}(t_k)$ and $\partial\mathbf{x}(t_{k+1})/\partial\mathbf{x}(t_k)$, are obtained numerically from eq. 2.

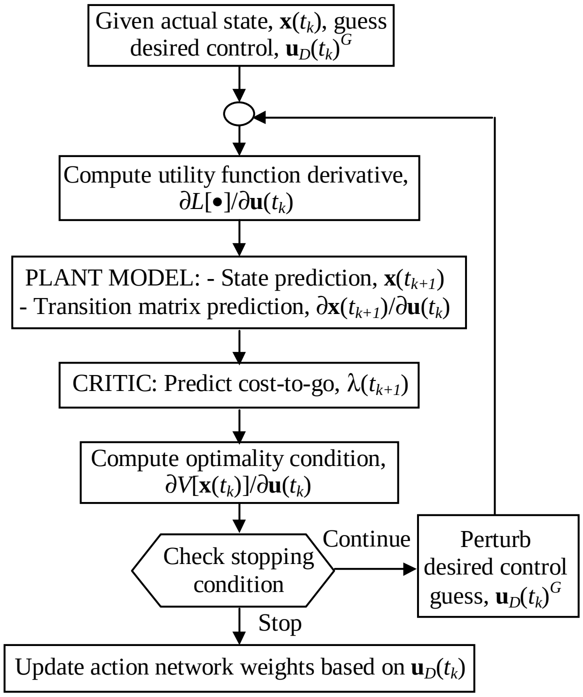

Given actual state, $\mathbf{x}(t_k)$, guess desired control, $\mathbf{u}_D(t_k)^G$
Compute utility function derivative, $\partial L[\bullet]/\partial \mathbf{u}(t_k)$
PLANT MODEL: - State prediction, $\mathbf{x}(t_{k+1})$
- Transition matrix prediction, $\partial \mathbf{x}(t_{k+1})/\partial \mathbf{u}(t_k)$
CRITIC: Predict cost-to-go, $\lambda(t_{k+1})$
Compute optimality condition, $\partial V[\mathbf{x}(t_k)]/\partial \mathbf{u}(t_k)$
Check stopping condition
Continue
Perturb desired control guess, $\mathbf{u}_D(t_k)^G$
Stop
Update action network weights based on $\mathbf{u}_D(t_k)$

Figure 3. Action network adaptation, during $\Delta t = t_{k+1} - t_k$.

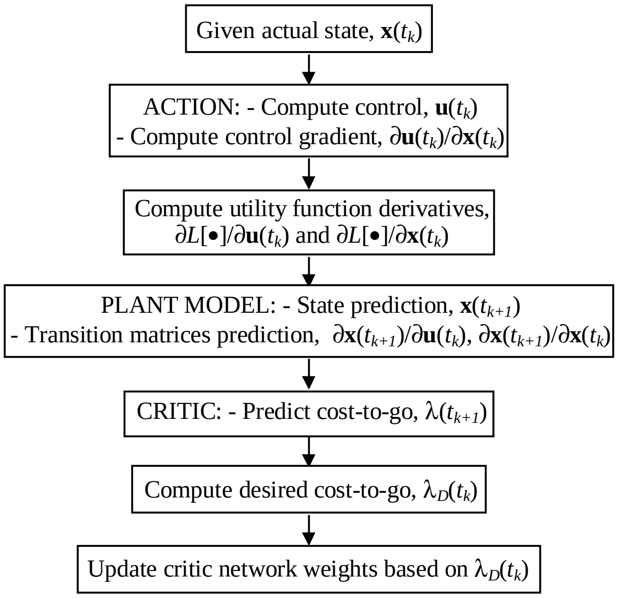

Given actual state, $\mathbf{x}(t_k)$
ACTION: - Compute control, $\mathbf{u}(t_k)$
- Compute control gradient, $\partial \mathbf{u}(t_k)/\partial \mathbf{x}(t_k)$
Compute utility function derivatives, $\partial L[\bullet]/\partial \mathbf{u}(t_k)$ and $\partial L[\bullet]/\partial \mathbf{x}(t_k)$
PLANT MODEL: - State prediction, $\mathbf{x}(t_{k+1})$
- Transition matrices prediction, $\partial \mathbf{x}(t_{k+1})/\partial \mathbf{u}(t_k)$, $\partial \mathbf{x}(t_{k+1})/\partial \mathbf{x}(t_k)$
CRITIC: - Predict cost-to-go, $\lambda(t_{k+1})$
Compute desired cost-to-go, $\lambda_D(t_k)$
Update critic network weights based on $\lambda_D(t_k)$

Figure 4. Critic network adaptation, during $\Delta t = t_{k+1} - t_k$.

3.2. On-line Training Algorithm

The on-line training algorithm minimizes an error function E, defined in terms of one desired output $\mathbf{z}_D$ and the actual network output $\mathbf{z}$, with respect to $\mathbf{w}$, a vector of ordered weights w_ℓ indexed by $\ell = 1, 2, \ldots$:

$$E(\mathbf{w}) \equiv \frac{1}{2}\left\|\mathbf{z}_D - \mathbf{z}(\mathbf{w})\right\|^2 \tag{19}$$

Since the initialized weights are close to being optimal, the on-line minimization is kept local. Based on the idea of backpropagation learning [11], at each epoch, i, the on-line training algorithm modifies each weight $w_\ell^{(i)}$ by an increment $\Delta w_\ell^{(i)}$, based on the derivative $\partial E(\mathbf{w})/\partial w_\ell$, i.e.:

$$w_\ell^{(i+1)} = w_\ell^{(i)} + \Delta w_\ell^{(i)} \tag{20}$$

The on-line phase is effective and reliable when the error begins decreasing at the onset of training, and the update algorithm does not degrade prior network weights. Because of the high-dimensional nature of real-world applications, the neural networks implemented typically are large and their parameters differ by many orders of magnitude, causing the derivatives to be highly dissimilar. Speed and memory requirements are particularly stringent on line, rendering this training phase arduous in practice.

For these reasons, the resilient backpropagation algorithm (RPROP) [12] is modified and implemented. RPROP is based only on the temporal behavior of the signs of the gradients [12]. Therefore, it has low memory requirements and no dependence on the size of the derivatives. The individual size of each increment, denoted by Δ_ℓ, is increased by a factor η^+ when the derivative is not changing sign, while it is decreased by a factor η^- when the derivative is changing sign. This process accelerates convergence in shallow regions and slows the search down when local minima are missed. Once all Δ_ℓ are adjusted, each weight is modified in the direction of gradient descent. When the error derivative changes sign, indicating that a minimum was missed, the weight $w_\ell^{(i+1)}$ is brought back to its previous value $w_\ell^{(i-1)}$ by a *backtracking* epoch [12].

Backtracking is a key algorithmic feature that allows the search to remain local. Another crucial element is the initial increment value $\Delta_\ell^{(0)}$. Setting all initial increments equal to the same constant value (e.g., 0.1) for weights of dissimilar sizes [12] is equivalent to disregarding prior network weights. Instead, initial increments are chosen commensurate with a fraction, f_w, of the corresponding prior weights and perturbed by f_0 to account for zero weights:

$$\Delta_\ell^{(0)} = f_w\left|w_\ell\right| + f_0 \tag{21}$$

The same weight update routine is used for the action and the critic networks by letting $\mathbf{z}_D = \mathbf{u}_D(t_k)$ in the action update, and $\mathbf{z}_D = \lambda_D(t_k)$ in the critic update.

4. Adaptive Critic Proportional Integral Neural Network Control Design

The neural controller structure is motivated by a multivariable linear controller; proportional-integral (PI) control is considered for illustration. A PI controller modifies the stability and transient response of the plant through the feedback gain matrix, $\mathbf{C}_B$, and provides Type-1 response to command inputs through the proportional gain matrix, $\mathbf{C}_F$, and the command-integral gain matrix $\mathbf{C}_I$ [10]. These gains are computed by minimizing a cost function in the form of eq. 9 formulated in terms of the augmented state

$\mathbf{x}_a$ and the control deviation $\tilde{\mathbf{u}}$. $\mathbf{x}_a$ includes the state deviation $\tilde{\mathbf{x}}$ and the output error's time integral ξ, i.e., $\mathbf{x}_a \equiv [\tilde{\mathbf{x}}^T \;\; \xi^T]^T$, where $\tilde{\mathbf{x}} \equiv \mathbf{x} - \mathbf{x}_c$. $\tilde{\mathbf{u}}$ and $\tilde{\mathbf{y}}$ are similarly defined. The set point ($\mathbf{x}_c$, $\mathbf{u}_c$) is a function of the command input, $\mathbf{y}_c$, [6]. The LQ law (eq. 12) provides for the optimal control in terms of the newly defined deviations:

$$\tilde{\mathbf{u}}(t) = -\mathbf{C}_a \mathbf{x}_a(t) = -\mathbf{C}_B \tilde{\mathbf{x}}(t) - \mathbf{C}_I \xi(t) \tag{22}$$

The gains and the Riccati matrix $\mathbf{P}_a$ are obtained by solving a matrix Riccati equation [6] formulated in terms of $\mathbf{x}_a$ and $\tilde{\mathbf{u}}$. The weighting matrices **Q**, **M**, and **R**, are designed using implicit model following [10].

The corresponding neural network structure is obtained by replacing each linear gain matrix with a nonlinear control network, $\mathbf{NN}_B$ for $\mathbf{C}_B$, $\mathbf{NN}_F$ for $\mathbf{C}_F$, and $\mathbf{NN}_I$ for $\mathbf{C}_I$ [10]. In addition to the scheduling vector, the networks $\mathbf{NN}_B$, $\mathbf{NN}_F$, and $\mathbf{NN}_I$ are provided with the state deviation, the command input, and the command error integral, respectively. Each network contributes to the total control,

$$\begin{aligned} \mathbf{u}(t) &= \mathbf{u}_c(t) + \Delta\mathbf{u}_B(t) + \Delta\mathbf{u}_I(t) \\ &= \mathbf{NN}_F[\mathbf{y}_c(t), \mathbf{a}(t)] + \mathbf{NN}_B[\tilde{\mathbf{x}}(t), \mathbf{a}(t)] + \mathbf{NN}_I[\xi(t), \mathbf{a}(t)] \end{aligned} \tag{23}$$

where $\tilde{\mathbf{u}} = \Delta\mathbf{u}_B + \Delta\mathbf{u}_I$ is the control to be optimized.

The action network, $\mathbf{NN}_A$, that approximates the minimizing control law consists of the algebraic sum of $\mathbf{NN}_B$ and $\mathbf{NN}_I$:

$$\tilde{\mathbf{u}}(t) = \mathbf{NN}_A[\mathbf{x}_a(t), \mathbf{a}(t)] \tag{24}$$

Given the same inputs, the critic network, $\mathbf{NN}_C$, computes the derivative of the value function $V[\mathbf{x}_a(t)]$ with respect to the augmented state:

$$\boldsymbol{\lambda}_a(t) \equiv \frac{\partial V[\mathbf{x}_a(t)]}{\partial \mathbf{x}_a(t)} = \mathbf{NN}_C(\mathbf{x}_a(t), \mathbf{a}(t)) \tag{25}$$

The final neural controller structure is shown in Fig. 5. The *Scheduling Variable Generator* (*SVG*) produces **a** based on $\mathbf{y}_c$ and an exogenous vector, **e**, of measured variables. The *Command State Generator* (*CSG*) provides secondary elements of the state that are compatible with $\mathbf{y}_c$.

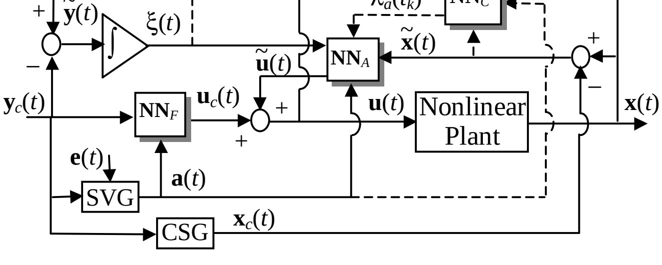


Figure 5. Action critic neural network controller.

Subsequently, eq. 17 and 18 formulated in terms of $\mathbf{x}_a$ and $\tilde{\mathbf{u}}$ are used on-line to adapt the action and the critic network, respectively.

5. Flight Control Simulation and Results

The adaptive controller is implemented on a six-degree-of-freedom business jet aircraft model. The simulation explores the full flight envelope, $OR = \{V, H, \gamma, \mu, \beta\}$. The control design is based on the state, $\mathbf{x} = [V\ \gamma\ q\ \theta\ r\ \beta\ p\ \mu]^T$, comprising airspeed V (m/s), path angle γ (rad), pitch rate q (rad), pitch angle θ (rad), yaw rate r (rad/s), sideslip angle β (rad), roll rate p (rad/s), and bank angle μ (rad). The independent controls being generated are throttle δT (%), stabilator δS (rad), aileron δA (rad), and rudder δR (rad); i.e., $\mathbf{u} = [\delta T\ \delta S\ \delta A\ \delta R]^T$. The command, $\mathbf{y}_c = [V_c\ \gamma_c\ \mu_c\ \beta_c]^T$, contains the state elements that, given the altitude H (m), uniquely specify a longitudinal-lateral-directional steady maneuver, postulating $\dot{\phi}_c = \dot{\theta}_c = 0$ with ϕ as the Euler roll angle. All angular and kinematic relations involved pertain to non-longitudinal, non-level flight, and are based on spherical trigonometry [13].

The neural control architecture specified in Section 4 is initialized based on the performance criteria established locally by the linear gains and the augmented Riccati matrix, with $\mathbf{NN}_F$ approximating the aircraft trim map [14]. Independent longitudinal and lateral-directional linear models are obtained for a set, *OP*, of thirty-four operating points chosen from $\{V, H\} \subset OR$, letting $\gamma_0 = \mu_0 = \beta_0 = 0$. Corresponding linear controllers are designed and used independently to initialize longitudinal and lateral-directional control networks [10]. Each initialized pair is algebraically joined into a full longitudinal-lateral-directional neural network. Then, the full feedback and command-integral networks, $\mathbf{NN}_B$ and $\mathbf{NN}_I$, are algebraically summed to form the action network, $\mathbf{NN}_A$.

During every time interval (0.1 sec), the critic and action networks are updated by the modified RPROP algorithm of Section 3.2 based on the respective targets, $\tilde{\mathbf{u}}_D$ and $(\boldsymbol{\lambda}_a)_D$ (Figs. 3 and 4). The user-defined update parameters are: $\eta^+ = 1.2$, $\eta^- = 0.5$ [12], $f_w \sim O(10^{-5})$, and $f_0 << 1$. The modified RPROP algorithm (Section 3.2) is validated by comparing its performance to that of the *MATLAB 5.3* "trainrp" learning function, for the update of the action network at $t = 0.2$ sec, as illustrated in Fig. 6. Further studies also show that the modified RPROP better preserves the original weights and avoids overfitting.

During the on-line phase, the update algorithm terminates after the mean-squared measure of each network error $[\mathbf{z}_D - \mathbf{z}(\mathbf{w})]$ has decreased by 10% and at least 3 epochs have elapsed. When more than 3 epochs are needed to decrease the network error by this amount, the terminating value of Δ_ℓ is saved and used as $\Delta_\ell^{(0)}$ for the next time interval, for $\forall\ \ell$. This typically requires several epochs during the first 0.2 sec to adjust the increment size, and three epochs during later time intervals to decrease the network error. The initialization phase and the modified RPROP algorithm render the on-line phase feasible and efficient with respect to both time and storage consumption.

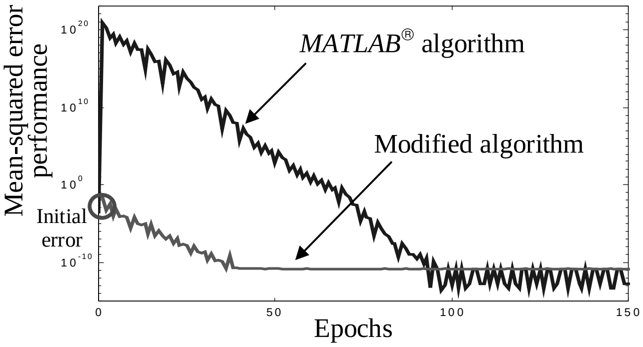


Figure 6. Comparison between the *MATLAB*® RPROP algorithm and its modified version ($\mathbf{NN}_A$, at t = 0.2 s).

Histories of the state elements directly commanded by $\mathbf{y}_c$ are used to evaluate performance during a large-angle asymmetric maneuver and are plotted with a solid line in Fig. 7. At the initial time, the aircraft is flying level at a nominal airspeed V_0 of 95 m/s and an altitude H_0 of 2, 000 m, with $(V_0, H_0) \not\subset OP$. The state response is judged against an equivalent PI neural network controller (represented by a dashed line in Fig. 7) that is initialized with the same linear data but does not undergo on-line adaptation. The comparison shows that on-line adaptation brings about an improvement with respect to the linear controllers. The action and critic networks minimize the cost-to-go on line, in the presence of coupling and nonlinear effects unaccounted for by the linear designs, and they do so without unlearning previous information.

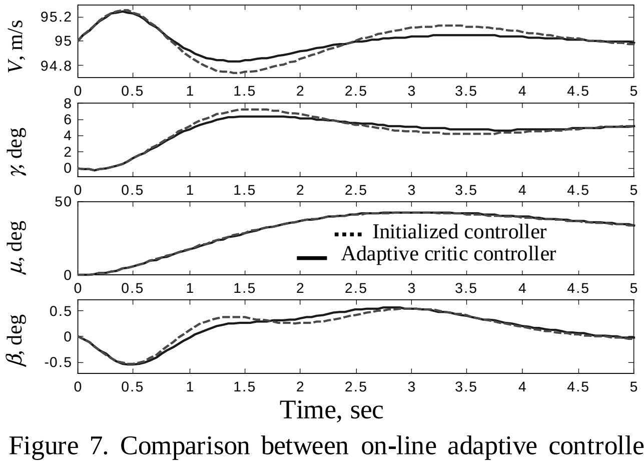


Figure 7. Comparison between on-line adaptive controller and initialized controller at (V_0, H_0) = (95 m/s, 2 Km), subject to 5-deg climb angle and 30-deg roll step command.

6. Conclusions

Advances in off-line and on-line learning techniques and in adaptive critic methods are presented and incorporated in a novel approach to neural control system design. The nonlinear control system is taught using a two-phase learning procedure encompassing an initialization phase that provides for reliability, and an on-line phase that accounts for actual plant dynamics. Both phases are founded on optimal control theory and are realized with significant computational savings. The nonlinear adaptive controller is successfully implemented for the command-input control of a full-scale aircraft simulation, with the neurocontrollers adjusting on line, while retaining their baseline performance. The adaptive controllers spontaneously utilize those parameters that were unused during initialization to learn newly available information. Also, a modified resilient backpropagation algorithm allows the networks to improve their performance in only one or few epochs over each time increment. The advancements of all key design stages combined bring about concrete potential for real-life applications.

Acknowledgement

This research has been supported by the Federal Aviation Administration and the National Aeronautics and Space Administration under FAA Grant No. 95-G-0011.

References

[1] Kirk, D. E., *Optimal Control Theory; and Introduction*, Prentice-Hall, Englewood Cliffs, NJ, 1970.

[2] Bellman, R. E., *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[3] Howard, R., *Dynamic Programming and Markov Processes*, MIT Press, Cambridge, MA, 1960.

[4] White, D. A, Sofge, D., *Handbook of Intelligent Control*, Van Nostrand Reinhold, New York, 1992.

[5] Narendra, K. S., Parthasaranthy, K., "Identification and control of dynamical systems using neural networks", *IEEE Trans. Neural Networks*, Vol. 1, 1990, pp. 4-27.

[6] Stengel, R. F., *Optimal Control and Estimation*, Dover Publications, Inc., New York, 1994.

[7] Barron, A. R., "Universal Approximation Bounds for Superposition of a Sigmoidal Function," *IEEE Transactions on Information Theory*, Vol. 39, No. 3, 1993, pp. 930-945.

[8] Åström, K. J., Stewart, G. W., "Solution of the Matrix Equation *AX + XB = C*," *Communications of the ACM*, Vol. 15, 1972, pp. 820-826.

[9] Stengel, R. F., Marrison, C., "Design of Robust Control Systems for Hypersonic Aircraft," *J. Guidance, Control, and Dynamics*, Vol. 21, No.1, 1997, pp.58-63.

[10] Ferrari, S., Stengel, R. F., "Classical/Neural Synthesis of Nonlinear Control Systems," to appear in *J. Guidance, Control and Dynamics*.

[11] Werbos, P. J., "Backpropagation Through Time: What It Does and How to Do It," *Proc. IEEE*, Vol. 78, 1990, pp. 1550-1560.

[12] Reidmiller, M., Braun, H., "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm," *Proc. IEEE Int. Conf. on NN (ICNN)*, San Francisco, CA, 1993.

[13] Kalviste, J., "Spherical Mapping and Analysis of Aircraft Angles for Maneuvering Flight," *J. Aircraft*, Vol. 24, No. 8, 1987, pp. 523-530.

[14] Ferrari, S., Stengel, R. F. "Algebraic Training of a Neural Network," *Proc. American Control Conference*, Arlington, VA, 2001.

Backpropagation through time and and its relationship with derivative adaptive critics

Danil Prokhorov, Ford Research Lab, Dearborn, MI
(dprokhor@ford.com)

Abstract

We discuss various forms of backpropagation through time (BPTT) and their differences with derivative adaptive critics.
We show that, in fact, BPTT is used in training derivative adaptive critics which means that the two approaches are closely related.
Our example suggests a problem challenging to solve with either approaches.

Bibliography

D. Prokhorov, L. Feldkamp, and I. Tyukin, "Adaptive Behavior with Fixed Weights in RNN: An Overview", to appear in Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, WCC'02, Honolulu, Hawaii, May 2002.

D. Prokhorov, G. Puskorius, and L. Feldkamp, "Dynamical Recurrent Networks for Control," in S. Kremer and J. Kolen (Eds.), Field Guide to Dynamic Recurrent Networks, IEEE Press, 2001.

P. Eaton, D. Prokhorov, and D. Wunsch, "Neurocontroller Alternatives to "Fuzzy" Ball-and-Beam Systems with Nonuniform Nonlinear Friction," IEEE Trans. on Neural Nets, March 2000, pp. 423-435.

L. Feldkamp and D. Prokhorov, "Observations on the Practical Use of Derivative Adaptive Critics." In Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'97), Orlando, FL, October, 1997, pp. 3061-3066.

D. Prokhorov and L. Feldkamp. "Primitive Adaptive Critics." In Proceedings of the IEEE/INNS International Joint Conference on Neural Networks (ICNN'97), Houston, TX, pp. 2263-2267, June 1997, pp. 2263-2267.

Research

dprokhor@ford.com

Backpropagation Through Time and its Relationship with Derivative Adaptive Critics

Danil Prokhorov
Group of Artificial Neural Systems
Ford Research Laboratory
Dearborn, MI

Outline

- Background
  (heterogeneous ordered system,
  differentiable optimization with quadratic criterion)
- Backpropagation through time (BPTT) and how it can be used in training derivative adaptive critic (DAC)
- Types of BPTT
- Exact match between BPTT and DAC derivatives (example)
- Summary
- Challenging problem
- Conclusions

Heterogeneous ordered system (general form)

$x_i(t)=x_i^{ext}(t),$ 1 £ i £ m

$x_i(t)= f_i\left(x_1(t),x_2(t),\ldots,x_{i-1}(t), x_{m+1}(t-1),x_{m+2}(t-1),\ldots,x_N(t-1),\Psi\right)$

m+1 £ i £ N

- Order of computing x_i is from 1 to N
- f_i are differentiable
- Ψ are parameters to be adapted

Optimization

Differentiable optimization with criterion

$$J(k)=\tfrac{1}{2}\sum_{t=k}^{k+h} g^{t-k}\sum_{j=N_1}^{N_2} U_j^2(t),$$

discount 0<g £ 1,

depth *h* is as large as required,

utility or error between desired and actual states $U_j(t) = x_j^d(t) - x_j(t)$, *m+1 £ [N*$_1$*,N*$_2$*] £ N.*

Find Ψ such that *J* is optimized in domain of interest.

Heterogeneous ordered network

$x_i(t)=x_i^{ext}(t),$ 1 £ i £ m

$$net_i(t)=\sum_{j=1}^{i-1}W_{ij}(t)x_j(t)+\sum_{j=m+1}^{N}W_{ij}^1(t)x_j(t-1),$$

$x_i(t)=f_i(net_i(t)),$ m+1 £ i £ N

- Order of computing x_i is from 1 to N
- f_i are differentiable

Ordered derivative (example)

Given x_1 compute

$$x_2 = \sin(x_1)$$
$$z = x_1 + 2x_2$$

Order of computation is specified (x_1 first, x_2 second, *z* last).

$\frac{\P z}{\P x_1} = 1$ is ordinary partial derivative.

$$F\_x_1 = \frac{\P z}{\P x_1} + \frac{\P z}{\P x_2}\frac{\P x_2}{\P x_1} = 1 + 2\cos(x_1)$$

F_x$_1$ is ordered derivative of *z* with respect to x_1.

Backpropagation Through Time (BPTT(h))

Differentiable optimization with criterion

$$J(k)=\tfrac{1}{2}\sum_{t=k}^{k+h} g^{\,t-k}\sum_{j=N_1}^{N_2} U_j^2(t),$$

Ordered derivative of J with respect to $x_i(t)$

$$F\_x_i(t)=U_i(t)\frac{\P U_i(t)}{\P x_i(t)}+\sum_{j=i+1}^{N} W_{ji}(t)\frac{f_j(net_j(t))}{\P net_j(t)}F\_x_j(t)$$

$$+g\sum_{j=m+1}^{N} W_{ji}^1(t+1)\frac{f_j(net_j(t+1))}{\P net_j(t+1)}F\_x_j(t+1),$$

$F\_x_i(k+h+1) = 0,$ $W^1(k+h+1) = W^1(k+h),$

$i = N, N\text{-}1, \ldots, 1;$ $t = k+h, k+h\text{-}1, \ldots, k.$

BPTT(h): derivatives with respect to weights

$$F\_W_{ij}(k)=F\_x_i(k)\frac{\P f(net_i(k))}{\P net_i(k)}x_j(k),$$

$$F\_W_{ij}^1(k)=F\_x_i(k)\frac{\P f(net_i(k))}{\P net_i(k)}x_j(k-1),$$

These derivatives are used to adapt W_{ij} and W^1_{ij} in order to optimize J.

DAC: l critic

Differentiable optimization with moving target

$$J'(t)=\tfrac{1}{2}\sum_{j=N_1}^{N_2} U_j^2(t)+gJ'(t+1),$$

Differentiating with respect to $x_i(t)$ and taking into account ordered nature of network we obtain

$$F\_x_i(t)=U_i(t)\frac{\P U_i(t)}{\P x_i(t)}+\sum_{j=i+1}^{N} W_{ji}(t)\frac{f_j(net_j(t))}{\P net_j(t)}F\_x_j(t)$$

$$+g\sum_{j=m+1}^{N} W_{ji}^1(t+1)\frac{f_j(net_j(t+1))}{\P net_j(t+1)}\cancel{F\_x_j}(t+1),$$

to train critic $e_i(t)=F\_x_i(t)-l_i(t).$

$$l_j(t+1)=\frac{\P J'(t+1)}{\P x(t+1)}$$

F_W_{ij} and $F\_W^1_{ij}$ are the same as in BPTT(h) ($k{=}t$).

DAC: alternative

k critic

$$F\_x_i(t+1)=k_i(t+1)$$

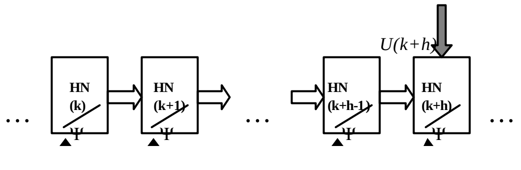
$$+U_i(t+1)\frac{\P U_i(t+1)}{\P x_i(t+1)}+\sum_{j=i+1}^{N} W_{ji}(t+1)\frac{f_j(net_j(t+1))}{\P net_j(t+1)}F\_x_j(t+1),$$

$$F\_x_j(t)=g\sum_{j=m+1}^{N} W_{ji}^1(t+1)\frac{f_j(net_j(t+1))}{\P net_j(t+1)}F\_x_j(t+1),$$

to train critic $e_i(t)=F\_x_i(t)-k_i(t).$

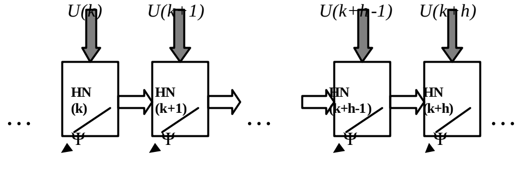
- provides smooth integration between DAC and BPTT(h)

Popular form of BPTT(h)

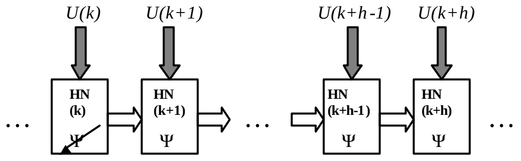


Ψ is subset of weights (W,W 1) in heterogeneous network (HN) to be adapted

Aggregate form of BPTT(h)



Alternative form of BPTT(h)



It matches derivatives from l critic if h→∞.

Simple example

$$x(k+1) = x_0(k+1) + wx(k)$$
$$x_d(k+1) = x_0(k+1) + w_d x_d(k) \quad (*)$$

$0 < |w, w_d| < 1$

$x_0(.)$ is i.i.d.; average is x_0

$$J(k) = \tfrac{1}{2}\sum_{t=0}^{\infty} g^t (x_d(k+t) - x(k+t))^2 \qquad (**)$$

Differentiating (**) with respect to $x(k)$

$$F\_x(k) = \sum_{t=0}^{\infty} (gw)^t (x(k+t) - x_d(k+t))$$

Taking into account (*) and averaging for all $x_0(.)$

$$\langle F\_x(k)\rangle = \frac{x(k)}{1-gw^2} - \frac{x_d(k)}{1-gww_d} + \frac{x_0 gw}{1-gw}\left(\frac{1}{1-gw^2} - \frac{1}{1-gww_d}\right)$$

Simple example (cont’d)

$$l(t) = Ax(t) + Bx_d(t) + C$$

$$l(t) = x(t) - x_d(t) + g\,l(t+1)\frac{\P x(t+1)}{\P x(t)}$$

$$Ax(t) + Bx_d(t) + C = x(t) - x_d(t) + gw\big(Ax(t+1) + Bx_d(t+1) + C\big)$$

At convergence: $A = \dfrac{1}{1-gw^2}$

$$B = -\frac{1}{1-gww_d}$$

$$C = \frac{gwx_0}{1-gw}\left(\frac{1}{1-gw^2} - \frac{1}{1-gww_d}\right)$$

$\Rightarrow \langle F\_x(k)\rangle$ can be restored exactly!

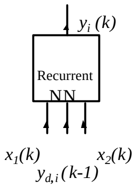
Derivatives by BPTT vs. DAC

- BPTT derivatives are computed directly, while critic derivatives are computed from a representation, e.g., a neural network; representation parameters must be learned.
- BPTT(h) derivatives generally involve a finite time horizon (equal to chosen truncation *h*), while critic derivatives are estimates for an infinite horizo n (possibly with gentle truncation). Large *h* are permissible due to linear scaling of computations, but often small *h* suffice.
- BPTT derivatives necessarily compute effect of changing a varia ble in the *past*, while a derivative critic may be used to estimate effect of a change at the present time. If critics are used only to adjust controller parameters, this distinction is irrelevant.
- BPTT derivative is essentially*exact* for specific trajectory for which it is computed, while a critic derivative is expected to estimate *an average* over trajectories that begin with a given state. Such estimate may be quite accurate (in slowly changing or statistically well behaved environments) or may be essentially worthless (in fast changing or unpredictable envir onments).

Interesting problem

Learning all quadratic functions of two variables:

$y_d = 0.25(\, a(x_1)^2 + b\,(x_1)^2 + cx_1x_2 + dx_1 + ex_2 + f\,),$
$a, b, c, d, e, f, x_1, x_2$ in [-1,+1].



Training set (1000 points per function)

function 1 | function 2 | … | function 128

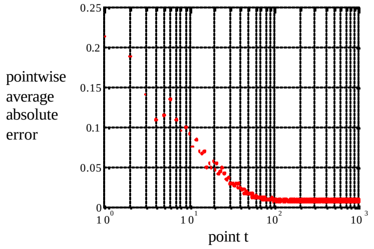
$(a, b, c, d, e, f)_i$

$y_{d,i}\,(k\text{-}1), x_1(k), x_2(k)$ ▸ $y_i\,(k)$ vs. $y_{d,i}\,(k)$

i = 1, 2, …, 128

Test results

- Recurrent multilayered perceptron with 40 states (> 1400 weights)
- Each data point is trained on about 250 times
- RMSE is reduced from 0.260 to 0.020 in training
- Test RMSE < 0.025 (many different test sets)



pointwise average absolute error (t) =

$$\frac{\sum_{i=1}^{128} |\, y_{d,i}(t) - y_i(t)\,|}{128}$$

t = 1, 2, …, 1000

Conclusions

- BPTT(h) can be successfully applied to all problems to which derivative adaptive critics are applicable
- BPTT(h) obviates the need for using a critic in problems with specified *U(x)* (utility as function of states)
- A critic may be needed if *U(x)* is to be learned ($\hat{U}(x)$ and its derivatives are to be provided by such critic).

Research

dprokhor@ford.com

Bibliography

- Prokhorov, D., Feldkamp, L., and Tyukin, I., "Adaptive Behavior with Fixed Weights in RNN: Overview." In *Proc. IJCNN'02* (to appear).
- Prokhorov, D., Puskorius, G., and Feldkamp, L. Dynamical Recurrent Networks for Control. In Kolen and Kremer (Eds.), *A Field Guide to Dynamical Recurrent Networks*, IEEE Press, 2001.
- Feldkamp, L., and Prokhorov, D., "Observations on the Practical Use of Derivative Adaptive Critics." In *Proc. IEEE SMC'97*, Orlando, FL, October 1997.
- Feldkamp, L., Puskorius, G., and Prokhorov, D., "Unified Formulation for Training Recurrent Networks with Derivative Adaptive Critics." In *Proc. IJCNN'97*, Houston, TX, June 1997.
- Prokhorov, D. Ph.D. dissertation, Texas Tech University, 1997.
- Werbos, P., "BPTT: What it Does and How to Do it." In *Proc. IEEE*, Vol. 78, October 1990.

What's beyond … for ACDs?

Donald C. Wunsch
(dwunsch@ece.umr.edu)

Abstract

Adaptive Critic Designs are a unifying framework for understanding much of, perhaps most of, the progress in reinforcement learning over the last quarter century. Unfortunately, despite this unifying framework, much of the field has become fragmented. Part of this is because the applications are fragmented, and part of it is because contributions come from researchers of diverse backgrounds. This talk will overview some of the main research trends that are grappling with the same underlying foundations. It will conclude with some nontraditional architectures and examples for future applications of Adaptive Critic Designs.

What’s Beyond … For ACDs?

Donald Wunsch

Applied Computational Intelligence Laboratory
University of Missouri - Rolla

What’s Beyond … For ACDs?

Adaptive Critic Designs are a unifying framework for understanding much of, perhaps most of, the progress in reinforcement learning over the last quarter century. Unfortunately, despite this unifying framework, much of the field has become fragmented. Part of this is because the applications are fragmented, and part of it is because contributions come from researchers of diverse backgrounds. This talk will overview some of the main research trends that are grappling with the same underlying foundations. It will conclude with some nontraditional architectures and examples for future applications of Adaptive Critic Designs.

Acknowledgements

- Funding
  - NSF
  - Sandia
  - Boeing
  - MK Finley Professorship
- Senior Personnel
  - Ganesh Kumar Venayagamoorthy
  - Ron Harley
  - Daryl Beetner
  - Danil Prokhorov
  - Raonak Uz-Zaman
  - Frank Harary
- Personnel
  - Narayan Vishwanathan
  - Amit Agarwahl
  - Sam Mulder
  - Wenxin Liu
  - Nian Zhang
  - Alexander Novokhodko
  - Xindi Cai
  - Rohit Dua
  - Hu Xiao
  - Rui Xu
  - Brian Blaha
  - Paul Pigg
  - Arvind Rapka Nath

Outline

- Caveats
- Applications
- The difference between critics and fans

Caveats

- Scientists aren’t good at predictions about science…
  - Who knows? Quantum Computing, etc…
- Tough…?
- Open…?
- Neurocomputing…?

- BUT…

Extrapolating from what we know…

- What ACDs do well
- What NNs and ACDs SHOULD do well
- NOT What NNs Don’t do well
- NOT about human intelligence
  - Fly, maybe …

What ACDs do well

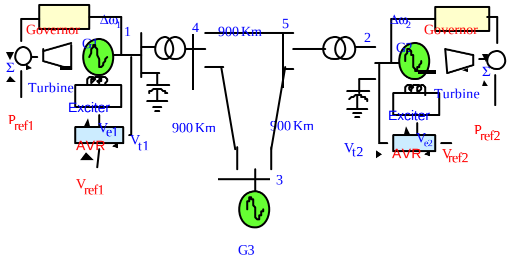
- Control
- Optimization (newer)

Power System Control

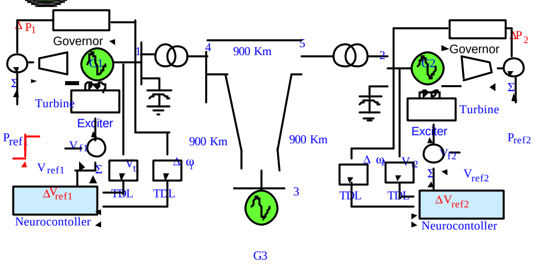
- Critical problem at present
- Significant performance improvements needed
- Nonlinear, time-varying problem
- Multiple controllers – significant learning issues

Multi-Machine Power System

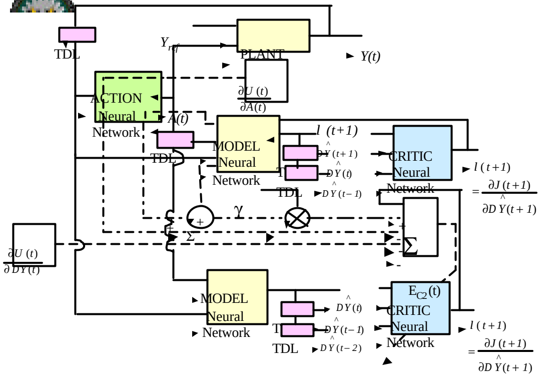
Multi-Machine Power System with Conventional Controllers



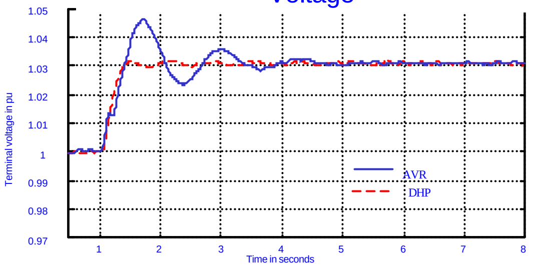
Multi-Machine Power System with DHP Neurocontrollers



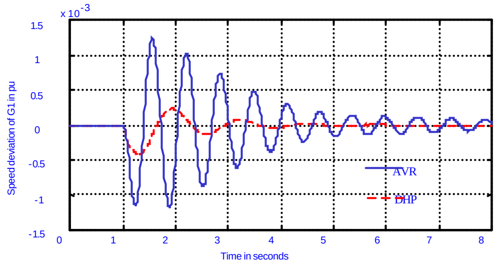
DHP Critic Network Adaptation



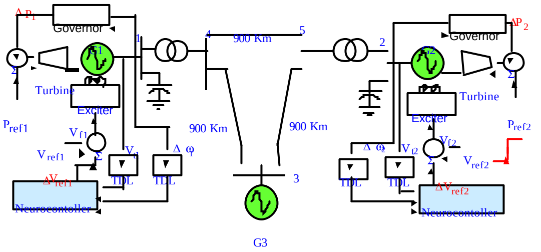
Terminal Voltage of Generator G1 for a 3% Step Change in its Desired Terminal Voltage



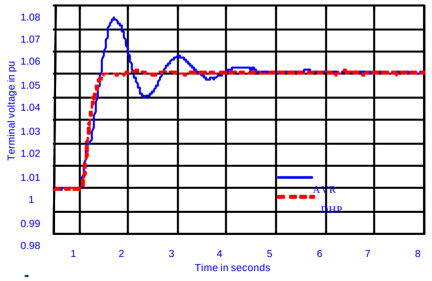
Speed Deviation of Generator G1 for a 3% Step Change in its Desired Terminal Voltage



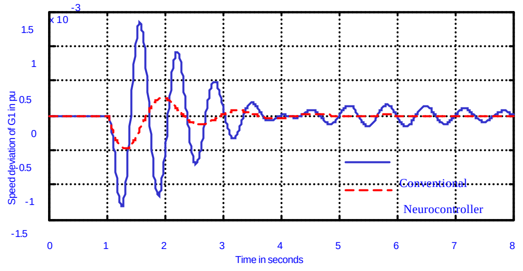
Multi-Machine Power System with DHP Neurocontrollers



Terminal Voltage of Generator G2 for a 5% Step Change in its Desired Terminal Voltage & Operating Point Changed



Speed Deviation of Generator G2 - Operating Point Changed

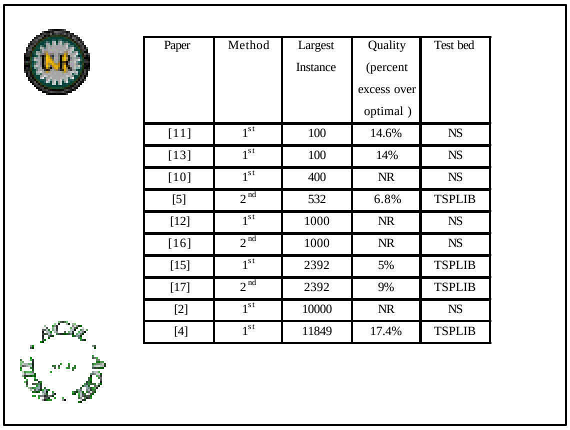


Where NNs SHOULD do well

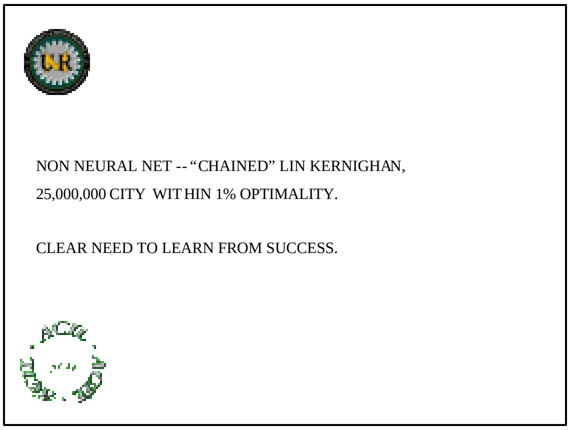
WEAK PERFORMANCE OF NN TO DATE ON TSP:

Hopfield nets 200 cities, 8% > optimal

SOFM 11849 city TSPLIB instance, 17.4% > optimal

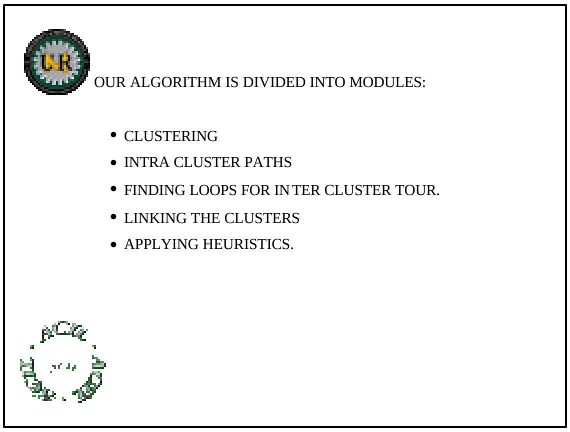


Paper	Method	Largest Instance	Quality (percent excess over optimal)	Test bed
[11]	1st	100	14.6%	NS
[13]	1st	100	14%	NS
[10]	1st	400	NR	NS
[5]	2nd	532	6.8%	TSPLIB
[12]	1st	1000	NR	NS
[16]	2nd	1000	NR	NS
[15]	1st	2392	5%	TSPLIB
[17]	2nd	2392	9%	TSPLIB
[2]	1st	10000	NR	NS
[4]	1st	11849	17.4%	TSPLIB



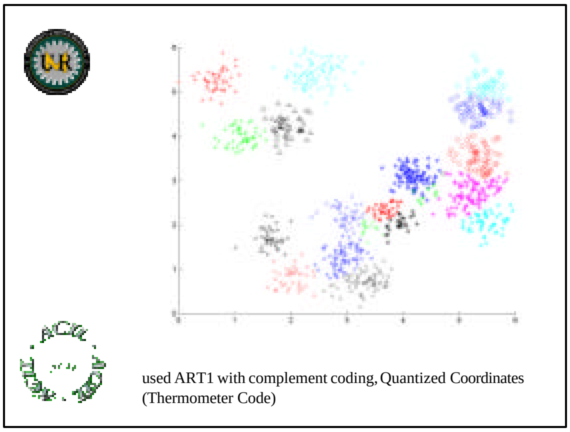
NON NEURAL NET -- "CHAINED" LIN KERNIGHAN,
25,000,000 CITY WITHIN 1% OPTIMALITY.

CLEAR NEED TO LEARN FROM SUCCESS.

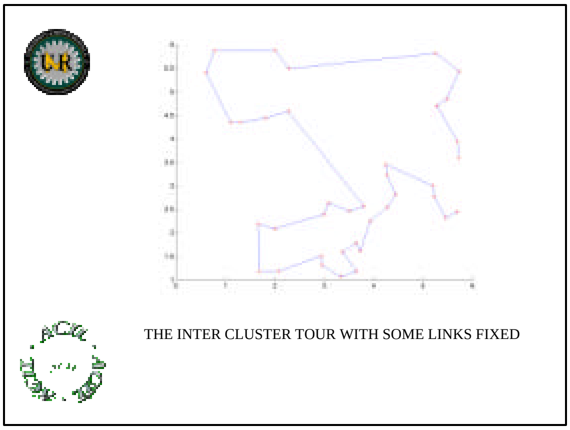


OUR ALGORITHM IS DIVIDED INTO MODULES:

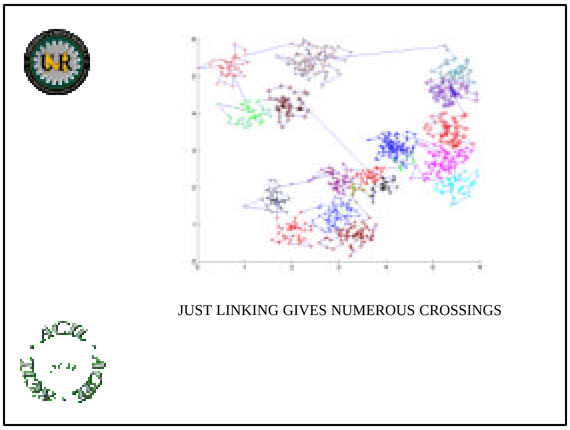
- CLUSTERING
- INTRA CLUSTER PATHS
- FINDING LOOPS FOR INTER CLUSTER TOUR.
- LINKING THE CLUSTERS
- APPLYING HEURISTICS.



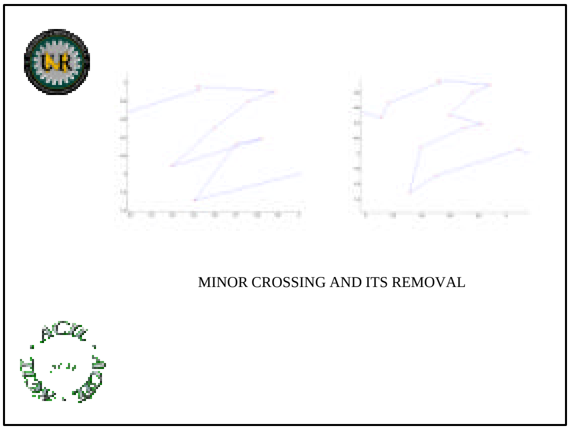
used ART1 with complement coding, Quantized Coordinates (Thermometer Code)



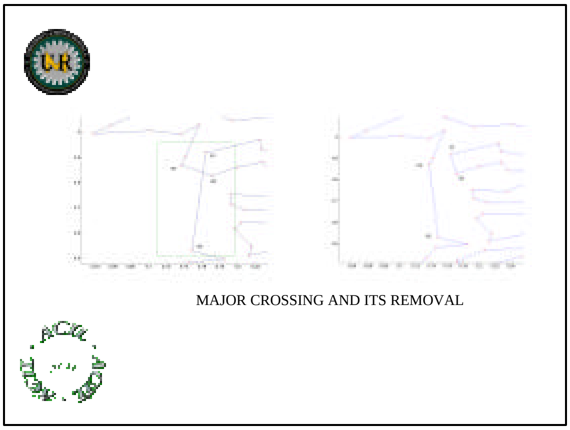
THE INTER CLUSTER TOUR WITH SOME LINKS FIXED



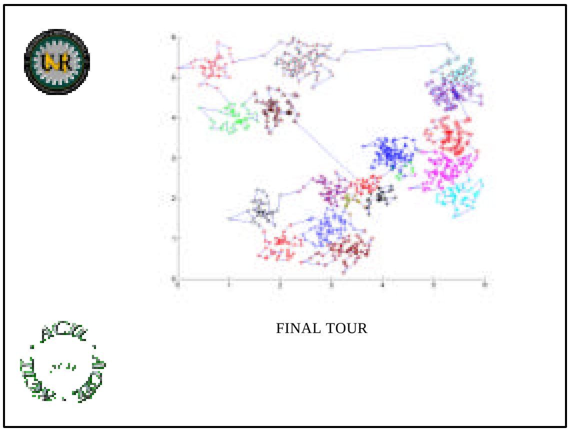
JUST LINKING GIVES NUMEROUS CROSSINGS



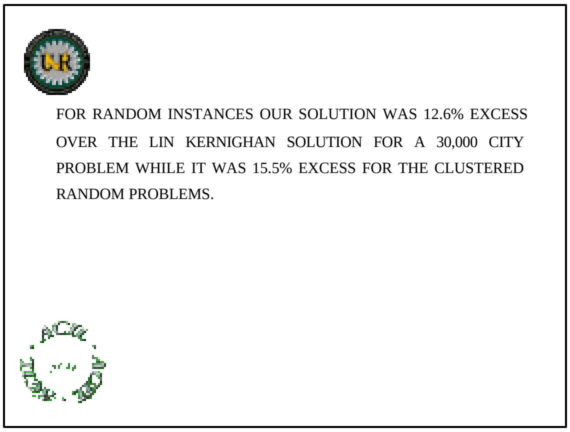
MINOR CROSSING AND ITS REMOVAL



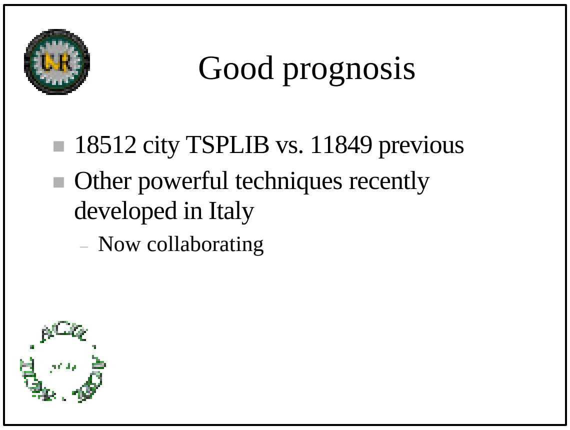
MAJOR CROSSING AND ITS REMOVAL



FINAL TOUR

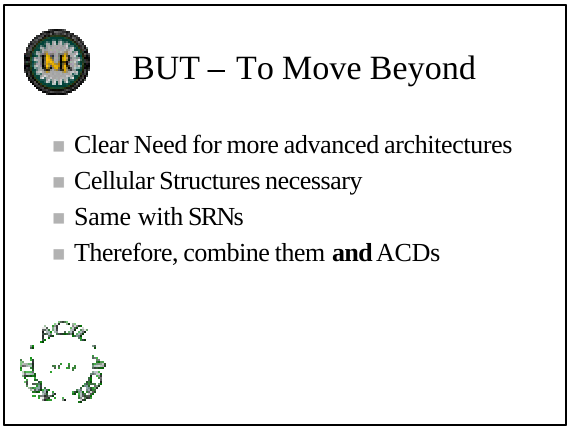


FOR RANDOM INSTANCES OUR SOLUTION WAS 12.6% EXCESS OVER THE LIN KERNIGHAN SOLUTION FOR A 30,000 CITY PROBLEM WHILE IT WAS 15.5% EXCESS FOR THE CLUSTERED RANDOM PROBLEMS.



Good prognosis

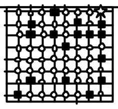
- 18512 city TSPLIB vs. 11849 previous
- Other powerful techniques recently developed in Italy
  - Now collaborating



BUT – To Move Beyond

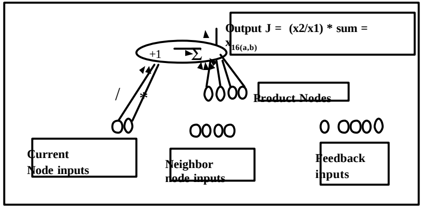
- Clear Need for more advanced architectures
- Cellular Structures necessary
- Same with SRNs
- Therefore, combine them **and** ACDs

The Generalized Maze Problem



- Graph Theoretic Representation
- SRN Necessary (Werbos & Pang, ’96 & ’98)
- Cellular structure – scaling
- Closed form now
- Convergence time now
- Importance of design principles

Cellular SRN Structure Complete



Output J = (x2/x1) * sum = $x_{16(a,b)}$

+1

Σ

/

*

Product Nodes

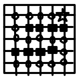
Current Node inputs

Neighbor node inputs

Feedback inputs

(Occurs at each node (a,b) in maze.)

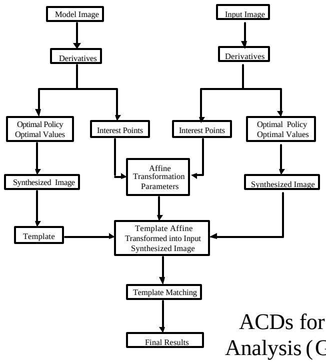
Analyze convergence from worst case for 5 x 5 maze.



WCT = $N^2 - 2N + N - 3 = N^2 - N - 3$.

Note that this is convergence in J steps.

Also true for N x N maze by simple induction proof.



Model Image

Derivatives

Optimal Policy Optimal Values

Interest Points

Synthesized Image

Template

Input Image

Derivatives

Interest Points

Optimal Policy Optimal Values

Synthesized Image

Affine Transformation Parameters

Template Affine Transformed into Input Synthesized Image

Template Matching

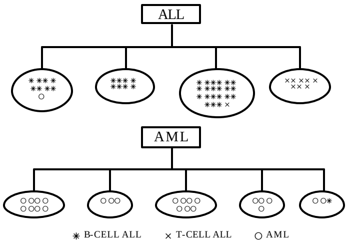
Final Results

ACDs for Image Analysis (Ge, TTU, 2002)

Reconfigurable control

- Want fast online learning plus generalization performance
- So far, better to push offline learning capabilities

Bioinformatics



ALL

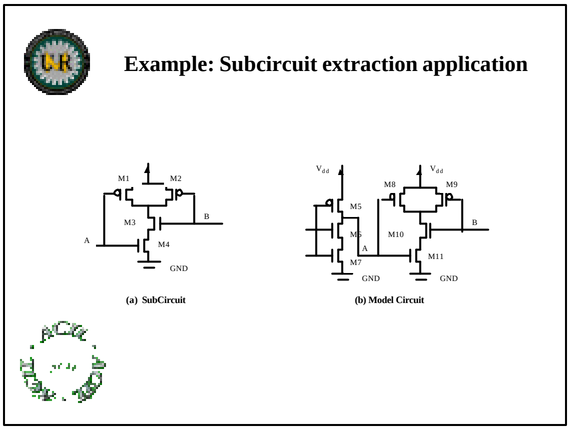
AML

* B-CELL ALL

× T-CELL ALL

○ AML

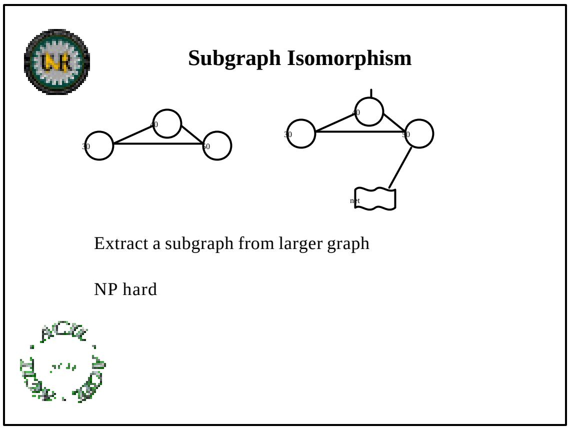
Clusters of ALL and AML by Ellipsoidal ARTMAP – but Reinforcement and Kernel-based approaches are in the future



Example: Subcircuit extraction application

(a) SubCircuit

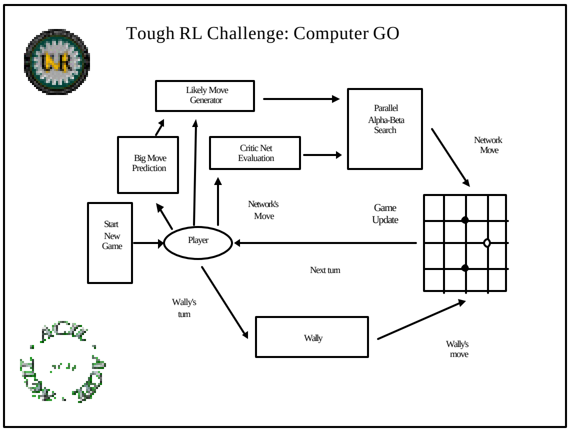
(b) Model Circuit



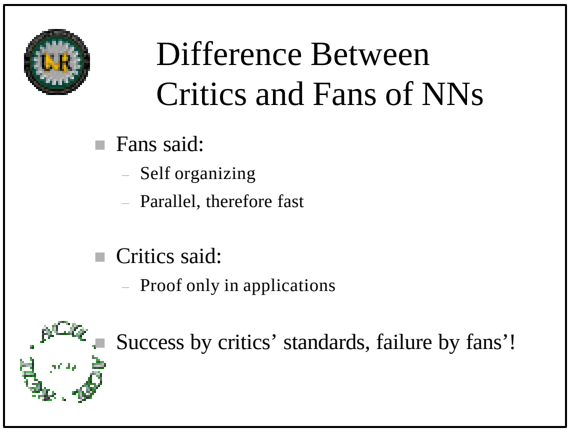
Subgraph Isomorphism

Extract a subgraph from larger graph

NP hard



Tough RL Challenge: Computer GO



Difference Between Critics and Fans of NNs

- Fans said:
  - Self organizing
  - Parallel, therefore fast
- Critics said:
  - Proof only in applications
- Success by critics' standards, failure by fans'!

ADAPTIVE CRITICS FOR CONTROL OF NONLINEAR AND DISTRIBUTED PARAMETER SYSTEMS

S.N.Balakrishnan
(bala@umr.edu)

Abstract

Optimal control of different and difficult nonlinear systems and distributed parameter systems with adaptive critics are presented in this talk. Formulation and results from a nonlinear aircraft problem, a missile problem and a heat diffusion problem are presented. Furthermore, results from *implementation* of adaptive critic based controllers to vibration control and heat diffusion problems are presented. Methods to deal with robustness are discussed. Methods of handling control and state variable inequality constraints are also presented with application to an agile missile.

Adaptive-Critics for Control of Nonlinear Systems

S.N. Balakrishnan

Professor of Aerospace Engineering

University of Missouri-Rolla

Dynamic Programming

- System Model

$$x(k+1) = f(x(k), u(k))$$

- Cost Function

$$J(x(k)) = \min_{u(k)}[J(x(k+1)) + U(x(k), u(k))]$$

- Define Co-state $\lambda(x(k))$ as

$$\lambda(x(k)) = \frac{\partial J(x(k))}{\partial x(k)}$$

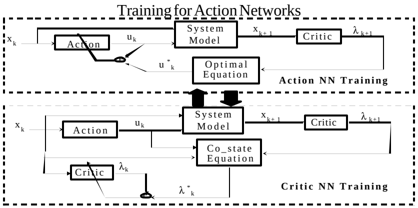
- Optimality Equation

$$\frac{\partial J(x(k))}{\partial u(k)} = 0$$

Dual Neural Network Controller Synthesis

- State Equation $\mathbf{x(k+1) = f(x(k)) + Bu(k)}$
- Co-state Equation $l\,\mathbf{(k) = Qx(k)} + \left(\mathbf{\frac{df(x(k))}{dx(k)}}\right)^{\mathbf{T}} l\,\mathbf{(k+1)}$
- Optimality Equation $\mathbf{u(k) = -R^{-1}B^{T}}\, l\,\mathbf{(k+1)}$

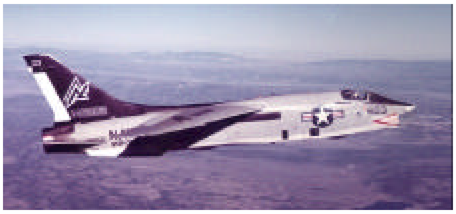
Training for Action Networks



Training for Critic Networks

Nonlinear Flight Control

Sergio Esteban Roncero



Dynamics of the F8 Crusader

- Nonlinear Plant

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ \dot{x}_3\end{bmatrix} = \begin{bmatrix}-0.877 & 0 & 1\\ 0 & 0 & 1\\ -4.208 & 0 & -0.396\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix} + \begin{bmatrix}-x_1^2x_3 - 0.088x_1x_3 - 0.019x_2^2 + 0.47x_1^2 + 3.846x_1^3\\ 0\\ -0.47x_1^2 - 3.564x_1^3\end{bmatrix} + \begin{bmatrix}-0.215\\ 0\\ -20.967\end{bmatrix}m$$

where μ=tail deflection

x_1 = angle of attack, x_2 = pitch angle, x_3 = pitch rate,

- Tail deflection limited to ±25°
- The angle of stall is encountered at 23.5°

Compared Solutions

- The angle of stall is encountered at 23.5°
- Garrard and Jordan discuss three control laws
  - LQR solution can control the aircraft with an initial angle of attack up to 25.69°.

$$m = -0.053x_1 + 0.5x_2 + 0.521x_3$$

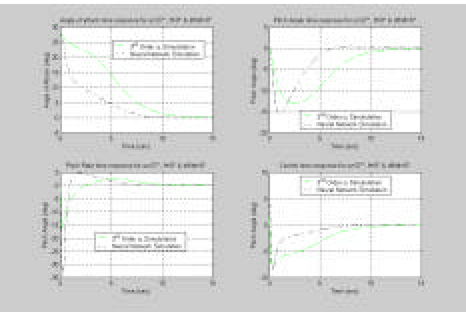
  - Second order can control the aircraft with an initial angle of attack up to 25.99°.

$$m_{2^{nd}} = -0.053x_1 + 0.5x_2 + 0.521x_3 + 0.04x_1^2 - 0.048x_1x_2$$

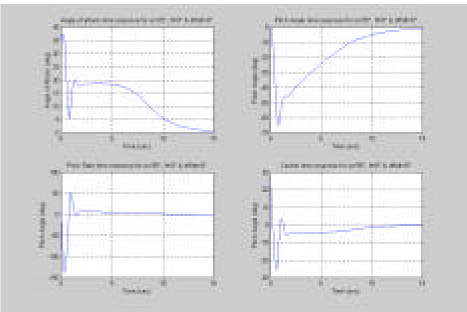
  - Third order can control the aircraft with an initial angle of attack up to 27°.

$$m_{3^{rd}} = -0.053x_1 + 0.5x_2 + 0.521x_3 + 0.04x_1^2 - 0.048x_1x_2 + 0.374x_1^3 - 0.312x_1^2x_2$$

Result: Third Order limit with (α=27º)



Result: Neural Network Limit (α=35º)



Missile Autopilot

Zhongwu Huang

Robust Missile Autopilot Design

$$\dot{a} = \boldsymbol{q} - (\boldsymbol{K}_1 + \boldsymbol{K}_2 a^2)a$$

$$\dot{\boldsymbol{q}} = (\boldsymbol{C}_1 + \boldsymbol{C}_2 a^2)a + (\boldsymbol{C}_3 + \boldsymbol{C}_4 a^2)\overline{\boldsymbol{u}}$$

$$\dot{\overline{\boldsymbol{u}}} = -(\mathbf{1}/t)(\overline{\boldsymbol{u}} - \boldsymbol{u})$$

a : angle of attack q : pitch rate u : fin position

Where t =0.01, K_1 =1.02, K_2 =1.28, C_1 =-57.17, C_2 =-322.16,
C_3 =-70.12, and C_4 =-360.27

Unmodeled Dynamics

$$\dot{x}_1 = \boldsymbol{f}_1(x_1, x_2) + \boldsymbol{d}_{11}(x_1, x_2) + \boldsymbol{d}_{12}$$

$$\dot{x}_2 = \boldsymbol{f}_2(x_1, x_2) + \boldsymbol{g}_2(x_1, x_2)(\boldsymbol{I} + \mathrm{D}(x_1, x_2))\boldsymbol{u}_{opt} + \boldsymbol{d}_{21}(x_1, x_2) + \boldsymbol{d}_{22}$$

Where: $\boldsymbol{d}_{11}(x_1, x_2), \boldsymbol{d}_{12}$, $\boldsymbol{d}_{21}(x_1, x_2), \boldsymbol{d}_{22}$: Unmodeled Dynamics
$\|\boldsymbol{d}_{12}\| \pounds \boldsymbol{d}_{1N}$, $\|\boldsymbol{d}_{22}\| \pounds \boldsymbol{d}_{2N}$.

$\mathrm{D}(x_1, x_2)$: Unmodeled input dynamics
$\|\mathrm{D}(x_1, x_2)\| \pounds \mathbf{1} - e_g$ with $\mathbf{0} < e_g \pounds \mathbf{1}$

Extra control u_e

- Objective: make states bounded around the desired trajectory (practical stability) with an extra control.
- Adding extra-control

$$\dot{x}_2 = \boldsymbol{f}_2(x_1, x_2) + \boldsymbol{g}_2(x_1, x_2)(\boldsymbol{I} + \mathrm{D}(x_1, x_2))(\boldsymbol{u}_{opt} + \boldsymbol{u}_e) + \boldsymbol{d}_{21}(x_1, x_2) + \boldsymbol{d}_{22}$$

Where $\boldsymbol{u}_{opt}$ is the optimal control for the nominal system
(without considering unmodeled dynamics).

Designing u$_e$

- Choosing

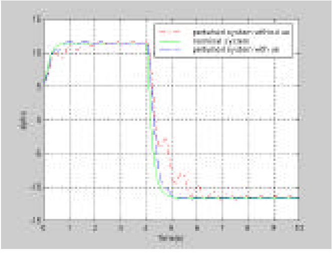
$$u_e = \frac{g_2^T e_2}{\left\| g_2^T e_2 \right\|^2}(-e_2{}^T K_2 e_2 - e_2{}^T \widehat{F}(x_1, x_2) - e_1{}^T K_1 e_1)$$

Where: $\widehat{F}(x_1, x_2)$: Output of a NN with $x_1, x_2, x_{1d}, x_{2d}, e_1, e_2$ as inputs.

$-e_1^T K_1 e_1$, $-e_2^T K_2 e_2$: Stablizing parts helping the initial convergence

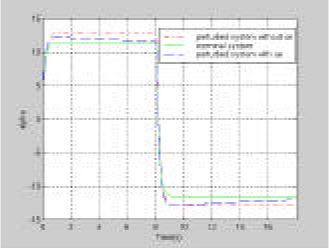
Simulation Results

- Input Uncertainty (t is changed from 0.01 to 0.5)



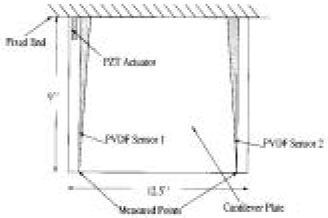
Simulation Results

- Unmodeled Uncertainty ($d = 0.5 K_1 a$ in a equation)

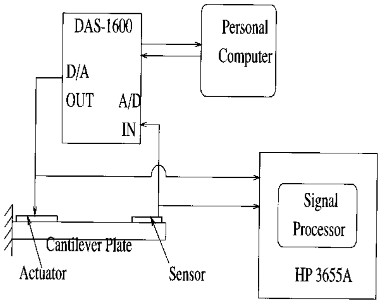


Vibration Suppression of A Cantilevered Plate

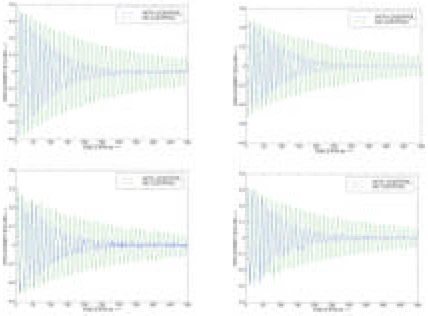
Abhishek Gupta



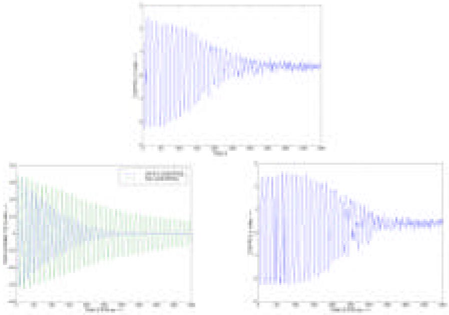
Signal Flow of the Implementation Se for the Plate System



Response of the Plate System for Initial Conditions inside the Training Set



Response of the Plate System for Initial Conditions outside the Training Set



1

$$\frac{\partial T(t,x)}{\partial t} = a(x)\frac{\partial^2 T(t,x)}{\partial x^2} + b(x)u(t,x)$$

$T(t,x)$: Transient temperature profile over the domain.

$a(x)$: Thermal diffusivity

$b(x)$: Heat input / heat generated by means other than thermal conduction

Neumann Boundary Conditions: $\left.\frac{\partial T(t,x)}{\partial x}\right|_{x=x_0,x_f} = 0$

$$J = \frac{1}{2}\left[\int_0^{t_f\to\infty}\int_{x_0}^{x_f}\left(QT^2 + Ru^2\right)dxdt\right]$$

Optimal Control Problem (Discrete)

- System Dynamics

$$x_{k+1,j} = x_{k,j} + \Delta t\left[a_j\left(x_{k,j+1} - 2x_{k,j} + x_{k,j-1}\right)/\Delta y^2 + b_j u_{k,j}\right]$$

- Cost Function

$$J = \frac{1}{2}\left[\sum_{k=1}^{N-1}\sum_{j=1}^{M}\left(Q_D x_{k,j}^2 + R_D u_{k,j}^2\right)\right]$$

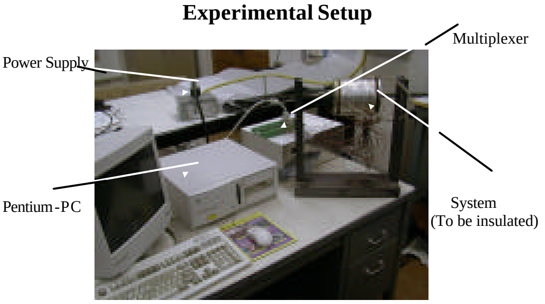
- Costate Equation

$$l_{k,j} = l_{k+1,j} + \Delta t\left[a_j\begin{pmatrix} l_{k+1,j+1} - 2l_{k+1,j} \\ + l_{k+1,j-1} \end{pmatrix}/\Delta y^2 + Q_D x_{k,j}\right]$$

- Optimal Control Equation

$$u_{k,j}^* = -R_D^{-1} b_j l_{k+1,j}$$

Experimental Setup



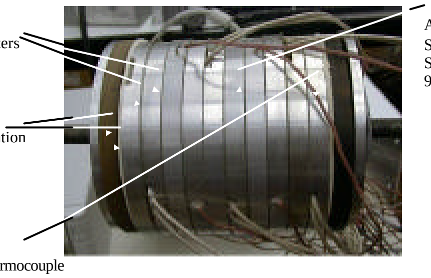
Power Supply

Multiplexer

Pentium-PC

System
(To be insulated)

Experimental Setup



Heaters

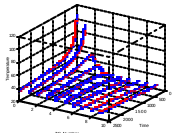
Aluminum
Slabs:
Skewed by
90°

Insulation

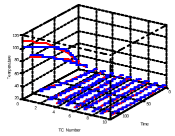
Thermocouple

Modeling (Heater-1)

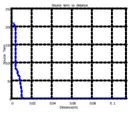
Tool : Tri Diagonal Matrix
Algorithm (TDMA)



$a = 8.5\times10^{-7} m^2/\mathrm{sec}$

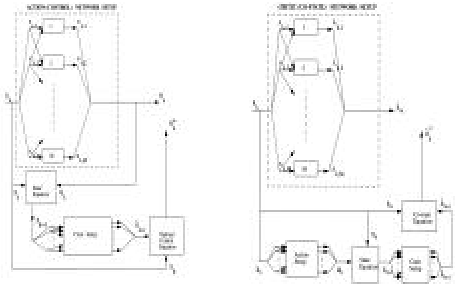


$b(x)$

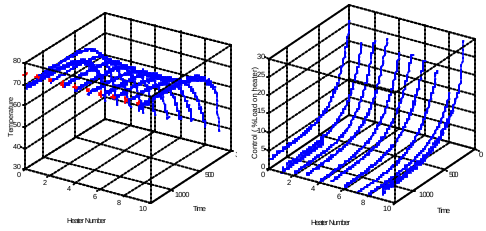


Distance(m)

Action-Critic Synthesis



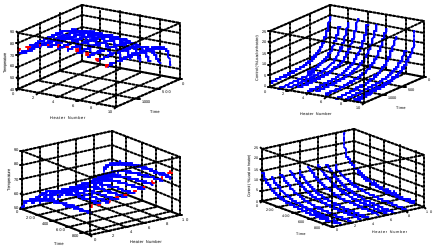
Experimental Results



Temperature History

Control History

Experimental Results



Temperature History

Control History

Conclusions

- Classes of problems solved with adaptive-critic approach so far:
  - Linear Systems
    - Finite-Time
    - Infinite-Time
  - Nonlinear Systems
    - Finite-Time
    - Infinite-Time

Conclusions (continued)

- Robust Control
  - Unmodeled Dynamics
  - Input Uncertainty
- Distributed Parameter Systems
  - Parabolic
  - Hyperbolic
- Systems Driven by Stochastic Inputs
  - Inventory Control

Problems Under Study And Unsolved Mysteries

- A fairly general approach to control of systems driven by stochastic inputs
- Network congestion
- Problems with control saturation and failed actuators
  - X-33, X-34 ascent trajectories (NASA MARSHALL)
  - X-33, X-34 landing guidance (NASA MARSHALL)
  - Airplane flights (NASA AMES)

Learning Control of a Class of Failure Avoidance Problems with Known Analytically Derived Cost Function

Derong Liu
E-mail: dliu@ece.uic.edu

Abstract

We apply adaptive critic designs to a class of failure avoidance control problems. We categorize such problems by the choice of local cost function as zero throughout a trial except at the last time step when a failure occurs. We will derive an analytical form of its overall cost function which is defined as the infinite summation of the local cost function over time. We will demonstrate that the outputs of the critic network after learning resemble well the analytically derived overall cost function.

Learning Control of Failure Avoidance Problems with Known Analytically Derived Cost Function

Derong Liu
Department of Electrical and Computer Engineering
University of Illinois, Chicago, IL 60607, U. S. A.

Abstract–We study adaptive critic designs for a class of failure avoidance control problems. We categorize such problems by the choice of local cost function as zero throughout a trial except at the last time step when a failure occurs. We derive an analytical form of its overall cost function defined as the infinite summation of the local cost function over time. We demonstrate that the outputs of the critic network after learning resemble well the analytically derived cost function.

I. INTRODUCTION

Adaptive critic designs (ACDs) [3], [7], [8], [9], [13], [15]–[21], [24]–[27] are defined as designs that *approximate dynamic programming in the general case*, i.e., approximate optimal control over time in nonlinear environments. There are many problems in practice which can be formulated as cost maximization or minimization problems. Examples include error minimization, energy minimization, profit maximization, and the like. Dynamic programming is a very useful tool in solving these problems. However, it is often computationally untenable to run dynamic programming due to the backward numerical process required for its solutions, i.e., due to the "curse of dimensionality" [6], [14]. Over the years, progress has been made to circumvent the "curse of dimensionality" by building a system, called "critic," to approximate the cost function in dynamic programming (cf. [3], [16], [18], [21], [26], [27]).

A typical ACD consists of three modules that can be implemented by using neural networks. These three modules provide functions of decision, prediction, and evaluation, respectively. When in ACDs the critic network (i.e., the evaluation module) takes the action/control signal as part of its inputs, the designs are referred to as action dependent ACDs (ADACDs). Researchers have shown the close relationship between ACDs and reinforcement learning (cf. [4], [26], [27]). One of the major differences is the way in which the cost function is represented. In ACDs, the cost function is approximated using networks which are *built up from differentiable functions* such as *neural networks*; while in reinforcement learning methods, the cost/value function is usually stored in *look-up tables* (which implies a finite or discrete set of states) [23].

The present paper is organized as follows. In Section II, we will introduce some necessary background materials for adaptive critic designs. In Section III, we describe a class of failure avoidance problems studied in the present paper. In Section IV, we will derive the analytical form of the cost function and discuss two approaches for the training of neural networks used in the present ACDs.

II. ACTION-DEPENDENT HEURISTIC DYNAMIC PROGRAMMING

Suppose that one is given a discrete-time nonlinear dynamical system

$$x(t+1) = F[x(t), u(t), t]$$

where $x \in R^n$ represents the state vector of the system and $u \in R^m$ denotes the control action. Suppose that one associates with this system the performance index (or cost)

$$J[x(i), i] = \sum_{k=i}^{\infty} \gamma^{k-i} U[x(k), u(k), k] \tag{1}$$

where U is called the utility function or local cost function and γ is the discount factor with $0 < \gamma \leq 1$. Note that J is dependent on the initial time i and the initial state $x(i)$, and it is referred to as the cost-to-go of state $x(i)$. The objective is to choose the control sequence $u(k)$, $k = i, i+1, \cdots$, so that the function J (i.e., the cost) in (1) is *minimized*.

Consider the model-free ADHDP shown in Figure 1 (cf. [15]). The critic network in this case will be trained by minimizing the following error measure over time,

$$\begin{aligned} \|E_q\| &= \sum_t E_q^2(t) \\ &= \sum_t \left[Q(t-1) - U(t) - \gamma Q(t)\right]^2 \end{aligned} \tag{2}$$

where $Q(t) = Q[x(t), u(t), t, W_C]$. When $E_q(t) = 0$ for all t, (2) implies that

$$\begin{aligned} Q(t-1) &= U(t) + \gamma Q(t) \\ &= U(t) + \gamma[U(t+1) + \gamma Q(t+1)] \\ &= \cdots \\ &= \sum_{k=t}^{\infty} \gamma^{k-t} U(k). \end{aligned} \tag{3}$$

This work was supported by the National Science Foundation under Grant ECS-9996428. Email: dliu@ieee.org.

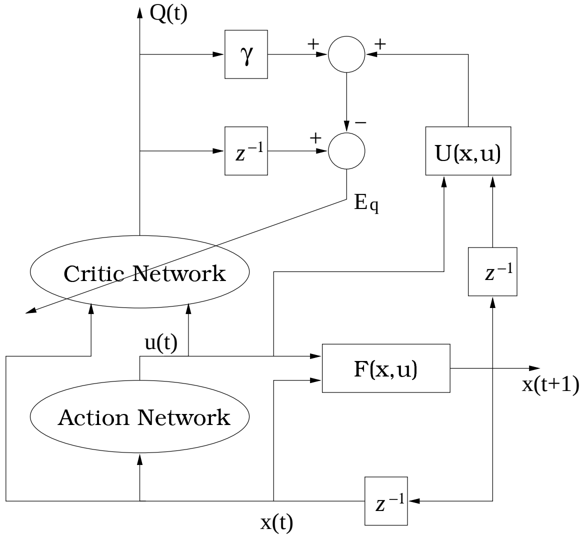


Fig. 1. A typical scheme of an action-dependent heuristic dynamic programming [15].

Comparing to (1), we can see that when minimizing the error function in (2), we have a neural network trained so that its output becomes an estimate of the cost function defined in dynamic programming for $i = t+1$, i.e., the value of the cost function in the immediate future [15].

The input-output relationship of the critic network in Figure 1 is given by

$$Q(t) = Q\left[x(t), u(t), t, W_C^{(p)}\right]$$

where $W_C^{(p)}$ represents the weights of the critic network after the pth weight update. There are two methods to train the critic network according to the error function (2) in the present case which are described next [15].

(1) *Backward-in-time:* We can train the critic network at time t, with the output target given by $[Q(t-1)-U(t)]/\gamma$. The training of the critic network is to realize the mapping given by

$$C_b\colon \left\{ \begin{array}{c} x(t) \\ u(t) \end{array} \right\} \to \left\{ \frac{1}{\gamma}\left[Q(t-1) - U(t)\right] \right\}. \qquad (4)$$

In this case, we consider $Q(t)$ in (2) as the output from the *network to be trained* and the target output value for the critic network is calculated using its output at time $t-1$. Thus, we refer to this method as the *backward-in-time* method.

(2) *Forward-in-time:* We can train the critic network at time $t-1$, with the output target given by $U(t) + \gamma Q(t)$. The training of the critic network is to realize the mapping given by

$$C_f\colon \left\{ \begin{array}{c} x(t-1) \\ u(t-1) \end{array} \right\} \to \{U(t) + \gamma Q(t)\}. \qquad (5)$$

In this case, we consider $Q(t-1)$ in (2) as the output from the *network to be trained* and the target output value for the critic network is calculated using its output at time t. Thus, we refer to this method as the *forward-in-time* method.

After the critic network's training is finished, the action network's training starts with the objective of minimizing $Q(t)$. The goal of the action network training is to minimize the critic network output $Q(t)$. In this case, we can choose the target of the action network training as zero, i.e., we will train the action network so that the output of the critic network becomes as small as possible. The desired mapping which will be used for the training of the action network in the present ADHDP is given by

$$A\colon \{x(t)\} \to \{0(t)\} \qquad (6)$$

where $0(t)$ indicates the target values of zero. We note that during the training of action network, it will be connected to the critic network as shown in Figure 1. The target in (6) is for the output of the *whole network*, i.e., the output of *the critic network after it is connected to the action network* as shown in Figure 1.

III. FAILURE AVOIDANCE CONTROL PROBLEMS

There are many problems in practice that have an objective of avoiding failures. The well-known example in this class of problems is the cart-pole problem. In the cart-pole problem, the objective is to balance an upright pole, whose bottom is attached by a pivot to a cart that travels along a track. The state of this system is given by the pole's angle and angular velocity and the cart's horizontal position and velocity. The only available control actions are to exert forces of fixed magnitude on the cart that push it to the left or right. The event of the pole falling past a certain angle or the cart running into the bounds of its track is called a *failure*. A sequence of forces must be applied so that failures can be avoided by balancing the pole within the bounds of the track. A naive, randomly initialized controller, before learning much about the task, will rarely be able to avoid failures. The cart-pole system is reset to its initial state after each failure and the controller must learn to balance the pole for as long as possible. The cart-pole problem has been used often in the literature as a benchmark problem for testing learning control algorithms [1], [5], [13], [15], [20], [21].

We will consider failure avoidance problems with the function U in ACDs given by

$$U(t) = \left\{ \begin{array}{ll} 0, & \text{before failure happens} \\ 1, & \text{when failure occurs.} \end{array} \right. \qquad (7)$$

For the illustrating example considered in this paper, i.e., for the cart-pole problem [2], [5], the function U is defined

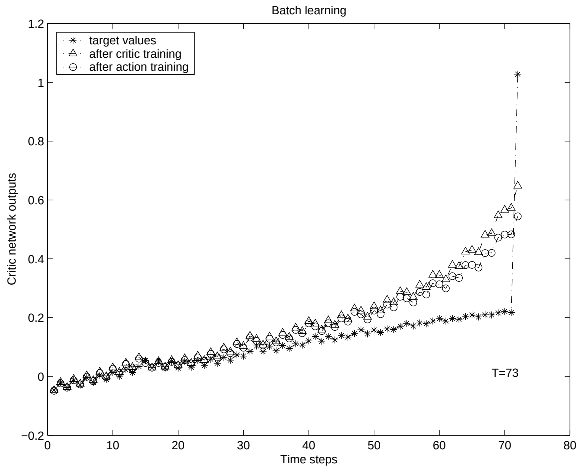


Fig. 2. The values of the critic network output in batch learning.

as

$$U(t) = \begin{cases} 0, & \text{if } |\theta(t)| < 12^\circ \text{ and } |x(t)| < 2.4\,\text{m} \\ 1, & \text{otherwise} \end{cases} \quad (8)$$

where $\theta(t)$ is the angle of the pole and $x(t)$ is the horizontal position of the cart on the track. Note that the definition given in (7) and (8) are for each time step t in a test/trial. The function U defined this way implies that a "reinforcement" signal [i.e., $U(t) \neq 0$] is only available when a failure occurs. In the cart-pole problem, the "reinforcement" signal is only available when the pole falls or when the cart runs into the bounds of its track [2], [5]. Failure avoidance is indicated by minimizing the sum of the function U over time given by

$$\sum_{k=0}^{\infty} \gamma^k U(k). \quad (9)$$

Assume that the pole falls when $t = T$. We will derive an analytical expression for the summation in (9) in the next section to show that it is a function of T. For the cart-pole problem, minimizing the summation in (9) is equivalent to maximizing T, which in turn implies keeping the pole from falling (and the cart from running into the bounds of its track) for as long as possible.

IV. ADAPTIVE CRITIC DESIGNS WITH ANALYTICALLY DERIVED COST FUNCTION

In the following, we describe two approaches for neural network learning in the present ADHDP. We will describe two batch learning approaches; one of them uses the target function given in (4) or (5) and the other uses an analytically derived cost function as the target function for the critic network training.

A. Batch Learning

For the cart-pole problem, the critic network is chosen as a 5–7–1 structure with 5 input neurons and 7 hidden layer neurons. The 5 inputs are the 4 states $\theta(t)$, $\dot{\theta}(t)$, $x(t)$, and $\dot{x}(t)$, and the output of the action network $u(t)$. The hidden layer uses the sigmoidal function given by

$$y = \frac{1 - e^{-x}}{1 + e^{-x}},$$

i.e., the `tansig` function in Matlab [10], and the output layer uses the linear function `purelin`. Utilizing the Matlab Neural Network Toolbox [10], we have applied `traingd` (the simple gradient descent algorithm) and `trainlm` (the Levenberg-Marquardt algorithm [11]) for the the training of neural networks in the present work. We use $\gamma = 0.9$ in the present experiments for the cart-pole problem. The structure of the action network is chosen as 4–6–1 with 4 input neurons and 6 hidden layer neurons. The 4 inputs are $\theta(t)$, $\dot{\theta}(t)$, $x(t)$, and $\dot{x}(t)$. Both the hidden layer and the output layer use the sigmoidal function `tansig`.

Training data will be collected as in (4) or (5) for the critic network and as in (6) for the action network. At the end of each (failed) trial, we start the critic network training. Results from a typical trial are illustrated in Figure 2. In this case, the critic network target are close to zero before the time $t = T$ ($T = 73$ in Figure 2) when the pole falls, and it becomes close to 1 when $t = T$. Due to the sudden change in the critic network target values from 0 to 1, the calculated target function to be learned is not smooth anymore. It is known that to learn a non-smooth function may require a very large network [12], [22]. The critic network outputs after the critic network training become a compromise between the target value at $t = T$ and those preceding it. We can see from Figure 2 that points that are close to the last sample at $t = T$ have their outputs lifted up (from the target values) and the last training sample is learned with some error. In this case, we typically have only one training sample that indicates a target close to 1 (i.e., the last pair of the training data) and we have the rest of training data samples indicating target values close to 0. When all these data samples are learned together in a batch mode, it is clear that the last training sample will be very difficult to learn due to its sudden increase/change from 0 to 1 in the target values corresponding to the environment (state) that may not be far away from those points preceding it in time.

The action network training can also be trained similarly in a batch mode. For the same example, the critic network output after the action network training using the batch mode is shown in Figure 2. We see that the training of the action network to minimize the outputs of the critic network will typically lower the surface of the critic network output. The closer the point is to the last point at $t = T$, the more reduction to the critic network output value will usually be.

We note that results shown in Figure 2 are very typical in the case of batch learning, i.e., most of the trials achieve similar training results to those shown in Figure 2. Also, we point out that the results shown in Figure 2 for outputs of the critic network after its training is a good approximation to the analytically derived cost function to be introduced next.

B. Learning With Analytically Derived Cost Function

In this subsection, we first derive the analytical form of the actual function $J(t)$ that is true for *all* failure avoidance problems with the function $U(t)$ defined in (7). We will then show some results in comparison with the approach introduced in the previous subsection.

Assume that the task of balancing the pole in a cart-pole system is on the infinite time horizon, i.e., each trial or test is consider for time $t \in [0, \infty)$. In this case, according to dynamic programming, the function $J(t)$ is given by

$$J(t) = \sum_{k=t}^{\infty} \gamma^{k-t} U(k). \tag{10}$$

Assume that the pole falls at time $t = T$. From (7), the function $U(t)$ in this case can be expressed as

$$U(t) = \begin{cases} 0, & \text{when } t < T \\ 1, & \text{when } t \geq T. \end{cases} \tag{11}$$

Combining (10) and (11), we obtain

$$J(t) = \begin{cases} \gamma^{T-t}/(1-\gamma), & t < T \\ 1/(1-\gamma), & t \geq T. \end{cases} \tag{12}$$

From (12), we can see that minimizing the cost function $J(t)$ is equivalent to maximizing the time T. A successful balancing of the pole is indicated by $T = \infty$ which corresponds to the minimum possible value that the cost function $J(t)$ in (12) can attain.

Our next approach for the training of neural networks in the present ADHDP is to train the critic network after each trial (i.e., in a batch mode) with the function $J(t)$ given by (12) as the target. The function in (12) depends on how long the pole is balanced (T) before failure happens and the discount factor γ. The parameter γ determines the shape/curve of the function $J(t)$ for $t \leq T$ and T determines the point after which the function becomes flat (constant). The action network can be trained as in the previous subsection by minimizing the output of the critic network after it is trained. A plot of the function $J(t)$ in (12) and the results of the critic network training are shown in Figure 3. We have scaled the function in (12) by $(1 - \gamma)$ in our computer programs. We note that this scaling is equivalent to replacing the value of $U(t)$ for $t \geq T$ by $1 - \gamma$. It is also equivalent to using a scaled neural network as is done extensively in [17]. In the present study, when using a normalized/scaled neural network, we have all the input and output signals of the network in the range from -1 to $+1$. It is a convenient choice when using Matlab that we employ scaled neural networks in the present implementations. Clearly, the results from the trained critic networks in Figure 2 resemble well the plot of the true, analytically derived $J(t)$ in Figure 3 (for $t \leq T$). That is to say, even though there are discrepancies in the learning described in the previous subsection, what we achieve after the critic network training actually approximates well the analytically derived cost function.

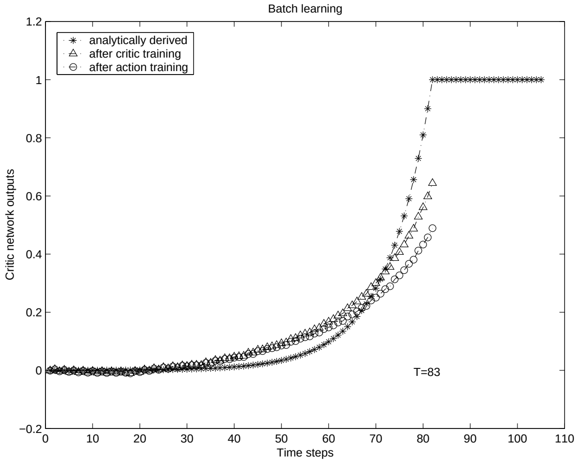


Fig. 3. The critic network learning with the analytically derived function J.

Results illustrated in Figure 3 also show discrepancies between the target values of the critic network outputs determined by (12) and the output values after the critic network training. We note that these discrepancies are caused by the small network size and sometimes insufficient number of iterations in training. For the cart-pole problem illustrated here, if we choose a larger network size, e.g., use 14 hidden layer neurons instead of 7 (used in Figure 3) for the critic network, we will be able to train the critic network using sufficiently large number of iterations so that its output values after training match exactly with the analytically determined values in most trials.

Finally, we emphasize that the analytical form of the cost function $J(t)$ derived in (12) is true for the whole class of failure avoidance control problems considered in the present paper with the local cost function $U(t)$ defined as in (7).

C. Comparison of the Two Learning Approaches

To compare the two learning approaches described in the present paper, in each experiment, we start with exactly the same initial weights for the critic network and with exactly the same initial weights for the action network. We repeat the experiments many times with different initial weights for each experiment. Our goal is to compare the performance of the two approaches in terms of the number of trials that is needed to balance the pole. It turns out after many experiments that the learning in batch mode with calculated critic network output target values as in (4) or (5) and with analytically derived cost function as in (12) perform very close to each other, i.e., they usually require similar number of trials to balance the pole. A more careful examination also reveals that the learning in batch mode with analytically derived cost function is more gradual than the learning in batch mode with calculated target values for the critic network. Using the former, the number of steps that the pole is balanced in a trial increases gradually most of the time from one trial to the next until the pole is balanced. Using the latter, the number of steps that the pole is balanced goes up and down from one trial to the next with the overall upward trend.

We can therefore conclude that both approaches presented in the present paper work well for the neural network learning in the present ADHDP. We can also conclude that for the critic network learning batch learning approach with calculated target functions resemble well the learning approach with analytically derived cost function. One implication of the present results is that the inability of learning exactly the calculated target values is often useful in critic network learning. The discrepancies between the calculated target values and the output values after the critic network learning actually indicate that the states near the end of a failed trial are also not good and should be avoided in future trials.

References

[1] C. W. Anderson, "Learning to control an inverted pendulum using neural networks," *IEEE Control Systems Magazine*, vol.9, pp.31–37, Apr. 1989.

[2] C. W. Anderson and W. T. Miller III, "Challenging control problems," in *Neural Networks for Control* (Appendix A), Edited by W. T. Miller III, R. S. Sutton, and P. J. Werbos, Cambridge, MA: The MIT Press, 1990.

[3] S. N. Balakrishnan and V. Biega, "Adaptive-critic-based neural networks for aircraft optimal control," *Journal of Guidance, Control, and Dynamics*, vol.19, pp.893–898, July-Aug. 1996.

[4] A. G. Barto, "Reinforcement learning and adaptive critic methods," in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (Chapter 12), Edited by D. A. White and D. A. Sofge, New York, NY: Van Nostrand Reinhold, 1992.

[5] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol.SMC-13, pp.834–846, Sept./Oct. 1983.

[6] R. E. Bellman, *Dynamic Programming*, Princeton, NJ: Princeton University Press, 1957.

[7] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Belmont, MA: Athena Scientific, 1996.

[8] C. Cox, S. Stepniewski, C. Jorgensen, R. Saeks, and C. Lewis, "On the design of a neural network autolander," *International Journal of Robust and Nonlinear Control*, vol.9, pp.1071–1096, Dec. 1999.

[9] J. Dalton and S. N. Balakrishnan, "A neighboring optimal adaptive critic for missile guidance," *Mathematical and Computer Modeling*, vol.23, pp.175–188, Jan. 1996.

[10] H. Demuth and M. Beale, *Neural Network Toolbox User's Guide*, Natick, MA: MathWorks, 1998 (Version 3).

[11] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol.5, pp.989–993, Nov. 1994.

[12] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Upper Saddle River, NJ: Prentice Hall, 1999.

[13] G. G. Lendaris and C. Paintz, "Training strategies for critic and action neural networks in dual heuristic programming method," *Proceedings of the 1997 IEEE International Conference on Neural Networks*, Houston, TX, June 1997, pp.712–717.

[14] F. L. Lewis and V. L. Syrmos, *Optimal Control*, New York, NY: John Wiley, 1995.

[15] D. Liu, X. Xiong, and Y. Zhang, "Action-dependent adaptive critic designs," *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, Washington, DC, July 2001, pp.990–995.

[16] D. V. Prokhorov, *Adaptive Critic Designs and Their Applications*, Ph.D. Dissertation, Texas Tech University, Lubbock, TX, Oct. 1997.

[17] D. V. Prokhorov, R. A. Santiago, and D. C. Wunsch, "Adaptive critic designs: A case study for neurocontrol," *Neural Networks*, vol.8, pp.1367–1372, 1995.

[18] D. V. Prokhorov and D. C. Wunsch, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. 8, pp. 997–1007, Sept. 1997.

[19] R. E. Saeks, C. J. Cox, K. Mathia, and A. J. Maren, "Asymptotic dynamic programming: Preliminary concepts and results," *Proceedings of the 1997 IEEE International Conference on Neural Networks*, Houston, TX, June 1997, pp.2273–2278.

[20] R. A. Santiago and P. J. Werbos, "New progress towards truly brain-like intelligent control," *Proceedings of the World Congress on Neural Networks*, San Diego, CA, June 1994, vol. I, pp.27–33.

[21] J. Si and Y.-T. Wang, "On-line learning control by association and reinforcement," *IEEE Transactions on Neural Networks*, vol.12, pp.264–276, Mar. 2001.

[22] E. D. Sontag, "Feedforward nets for interpolation and classification," *Journal of Computer and System Sciences*, vol.45, pp.20–48, 1992.

[23] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol.3, pp.9–44, 1988.

[24] P. J. Werbos, "Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research," *IEEE Transactions on Systems, Man, and Cybernetics*, vol.SMC-17, pp.7–20, Jan./Feb. 1987.

[25] P. J. Werbos, "Consistency of HDP applied to a simple reinforcement learning problem," *Neural Networks*, vol. 3, pp. 179–189, 1990.

[26] P. J. Werbos, "A menu of designs for reinforcement learning over time," in *Neural Networks for Control* (Chapter 3), Edited by W. T. Miller, R. S. Sutton, and P. J. Werbos, Cambridge, MA: The MIT Press, 1990.

[27] P. J. Werbos, "Approximate dynamic programming for real-time control and neural modeling," in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (Chapter 13), Edited by D. A. White and D. A. Sofge, New York, NY: Van Nostrand Reinhold, 1992.

Excitation and Turbine Neurocontrol with Derivative Adaptive Critics of Multiple Generators on the Power Grid

G.K.Venayagamoorthy, R.G.Harley and D.C.Wunch

Abstract

Power Systems containing turbogenerators are large scale nonlinear systems. The conventional controllers for the turbogenerators are designed by linear control theory based on a single-machine-infinite-bus (SMIB) power system model. These SMIB power system mathematical models are linearized at specific operating points and then the conventional controllers are designed. The machine parameters change with loading in a complex manner, resulting in different behavior at different operating points and the controller which stabilizes the system under specific operating conditions, may no longer yield satisfactory results when there is a drastic change in the power system operating conditions and configurations. Adaptive critic designs combining concepts of reinforcement learning and approximate dynamic programming, are neural network designs capable of optimization over time under conditions of noise and uncertainty. The design and implementation of adaptive critic based neurocontrollers that replace the conventional automatic voltage regulators (AVRs) and turbine governors will be presented. The feedback variables to the neurocontroller are completely based on local measurements from the turbogenerator. Simulation and experimental results will be presented to show the superior performance of adaptive critic based neurocontroller compared to the conventional AVR and turbine governor controllers.

Excitation and Turbine Neurocontrol with Derivative Adaptive Critics of Multiple Generators on the Power Grid

[*][†]G.K.Venayagamoorthy
Member, IEEE
Department of Electronic Engineering,
ML Sultan Technikon, POBox 1334
Durban 4000 South Africa
gkumar@ieee.org

[*]R.G.Harley *Fellow, IEEE*
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta GA 30332-0250 USA
ron.harley@ee.gatech.edu

D.C.Wunsch *Senior Member, IEEE*
[†]*Applied Computational Intelligence Laboratory*
University of Missouri-Rolla
MO 65409-0249 USA
dwunsch@ece.umr.edu

[*]*Electrical Engineering Department University of Natal Durban 4041 South Africa*

Abstract

Based on derivative adaptive critics, neurocontrollers for excitation and turbine control of multiple generators on the electric power grid are presented. The feedback variables are completely based on local measurements. Simulations on a three-machine power system demonstrate that the neurocontrollers are much more effective than conventional PID controllers, the automatic voltage regulators and the governors, for improving dynamic performance and stability under small and large disturbances.

1 Introduction

Power systems containing turbogenerators are large-scale nonlinear systems. The conventional controllers for the generators are designed by linear control theory based on a single-machine infinite bus (SMIB) power system model. These SMIB power system models are linearized at specific operating points, and then excitation and turbine controllers are designed, based on the linearized models. The drawback of this approach is that once the operating point or the system configuration changes, the performance of the controller degrades. Conservative designs are therefore used, particularly in multimachine systems, to attempt satisfactory control over the entire operating range of the power system.

In recent years, renewed interest has been shown in power systems control using nonlinear control theory, particularly to improve system transient stability [1,2]. Instead of using an approximate linear model, as in the design of the conventional power system stabilizer, nonlinear models are used and nonlinear feedback linearization techniques are employed on the power system models, thereby alleviating the operating point dependent nature of the linear designs. Nonlinear controllers significantly improve the power system's transient stability. However, nonlinear controllers have a more complicated structure and are difficult to implement relative to linear controllers. In addition, feedback linearization methods require exact system parameters to cancel the inherent system nonlinearities, and this contributes further to the complexity of stability analysis. The design of decentralized linear controllers to enhance the stability of interconnected nonlinear power systems within the whole operating region is still a challenging task [3]. However, the use of Artificial Neural Networks offers a possibility to overcome this problem.

Multilayer perceptron type artificial neural networks (ANNs) are able to identify/ model time varying single turbogenerator systems [4] and, with continually online training, these models can track the dynamics of the power system, thus yielding adaptive identification. ANN controllers have been successfully implemented on single turbogenerators using ANN identifiers and indirect feedback control [5-6]. Moreover, ANN identification of turbogenerators in a multi-machine power system has also been reported [7].

In this paper, the electric power grid is modeled using artificial neural networks and used in the development of neurocontrollers based on derivative adaptive critics, to replace the conventional automatic voltage regulators (AVRs) and turbine governors. With derivative adaptive critics, optimal neurocontrollers can be designed by using pre-recorded data from the power system operation, and offline training, before allowing the neural network to control the generators. With adaptive critics, the computational load of online training is therefore avoided. The method presented in this paper can therefore be used in the development of neurocontrollers to be retrofitted to existing plants.

A three-machine laboratory power system example is simulated, with neurocontrollers on two generators. The third generator is the infinite bus, with a fixed voltage and frequency. The simulation results show that both voltage regulation and system stability enhancement can be achieved with this proposed neurocontroller, regardless of the system operating conditions and types of disturbances.

2 Electric Power Grid

The multi-machine laboratory power system in figure 1 is modeled in the MATLAB/SIMULINK environment using the Power System Blockset (PSB) [8]. Each machine is represented by a seventh order model. There are three coils on the d-axis and two coils on the q-axis and the stator transient terms are not neglected. A three machine five-bus power system is chosen, to illustrate the effectiveness of the adaptive critic based controllers. The power system in figure 1 consists of two micro-generators.

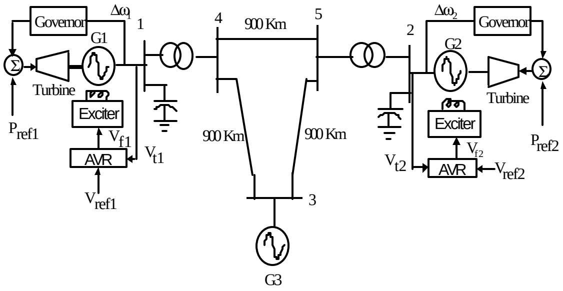


Figure 1: Multimachine Power System Model

Each of the 3 kW, 220 V, three phase micro-generator was designed to have all its per-unit parameters, except the field winding resistance, the same as those normally expected of a 1000 MW generator. The parameters of the micro-generators, determined by the IEEE standards are given in Table 1 [9]. A time constant regulator is used on each micro-generator to insert negative resistance in series with the field winding circuit, in order to reduce the actual field winding resistance to the correct per-unit value.

The conventional AVR and exciter combination transfer function block diagram is similar for both generators and is shown in figure 2 and the time constants are given in Table 2. The exciter saturation factor S_e is given by

$$S_e = 0.6093\exp(0.2165V_{fd}) \qquad (1)$$

T_{v1}, T_{v2}, T_{v3} and T_{v4} are the time constants of the PID voltage regulator compensator; T_{v5} is the input filter time constant; T_e is the exciter time constant; K_{av} is the AVR gain; V_{fdm} is the exciter ceiling; and, V_{ma} and V_{mi} are the AVR maximum and minimum ceilings.

Table 1: Micro-Generator Parameters.

T_{d0}' = 4.50 s	X_d' = 0.205 pu	Rs = 0.006
T_{d0}" = 33 ms	X_d" = 0.164 pu	H = 5.68
T_{q0}" = 0.25 s	X_q = 1.98 pu	F = 0
X_d = 2.09 pu	X_q" = 0.213 pu	p = 2

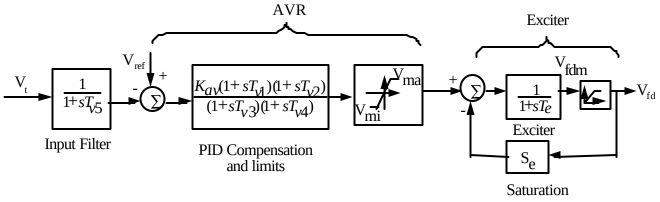


Figure 2: Block Diagram of the AVR and Exciter Combination.

Table 2: AVR and Exciter Time Constants.

T_{v1}	0.616 s	T_{v4}	0.039 s
T_{v2}	2.266 s	T_{v5}	0.0235 s
T_{v3}	0.189 s	T_e	0.47 s

A separately excited 5.6 kW dc motor is used as a prime mover, called the micro-turbine, to drive each of the micro-generators. The torque-speed characteristic of the dc motor is controlled to follow a family of rectangular hyperbola for different positions of the steam valve, as would occur in a real typical high pressure (HP) turbine cylinder. The three low pressure (LP) cylinders' inertia are represented by appropriately scaled flywheels. The micro-turbine and the governor transfer function block diagram is shown in figure 3, where, P_{ref} is the turbine input power set point value, P_m is the turbine output power, and *Äù* is the speed deviation. The turbine and governor time constants are given in Table 3.

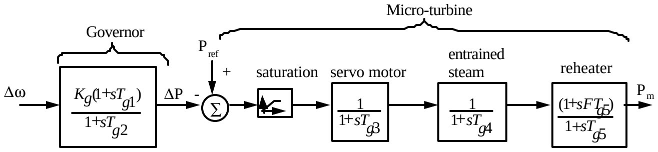


Figure 3: Block Diagram of the Micro-Turbine and Governor Combination.

Table 3: Micro-Turbine and Governor Time Constants

Phase advance compensation, T_{g1}	0.264
Phase advance compensation, T_{g2}	0.0264
Servo time constant, T_{g3}	0.15
Entrained steam delay, T_{g4}	0.594
Steam reheat time constant, T_{g5}	2.662
pu shaft output ahead of reheater, F	0.322

3 Derivative Adaptive Critics' Based Neurocontrollers

Adaptive Critic Designs (ACDs) are neural network designs capable of optimization over time under conditions of noise and uncertainty. A family of ACDs was proposed by Werbos [10] as a new optimization technique combining concepts of reinforcement learning and approximate dynamic programming. For a given series of control actions, that must be taken in sequence, and not knowing the quality of these actions until the end of the sequence, it is impossible to design an optimal controller using traditional supervised learning.

Dynamic programming prescribes a search which tracks backward from the final step, rejecting all suboptimal paths from any given point to the finish, but retains all other possible trajectories in memory until the starting point is reached. However, many paths which may be unimportant, are nevertheless also retained until the search is complete. The result is that the procedure is too computationally demanding for most real problems. In supervised learning, an ANN training algorithm utilizes a desired output and, comparing it to the actual output, generates an error term to allow learning. For an MLP type ANN the backpropagation algorithm is typically used to get the necessary derivatives of the error term with respect to the training parameters and/or the inputs of the network. However, backpropagation can be linked to reinforcement learning via a network called the ***Critic*** network, which has certain desirable attributes.

Critic based methods remove the learning process one step from the control network (traditionally called the "***Action*** network" or "***actor***" in ACD literature), so the desired trajectory or control action information is not necessary. The critic network learns to approximate the cost-to-go or strategic utility function, and uses the output of an action network as one of its inputs directly or indirectly. When the critic network learns, backpropagation of error signals is possible along its input pathway from the action network. To the backpropagation algorithm, this input pathway looks like just another synaptic connection that needs weight adjustment. Thus, no desired signal is needed. All that is required is a desired cost function *J* given in eq. (2).

$$J(t) = \sum_{k=0}^{\infty} \boldsymbol{g}^k U(t+k) \tag{2}$$

where $\boldsymbol{g}$ is a discount factor for finite horizon problems ($0 < \boldsymbol{g} < 1$), and *U(.)* is the utility function or local cost.

The Critic and the Action networks, can be connected together directly (Action-dependent designs) or through an identification model of a plant (Model-dependent designs). There are three classes of implementations of ACDs called Heuristic Dynamic Programming (HDP), Dual Heuristic Programming (DHP), and Globalized Dual Heuristic Dynamic Programming (GDHP), listed in order of increasing complexity and power [11]. This paper presents the DHP model dependent design, and compares its performance against the results obtained using conventional PID controllers.

The critic network is trained forward in time, which is of great importance for real-time operation. DHP has a critic network which estimates the derivatives of *J* with respect to a vector of observables of the plant, ***DY***. The critic network learns minimization of the following error measure over time:

$$\left\| E \right\| = \sum_t E^T(t) E(t) \tag{3}$$

where

$$E(t) = \frac{\P J[\boldsymbol{D}Y(t)]}{\P\boldsymbol{D}Y(t)} - \boldsymbol{g}\frac{\P J[\boldsymbol{D}Y(t+1)]}{\P\boldsymbol{D}Y(t)} - \frac{\P U(t)}{\P\boldsymbol{D}Y(t)} \tag{4}$$

where $\P(.)/\P\boldsymbol{D}Y(t))$ is a vector containing partial derivatives of the scalar (.) with respect to the components of the vector ***DY***. The critic network's training is more complicated than in HDP since there is a need to take into account all relevant pathways of backpropagation as shown in figure 4, where the paths of derivatives and adaptation of the critic are depicted by dashed lines.

In DHP, application of the chain rule for derivatives yields

$$\begin{aligned}\frac{\partial J(t+1)}{\partial \boldsymbol{D}Y_j(t)} &= \sum_{i=1}^{n} \boldsymbol{l}_i(t+1)\frac{\partial \boldsymbol{D}Y_i(t+1)}{\partial \boldsymbol{D}Y_j(t)} \\ &\sum_{k=1}^{m}\sum_{i=1}^{n} \boldsymbol{l}_i(t+1)\frac{\partial \boldsymbol{D}Y_i(t+1)}{\partial A_k(t)}\frac{\partial A_k(t)}{\partial \boldsymbol{D}Y_i(t)}\end{aligned} \tag{5}$$

where $\boldsymbol{l}_i(t+1) = \P J(t+1)/\P\boldsymbol{D}Y_i(t+1))$, and *n*, *m* are the numbers of outputs of the model and the action networks, respectively. By exploiting eq. (5), each of *n* components of the vector *E(t)* from eq. (4) is determined by

$$\begin{aligned}E_j(t) &= \frac{\partial J(t)}{\partial \boldsymbol{D}Y_j(t)} - \boldsymbol{g}\frac{\partial J(t+1)}{\partial \boldsymbol{D}Y_j(t)} - \frac{\partial U(t)}{\partial \boldsymbol{D}Y_j(t)} \\ &- \sum_{k=1}^{m}\frac{\partial U(t)}{\partial A_k(t)}\frac{\partial A_k(t)}{\partial \boldsymbol{D}Y_j(t)}\end{aligned} \tag{6}$$

The action network is adapted in figure 5 by propagating $l(t+1)$ back through the model to the action.

The goal of such adaptation can be expressed as:

$$\frac{\partial U(t)}{\partial A(t)}+g\frac{\partial J(t+1)}{\partial A(t)}=0 \quad \forall\, t \tag{7}$$

The weights' update expression is:

$$DW_A=-a\left[\frac{¶U(t)}{¶A(t)}+g\frac{¶J(t+1)}{¶A(t)}\right]^T\frac{¶A(t)}{¶W_A} \tag{8}$$

where a is a positive learning rate.

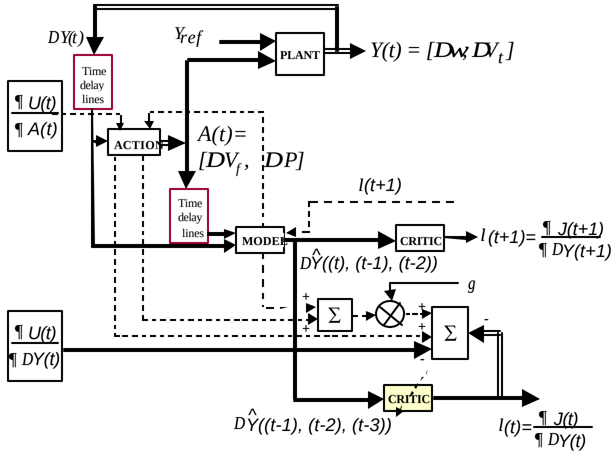


Figure 4: DHP Critic Network Adaptation

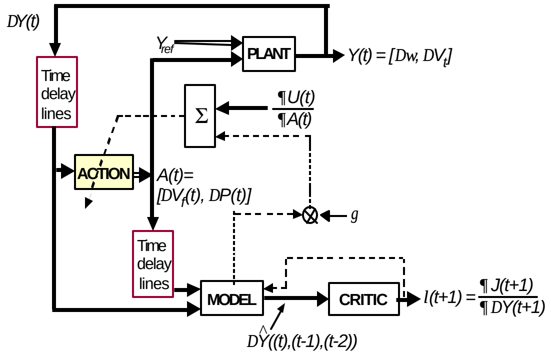


Figure 5: DHP Action Network Adaptation

4 Three Artificial Neural Networks -Model, Critic and Action

Neurocontrollers are designed to replace the AVRs and governors on generator G1 and G2, and therefore ANN models of generator G1 and G2, and the networks to which they are connected are obtained as described in [7]. The ANN model in figures 4 & 5 is a three layer feedforward network with twelve inputs, a single hidden layer of fourteen neurons and two outputs. The inputs to the ANN are the *deviation* of the *actual* power ΔP to its turbine, the *deviation* of the *actual* field voltage DV_f to its exciter, the *deviation* of the *actual* speed Dw, and the *deviation* of the *actual* RMS terminal voltage DV_t of its generator. These four inputs are also delayed by the sample period of 10 ms and, together with eight previously delayed values, form twelve inputs altogether. For this set of inputs, the outputs are the *estimated* speed deviation $\Delta\hat{w}$ and the *estimated* terminal voltage deviation $\Delta\hat{V}_t$, of the generator.

The critic network in figures 4 & 5 is also a three layer feedforward network with six inputs, thirteen hidden neurons and, two outputs. The inputs to the critic network are the speed *deviation* Dw and terminal voltage *deviation* DV_t. These inputs are time delayed by a sample period of 10 ms, and together with the four previously delayed values, form the six inputs for the critic network. The outputs of the critic are the derivatives of the J function with respect to the output states of the generators.

The action network in figures 4 & 5 is also a three layer feedforward network with six inputs, a single hidden layer with ten neurons and a single output. The inputs are the generator's *actual* speed and *actual* terminal voltage deviations, Dw and DV_t respectively. Each of these inputs is time delayed by 10 ms and, together with four previously delayed values, form the six inputs. The output of the action network (neurocontroller), $A(t)$ = [DV_f, DP], the *deviation* in the field voltage, which augments the input to the generator's exciter and the deviation in the power, which augments the input to the generator's turbine.

5 Simulation of the Neurocontrollers and Results

The training procedure for the critic and action networks is similar to adaptive critic designs for SMIB [6]. It consists of two training cycles: the critic's and the action's. The critic's adaptation is done initially with a pretrained action network, to ensure that the whole system, consisting of the ACD and the power system, remains stable. The action network is pretrained on a linearized model of the generator. The action is trained further while keeping the critic network parameters fixed. This process of training the critic and the action one after the other is repeated until an acceptable performance is achieved. The ANN model parameters are assumed to have converged globally during its offline training [7] and, it is not adapted concurrently with the critic and action networks.

A discount factor g of 0.5 and the utility function given in eq. (9) are used in the Bellman's equation (eq. (2)) for the training of the critic network (eqs. (4)) and the action network (eq. (7)). Once the critic network's and action network's weights have converged, the action network (neurocontroller) is connected to the generator G1 (figure 6). A similar procedure is carried out in developing G2's neurocontroller.

$$U(t) = [4DV(t) + 4DV(t-1) + 16DV(t-2)]^2 \\ + [0.4Dw(t) + 0.4Dw(t-1) + 0.16Dw(t-2)]^2 \qquad (9)$$

At two different operating conditions and three different disturbances, the transient performance of the neurocontrollers are compared, with that of conventional controllers [12] (whose parameters are carefully tuned for the first set of the operating condition given in Appendix).

3% Step change in V_{t1} at first operating condition

At the *first* operating condition (Appendix), a 3% step increase occurs in the desired terminal voltage of G1. Figures 7 and 8 show that the neurocontroller ensures no overshoot on the terminal voltage and provides superior speed deviation damping unlike with the AVR and governor combination.

5% Step change in V_{t2} at second operating point

At the *second* operating condition (Appendix), a 5% step increase occurs in the desired terminal voltage of G2. Figures 9 and 10 show that the neurocontroller again provides the best damping, which proves that the neurocontroller has learned and adapted itself to the new operating condition. In fact, figure 10 shows signs of an inter-area mode starting up at about 4.5 seconds, and the neurocontroller is far more successful in damping this, than the conventional controllers.

Three phase short circuit

At the *second* operating condition (Appendix), a 100 ms short circuit occurs halfway between buses 3 and 4 (figure 6). Figure 11 shows that the neurocontroller again has better damping on the speed deviation of G1 and also on the terminal voltage (though not shown to conserve space).

All these results show that at operating conditions different from the one at which the AVRs and governors were tuned, and for large disturbances, their performance has degraded. The neurocontrollers on the other hand have given excellent performance under all the conditions tested. Many more tests were done to confirm this.

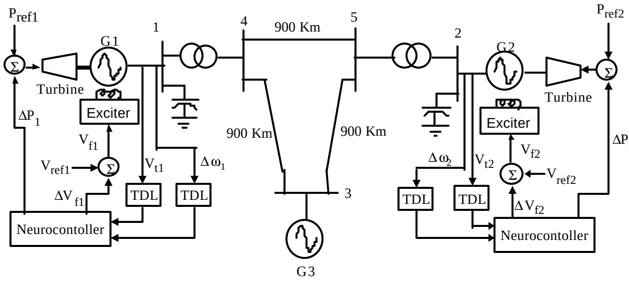


Figure 6: Multi-machine Power System with Neurocontrollers on Generators G1 and G2

6 Conclusions

A new method, based on derivative adaptive critics for for the design of neurocontrollers for generators in a multi-machine power system has been presented. All control variables are based on local measurements, thus, the control is decentralized. The results show that the neurocontrollers ensure a superior transient response throughout the system, for different disturbances and different operating conditions, compared to the conventional controllers, the AVRs and governors. Further studies on the practical implementation of these neurocontrollers on multiple generators on a laboratory system are currently in progress and preliminary results look encouraging. The success of the neurocontrollers are based on using deviation signals, and having a complete nonlinear model of the system. The use of such intelligent nonlinear controllers will allow power plants on the electric power grid to operate closer to their stability limits.

7 References

[1] Q.Lu and Y.Sun, "Nonlinear stabilizing control of multimachine systems," *IEEE Trans. Power System*, vol. 4, 1989, pp. 236-241.

[2] Y.Wang, D.J.Hill, L.Gao and R.H.Middleton, "Transient stability enhancement and voltage regulation of power system", *IEEE Trans. Power System*, vol. 8, 1993, pp. 620-627.

[3] Z.Qiu, J.F.Dorsey, J.Bond and J.D.McCalley, "Application of robust control to sustained oscillations in power systems", *IEEE Trans. Circuits System*, I, vol. 39, 1992, pp. 470-476.

[4] G.K.Venayagamoorthy and R.G.Harley, "A continually online trained artificial neural network identifier for a turbogenerator", *Proceedings of 1999 IEEE International Electric Machines and Drives Conference*, 0-7803-5293-9/99, pp. 404 – 406.

[5] G.K.Venayagamoorthy and R.G.Harley, "Experimental studies with a continually online trained artificial neural network controller for a turbogenerator", *Proceedings of the*

International Joint Conference on Neural Networks, IJCNN 1999, vol. 3, pp. 2158-2163.

[6] G.K.Venayagamoorthy, R.G.Harley, and D.C.Wunsch, “Comparison of a heuristic dynamic programming and a dual heuristic programming based adaptive critics neurocontroller for a turbogenerator”, *Proceedings of the International Joint Conference on Neural Networks*, IJCNN 2000, paper no. 660.

[7] G.K.Venayagamoorthy, R.G.Harley, and D.C.Wunsch, “Adaptive neural network identifiers for effective control of turbogenerators in a multimachine power system”, *Proceedings of the IEEE PES Winter Meeting 2001*, paper no. 2001WM154.

[8] G.Sybille, P.Brunelle, R.Champagne, L.Dessaint and Hoang Lehuy, *Power system blockset version 2.0*, Mathworks Inc., 2000.

[9] D.J.Limebeer, R.G.Harley and S.M.Schuck, “Subsychrnous Rresonance of Koeberg Turbogenerators and of a Laboratory Micro-alternator System”, *Transactions of the South African Institute of Electrical Engineers,* November 1979, pp. 278-297.

[10] P.Werbos, *Approximate dynamic programming for real-time control and neural modeling*, in Handbook of Intelligent Control, White and Sofge, Eds., Van Nostrand Reinhold, ISBN 0-442-30857-4, pp 493 – 525.

[11] D.Prokhorov, D Wunsch, “Adaptive Critic Designs”, *IEEE Trans. on Neural Networks,* vol. 8, no.5, pp 997-1007.

[12] W.K.Ho, C.C.Hang, L.S.Cao, “Tuning of PID controllers based on gain and phase margin specifications”, *Proceedings of the 12th Triennial World Congress on Automatic Control,* 1993, pp. 199 –202.

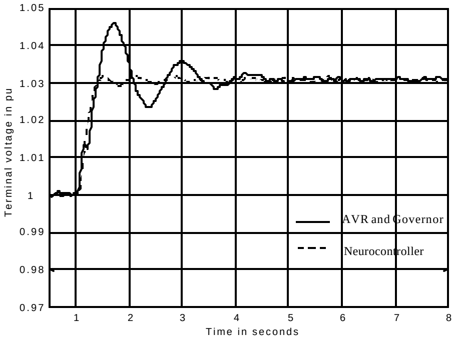


Figure 7: Terminal Voltage of Generator G1 for a 3% Step Change in its Desired Terminal Voltage

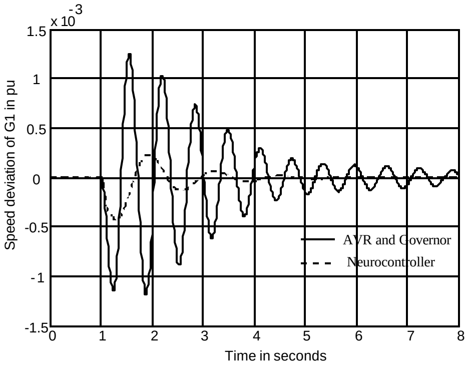


Figure 8: Speed Deviations of Generator G1 for a 3% Step Change in its Desired Terminal Voltage

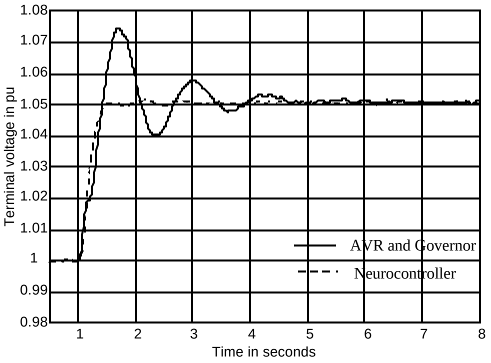


Figure 10: Terminal Voltage of Generator G2 for a 5% Step Change in its Desired Terminal Voltage

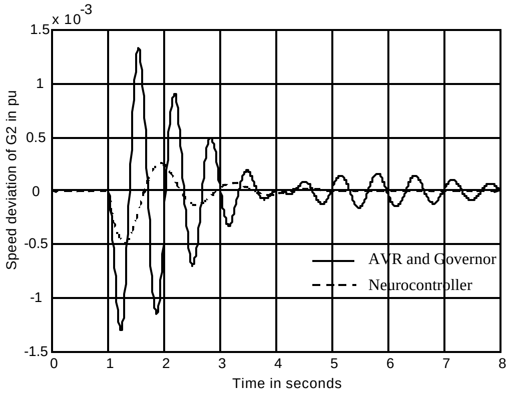


Figure 10: Speed Deviation of Generator G2 for a 5% Step Change in its Desired Terminal Voltage

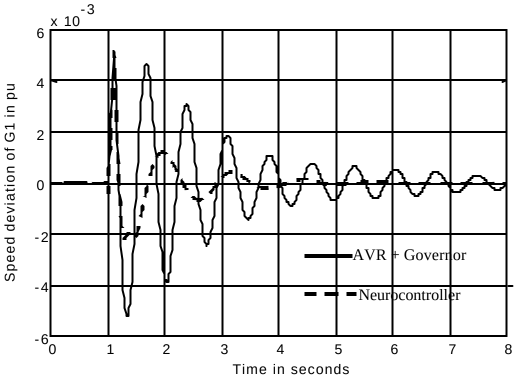


Figure 11: Speed Deviation of Generator G1 for a 100 ms 3-Phase Short Circuit between bus 3 and 4

8 Appendix

	Condition one		Condition two	
	G1	*G2*	*G1*	*G2*
P_e (pu)	0.200	0.200	0.3000	0.300
Q (pu)	-0.0216	-0.0218	-0.0493	-0.0341
V_t (pu)	1	1	1	1

Some ideas about reinforcement learning

Stuart Russell
(russell@EECS.Berkeley.EDU)
Computer Science Division, UC Berkeley

I will present three ideas that may help in scaling reinforcement learning and making it into a tool for understanding real biological behaviours:

1) that partial, structural constraints on behaviour may be combined easily with reinforcement learning to achieve complex skilled behaviours;
2) that the reward signals optimized by real organisms may be inferred from observation of behaviour;
3) that noise in biological control systems may have an important influence on the control strategies employed, and can be minimzed by some simple learning methods.

David Andre and Stuart Russell, ``State Abstraction for Programmable Reinforcement Learning Agents." Technical Report CSD-01-1156, Computer Science Division, UC Berkeley, 2001.

David Andre and Stuart Russell, ``Programmable Reinforcement Learning Agents." In Advances in Neural Information Processing Systems 13, MIT Press, 2001.

Andrew Y. Ng and Stuart Russell, ``Algorithms for inverse reinforcement learning." In Proceedings of the Seventeenth International Conference on Machine Learning, Stanford, California: Morgan Kaufmann, 2000.

Vassilis Papavassiliou and Stuart Russell, ``Convergence of reinforcement learning with general function approximators." In Proc. IJCAI-99, Stockholm, 1999.

Andrew Y. Ng, Daishi Harada, and Stuart Russell, ``Policy invariance under reward transformations: Theory and application to reward shaping." In Proc. ICML-99, Bled, Slovenia, 1999.

Daishi Harada and Stuart Russell, ``Extended abstract: Learning search strategies." In Proc. AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information, Stanford, CA, 1999.

R. Dearden, N. Friedman, and S. Russell, ``Bayesian Q-Learning." In Proc. AAAI-98, Madison, Wisconsin: AAAI Press, 1998.

S. Russell, ``Learning agents for uncertain environments (extended abstract)." In Proc. COLT-98, Madison, Wisconsin: ACM Press, 1998.
Ron Parr and Stuart Russell, ``Reinforcement Learning with Hierarchies of Machines." In Advances in Neural Information Processing Systems 10, MIT Press, 1998.

Ron Parr and Stuart Russell, ``Approximating Optimal Policies for Partially Observable Stochastic Domains." In Proc. Fourteenth International Joint Conference on Artificial Intelligence, Montreal, Canada, 1995.

State Abstraction for Programmable Reinforcement Learning Agents

Paper Tracking Number: 444

Keywords: Reinforcement learning, Markov decision processes.

Abstract

Safe state abstraction in reinforcement learning allows an agent to ignore aspects of its current state that are irrelevant to its current decision, and therefore speeds up dynamic programming and learning. This paper explores safe state abstraction in hierarchical reinforcement learning, where learned behaviors must conform to a given partial, hierarchical program. Unlike previous approaches to this problem, our methods yield significant state abstraction while maintaining *hierarchical optimality*, i.e., optimality among all policies consistent with the partial program. We show how to achieve this for a partial programming language that is essentially Lisp augmented with nondeterministic constructs. We demonstrate our methods on two variants of Dietterich's taxi domain, showing how state abstraction and hierarchical optimality result in faster learning of better policies and enable the transfer of learned skills from one problem to another.

Introduction

The ability to make decisions based on only *relevant* features is a critical aspect of intelligence. For example, if one is driving a taxi from A to B, decisions about which street to take should not depend on the current price of tea in China; when changing lanes, the traffic conditions matter but not the name of the street; and so on. *State abstraction* is the process of eliminating features to reduce the effective state space; such reductions can speed up dynamic programming and reinforcement learning (RL) algorithms considerably. Without state abstraction, every trip from A to B is a new trip; every lane change is a new task to be learned from scratch.

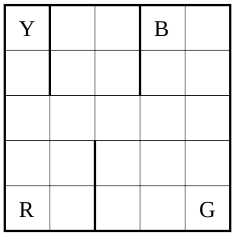
An abstraction is called *safe* if optimal solutions in the abstract space are also optimal in the original space. Safe abstractions were introduced by Amarel (1968) for the Missionaries and Cannibals problem. In our example, the taxi driver can safely omit the price of tea in China from the state space for navigating from A to B. More formally, the value of every state (or of every state-action pair) is independent of the price of tea, so the price of tea is irrelevant in selecting optimal actions. Boutilier *et al.* (1995) developed a general method for deriving such irrelevance assertions from the formal specification of a decision problem.

It has been noted (Dietterich 2000) that a variable can be irrelevant to the optimal decision in a state *even if it affects the value of that state*. For example, suppose that the taxi is driving from A to B to pick up a passenger whose destination is C. Now, C is part of the state, but is not relevant to navigation decisions between A and B. This is because the value (sum of future rewards or costs) of each state between A and B can be *decomposed* into a part dealing with the cost of getting to B and a part dealing with the cost from B to C. The latter part is unaffected by the choice of A; the former part is unaffected by the choice of C.

This idea—that a variable can be irrelevant to *part* of the value of a state—is closely connected to the area of *hierarchical reinforcement learning*, in which learned behaviors must conform to a given partial, hierarchical program. The connection arises because the partial program naturally divides state sequences into parts. For example, the task described above may be achieved by executing two subroutine calls, one to drive from A to B and one to deliver the passenger from B to C. The partial programmer may state (or a Boutilier-style algorithm may derive) the fact that the navigation choices in the first subroutine call are independent of the passenger's final destination. More generally, the notion of *modularity* for behavioral subroutines is precisely the requirement that decisions internal to the subroutine be independent of all external variables other than those passed as arguments to the subroutine.

Several different partial programming languages have been proposed, with varying degrees of expressive power. Expressiveness is important for two reasons: first, an expressive language makes it possible to state complex partial specifications concisely; second, it enables irrelevance assertions to be made at a high level of abstraction rather than repeated across many instances of what is conceptually the same subroutine. The first contribution of this paper is an agent programming language, ALisp, that is essentially Lisp augmented with nondeterministic constructs; the language subsumes MAXQ (Dietterich 2000), options (Precup & Sutton 1998), and the PHAM language (Andre & Russell 2001).

Given a partial program, a hierarchical RL algorithm finds a policy that is consistent with the program. The policy may be *hierarchically optimal*—i.e., optimal among all policies consistent with the program; or it may be *recursively optimal*, i.e., the policy within each subroutine is optimized *ignoring the calling context*. Recursively optimal policies may be worse than hierarchically optimal policies if the context



```
(defun root () (if (not (have-pass)) (get)) (put))
(defun get () (choice get-choice
                (action 'load)
                (call navigate (pickup))))
(defun put () (choice put-choice
                (action 'unload)
                (call navigate (dest))))
(defun navigate(t)
      (loop until (at t) do
           (choice nav (action 'N)
                       (action 'E)
                       (action 'S)
                       (action 'W))))
```

Figure 1: The taxi world. It is a 5x5 world with 4 special cells (RGBY) where passengers are loaded and unloaded. There are 4 features, `x,y,pickup,dest`. In each episode, the taxi starts in a randomly chosen square, and there is a passenger at a random special cell with a random destination. The taxi must travel to, pick up, and deliver the passenger, using the commands `N,S,E,W,load,unload`. The taxi receives a reward of -1 for every action, +20 for successfully delivering the passenger, -10 for attempting to `load` or `unload` the passenger at incorrect locations. The discount factor is 1.0. The partial program shown is an ALisp program expressing the same constraints as Dietterich's taxi MAXQ program. It breaks the problem down into the tasks of getting and putting the passenger, and further isolates navigation.

is relevant. Dietterich 2000 shows how a two-part decomposition of the value function allows state abstractions that are safe with respect to recursive optimality, and argues that "State abstractions [of this kind] cannot be employed without losing hierarchical optimality." The second, and more important, contribution of this paper is a three-part decomposition of the value function allowing state abstractions that are safe with respect to hierarchical optimality.

The remainder of the paper begins with background material on Markov decision processes and hierarchical RL, and a brief description of the ALisp language. Then we present the three-part value function decomposition and associated Bellman equations. We explain how ALisp programs are annotated with (ir)relevance assertions, and describe a model-free hierarchical RL algorithm for annotated ALisp programs that is guaranteed to converge to hierarchically optimal solutions[1]. Finally, we describe experimental results for this algorithm using two domains: Dietterich's original taxi domain and a variant of it that illustrates the differences between hierarchical and recursive optimality.

Background

Our framework for MDPs is standard (Kaelbling, Littman, & Moore 1996). An MDP is a 4-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ a set of actions, $\mathcal{T}$ a probabilistic transition function mapping $\mathcal{S}\times\mathcal{A}\times\mathcal{S} \to [0,1]$, and $\mathcal{R}$ a reward function mapping $\mathcal{S}\times\mathcal{A}\times\mathcal{S}$ to the reals. We focus on infinite-horizon MDPs with a discount factor β. A solution to an MDP is an optimal policy π^* mapping from $\mathcal{S} \to \mathcal{A}$ and achieves the maximum expected discounted reward. An SMDP (semi-MDP) allows for actions that take more than one time step. $\mathcal{T}$ is now a mapping from $\mathcal{S}\times\mathbf{N}\times\mathcal{S}\times\mathcal{A} \to [0,1]$, where $\mathbf{N}$ is the natural numbers; i.e., it specifies a distribution over both outcome states and action durations. $\mathcal{R}$ then maps from $\mathcal{S}\times\mathbf{N}\times\mathcal{S}\times\mathcal{A}$ to the reals. The expected discounted reward for taking action a in state s and then following policy π is known as the Q value, and is defined as $Q^\pi(s,a) = E[r_0 + \beta r_1 + \beta^2 r_2 + ...]$. Q values are related to one another through the Bellman equations (Bellman 1957): $Q^\pi(s,a) = \sum_{s',N} \mathcal{T}(s',N,s,a)[\mathcal{R}(s',N,s,a) + \beta^N Q^\pi(s',\pi(s'))]$. Note that $\pi \in \pi^*$ iff $\forall_s \pi(s) \in \arg\max_a Q^\pi(s,a)$.

In most languages for partial reinforcement learning programs, the programmer specifies a program containing choice points. A *choice point* is a place in the program where the learning algorithm must choose among a set of provided options (which may be primitives or subroutines). Formally, the program can be viewed as a finite state machine with state space Θ (consisting of the stack, heap, and program pointer). Let us define a joint state space Y for a program $\mathcal{H}$ as the cross product of Θ and the states, S, in an MDP $\mathcal{M}$. Let us also define Ω as the set of *choice states*, that is, Ω is the subset of Y where the machine state is at a choice point. With most hierarchical languages for reinforcement learning, one can then construct a joint SMDP $\mathcal{H} \circ \mathcal{M}$ where $\mathcal{H} \circ \mathcal{M}$ has state space Ω, and the actions at each state in Ω are the choices specified by the partial program $\mathcal{H}$. For several simple RL-specific languages, it has been shown that policies optimal under $\mathcal{H}\circ\mathcal{M}$ correspond to the best policies achievable in $\mathcal{M}$ given the constraints expressed by $\mathcal{H}$ (Andre & Russell 2001; Parr & Russell 1998).

The ALisp language

The ALisp programming language consists of the Lisp language augmented with three special macros:

- `(choice` *<label> <form0> <form1>* `...)` takes 2 or more arguments, where *<formN>* is a Lisp S-expression. The agent learns which form to execute.
- `(call` *<subroutine> <arg0> <arg1>*`)` calls a subroutine with its arguments and alerts the learning mechanism that a subroutine has been called.
- `(action` *<action-name>*`)` executes a "primitive" action in the MDP.

An ALisp program consists of an arbitrary Lisp program that is allowed to use these macros and obeys the constraint that all subroutines that include the choice macro (either directly, or indirectly, through nested subroutine calls) are called with the `call` macro. An example ALisp program is shown in Figure 1 for Dietterich's Taxi world (Dietterich 2000). It can be shown that, under appropriate restrictions (such as that the number of machine states Y stays bounded in every run of the environment), that optimal policies for the joint SMDP $\mathcal{H} \circ \mathcal{M}$ for an ALisp program $\mathcal{H}$ are optimal for the MDP $\mathcal{M}$ among those policies allowed by $\mathcal{H}$ (XXXX 2002).

[1] Proofs of all theorems are omitted for space reasons and can be found in an accompanying technical report (XXXX 2002).

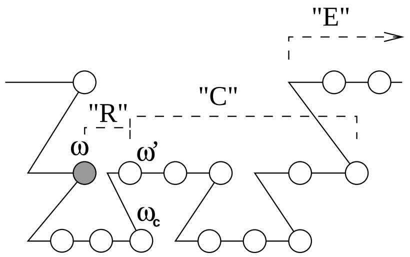


Figure 2: Decomposing the value function for the shaded state, ω. Each circle is a choice state of the SMDP visited by the agent, where the vertical axis represents depth in the hierarchy. The trajectory is broken into 3 parts: the reward "R" for executing the macro action at ω, the completion value "C", for finishing the subroutine, and "E", the external value.

Value Function Decomposition

A value function decomposition splits the value of a state/action pair into multiple additive components. Modularity in the hierarchical structure of a program allows us to do this decomposition along subroutine boundaries. Consider, for example, Figure 2. The three parts of the decomposition correspond to executing the current action (which might itself be a subroutine), completing the rest of the current subroutine, and all actions outside the current subroutine. More formally, we can write the Q-value for executing action a in $\omega \in \Omega$ as follows:

$$\begin{aligned} Q^{\pi}(\omega,a) = \quad & E\left[\sum_{t=0}^{\infty}\beta^t r_t\right] \\ = E\left[\sum_{t=0}^{N_1-1}\beta^t r_t\right] &+ E\left[\sum_{t=N_1}^{N_2-1}\beta^t r_t\right] + E\left[\sum_{t=N_2}^{\infty}\beta^t r_t\right] \\ = \quad Q_r^{\pi}(\omega,a) \quad &+ \quad Q_c^{\pi}(\omega,a) \quad + \quad Q_e^{\pi}(\omega,a) \end{aligned}$$

where N_1 is the number of primitive steps to finish action a, N_2 is the number of primitive steps to finish the current subroutine, and the expectation is over trajectories starting in ω with action a and following π. N_1, N_2, and the rewards, r_t, are defined by the trajectory. Q_r thus expresses the expected discounted reward for doing the current action ("R" from Figure 2), Q_c for completing rest of the current subroutine ("C"), and Q_e for all the reward external to the current subroutine ("E").

It is important to see how this three-part decomposition allows greater state abstraction. Consider the taxi domain, where there are many opportunities for state abstraction (as pointed out by Dietterich (2000) for his two-part decomposition). While completing the `get` subroutine, the passenger's destination is not relevant to decisions about getting to the passenger's location. Similarly, when navigating, only the current x/y location and the target location are important – whether the taxi is carrying a passenger is not relevant. Taking advantage of these intuitively appealing abstractions requires a value function decomposition, as Table 1 shows.

Before presenting the Bellman equations for the decomposed value function, we must first define transition probability measures that take the program's hierarchy into account. First, we have the SMDP transition probability $p(\omega', N|\omega, a)$, which is the probability of an SMDP transition to ω' taking N steps given that action a is taken in ω.

x	y	pickup	dest	Q	Q_r	Q_c	Q_e
3	3	R	G	0.23	-7.5	-1.0	8.74
3	3	R	B	1.13	-7.5	-1.0	9.63
3	2	R	G	1.29	-6.45	-1.0	8.74

Table 1: Table of Q values and decomposed Q values for 3 states and action $a =$ `(nav pickup)`, where the machine state is equal to {`get-choice`}. The first four columns specify the environment state. Note that although none of the Q values listed are identical, Q_c is the same for all three cases, and Q_e is the same for 2 out of 3, and Q_r is the same for 2 out of 3.

Next, let S be a set of states, and let $F_S^{\pi}(\omega', N|\omega, a)$ be the probability that ω' is the first element of S reached and that this occurs in N primitive steps, given that a is taken in ω and π is followed thereafter. Two such distributions are useful, $F_{SS(\omega)}^{\pi}$ and $F_{EX(\omega)}^{\pi}$, where $SS(\omega)$ are those states in the same subroutine as ω and $EX(\omega)$ are those states that are exit points for the subroutine containing ω. We can now write the Bellman equations using our decomposed value function, as shown in Equations 1, 2, and 3 in Figure 3, where $o(\omega)$ returns the next choice state at the parent level of the hierarchy, $i_a(\omega)$ returns the first choice state at the child level, given action a [2], and A_p is the set of actions that are not `calls` to subroutines. With some algebra, we can then prove the following results.

Theorem 1 *If Q_r^*, Q_c^*, and Q_e^* are solutions to Equations 1, 2, and 3 for π^*, then $Q^* = Q_r^* + Q_c^* + Q_e^*$ is a solution to the standard Bellman equation.*

Theorem 2 *Decomposed value iteration and policy iteration algorithms (omitted here) derived from Equations 1, 2, and 3 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

Extending policy iteration and value iteration to work with these decomposed equations is straightforward, but it does require that the full model is known – including $F_{SS(\omega)}^{\pi}(\omega', N|\omega, a)$ and $F_{EX(\omega)}^{\pi}(\omega', N|\omega, a)$, which can be found through dynamic programming. After explaining how the decomposition enables state abstraction, we will present an online learning method which avoids the problem of having to specify or determine a complex model.

State Abstraction

One method for doing learning with ALisp programs would be to flatten the subroutines out into the full joint state space of the SMDP. This has the result of creating a copy of each subroutine for every place where it is called with different parameters. For example, in the Taxi problem, the flattened program would have 8 copies (4 destinations, 2 calling contexts) of the `navigate` subroutine, each of which have to be learned separately. Because of the three-part decomposition discussed above, we can take advantage of state abstraction and avoid flattening the state space.

To do this, we require that the user specify which features matter for each of the components of the value function. The user must do this for each action at each choice point in the

[2] We make a trivial assumption that calls to subroutines are surrounded by choice points with no intervening primitive actions at the calling level. $i_a(\omega)$ and $o(\omega)$ are thus simple deterministic functions, determined from the program structure.

$$Q_r^\pi(\omega,a) = \begin{cases} \sum\limits_{\omega',N'} p(\omega',N|\omega,a)r(\omega',N,\omega,a) & \text{if } a \in A_p \\ Q_r^\pi(i_a(\omega),\pi(i_a(\omega))) + Q_c^\pi(i_a(\omega),\pi(i_a(\omega))) & \text{otherwise.} \end{cases} \quad (1)$$

$$Q_c^\pi(\omega,a) = \sum\nolimits_{(\omega',N)} F^\pi_{SS(\omega)}(\omega',N|\omega,a)\beta^N[Q_r^\pi(\omega',\pi(\omega')) + Q_c^\pi(\omega',\pi(\omega'))] \quad (2)$$

$$Q_e^\pi(\omega,a) = \sum\nolimits_{(\omega',N)} F^\pi_{EX(\omega)}(\omega',N|\omega,a)\beta^N[Q^\pi(o(\omega'),\pi(o(\omega')))] \quad (3)$$

$$\forall_{a\in A_p} Q_r^*(z_p(\omega,a),a) = \sum\nolimits_{(\omega',N)} p(\omega',N|\omega,a)r(\omega',N,\omega,a) \quad (4)$$

$$\forall_{a\notin A_p} Q_r^*(z_r(\omega,a),a) = Q_r^*(z_r(\omega'),a') + Q_c^*(z_c(\omega'),a'),\ \text{where } \omega' = i_a(\omega) \text{ and } a' = \arg\max\nolimits_b Q^*(\omega',b) \quad (5)$$

$$\forall_a Q_c^*(z_c(\omega,a),a) = \sum\nolimits_{(\omega',N)} F^*_{SS(\omega)}(\omega',N|\omega,a)\beta^N[Q_r^*(z_r(\omega',a),a') + Q_c^*(z_c(\omega',a),a')] \text{ where } a' = \arg\max\nolimits_b Q^*(\omega',b) \quad (6)$$

$$\forall_a Q_e^*(z_e(\omega,a),a) = \sum\nolimits_{(\omega',N)} F^*_{EX(z_e(\omega,a))}(\omega',N|z_e(\omega,a),a)\beta^N[Q^*(o(\omega'),a')] \text{ where } a' = \arg\max\nolimits_b Q^*(o(\omega'),b) \quad (7)$$

Figure 3: Top: Bellman equations for the three-part decomposed value function. Bottom: Bellman equations for the abstracted case.

program. We thus annotate the language with `:depends-on` keywords. For example, in the `navigate` subroutine, the `(action 'N)` choice is changed to

```
((action 'N)
  :reward-depends-on nil
  :completion-depends-on '(x y t)
  :external-depends-on '(pickup dest))
```

Note that `t` is the parameter passed into `navigate`. The Q_r-value for this action is constant – it doesn't depend on any features at all (because all actions in the Taxi domain have fixed cost). The Q_c value only depends on where the taxi is, and where it's trying to get to. The Q_e value only depends on the passenger's location (either in the Taxi or at R,G,B, or Y) and the passenger's destination. Thus, whereas a program with no state abstraction would be required to store 800 values, here, we only must store 117.

Safe state abstraction

Now that we have the programmatic machinery to define abstractions, we'd like to know when a given set of abstraction functions is safe for a given problem. To do this, we first need a formal notation for defining abstractions. Let $z_p[\theta,a]$, $z_r[\theta,a]$, $z_c[\theta,a]$, and $z_e[\theta,a]$ be abstraction functions specifying the set of relevant machine and environment features for each choice point θ and action a for the primitive reward, non-primitive reward, completion cost, and external cost respectively. In the example above, $z_c[nav,N] = \{\mathtt{x},\mathtt{y},\mathtt{t}\}$. Note that this function z groups states together into equivalence classes (for example, all states that agree on assignments to `x`, `y`, and `t` would be in an equivalence class). Let $z(\omega,a)$ be a mapping from a state-action pair to a canonical member of the equivalence class to which it under the abstraction z. We must also discuss how policies interact with abstractions. We will say that a policy π and an abstraction z are consistent iff $\forall_{\omega,a}\pi(\omega) = \pi(z(\omega,a))$ and $\forall_{a,b} z(\omega,a) = z(\omega,b)$. We will denote the set of such policies as Π_z.

Now, we can begin to examine when abstractions are safe. To do this, we define several notions of equivalence. We'll say that z_p is P-equivalent (Primitive equivalent) iff $\forall_{\omega,a\in A_p}$, $Q_r(\omega,a) = Q_r(z_p(\omega,a),a)$. We'll say that z_r is R-equivalent iff $\forall_{\omega,a\notin A_p,\pi\in\Pi_{z_r}}$, $Q_r(\omega,a) = Q_r(z_r(\omega),a)$. These two specify that states are abstracted together under z_p and z_r only if their Q_r values are equal. C-equivalence can be defined similarly.

For the E component, we can be more aggressive. The exact value of the external reward isn't what's important, rather, it's the behavior that it imposes on the subroutine. For example, in the Taxi problem, the external value after reaching the end of the `navigate` subroutine will be very different when the passenger is in the taxi and when she's not – but the optimal behavior for `navigate` is the same in both cases. Let h be a subroutine of a program $\mathcal{H}$, and let Ω_h be the set of choice states reachable while control remains in h. Then, we can define E-equivalence as follows:

Definition 1 (E-equivalence) *:*
z_e is E-equivalent iff (1) $\forall_{h\in\mathcal{H}}\forall_{\omega_1,\omega_2\in\Omega_h} z_e[\omega_1] = z_e[\omega_2]$ and (2) $\forall_\omega \arg\max_a Q_r^*(\omega,a) + Q_c^*(\omega,a) + Q_e^*(\omega,a) = \arg\max_a Q_r^*(\omega,a) + Q_c^*(\omega,a) + Q_e^*(z_e(\omega,a),a)$.

The last condition says that states are abstracted together only if they have the same set of optimal actions in the set of optimal policies. It could also be described as "passing in enough information to determine the policy". This is the critical constraint that allows us to maintain hierarchical optimality while still performing state abstraction.

We can show that if abstraction functions satisfy these four properties, then the optimal policies when using these abstractions are the same as the optimal policies without them. To do this, we first express the abstracted Bellman equations as shown in Equations 4 - 7 in Figure 3. Now, if z_p z_r, z_c, and z_e are P-, R-,C-, and E-equivalent, respectively, then we can show that we have a safe abstraction.

Theorem 3 *If z_p is P-equivalent, z_r is R-equivalent, z_c is C-equivalent, and z_e is E-equivalent, then, if Q_r^*, Q_c^*, and Q_e^* are solutions to Equations 4 - 7, for MDP $\mathcal{M}$ and ALisp program $\mathcal{H}$, then π such that $\pi(\omega) \in \arg\max_a Q^*(\omega,a)$ is an optimal policy for $\mathcal{H} \circ \mathcal{M}$.*

Theorem 4 *Decomposed abstracted value iteration and policy iteration algorithms (omitted here) derived from Equations 4 - 7 converge to Q_r^*, Q_c^*, Q_e^*, and π^*.*

Proving these various forms of equivalence might be difficult for a given problem. It would be easier to create abstractions based on conditions about the model, rather than conditions on the value function. Dietterich (2000) defines four conditions for safe state abstraction under recursive optimality. For each, we can define a similar condition for hierarchical optimality and show how it implies abstractions that satisfy the equivalence conditions we've defined. These simpler conditions are leaf abstraction (essentially the same as P-equivalence), subroutine irrelevance (features that are totally irrelevant to a subroutine), result-distribution irrelevance (features are irrelevant to the F_{SS} distribution for all policies), and termination (all actions from a state lead to an exit state, so Q_c is 0). We can encompass the last three conditions into a strong form of equivalence, defined as follows.

Definition 2 (SSR-equivalence) *An abstraction function z_c is strongly subroutine (SSR) equivalent for an ALisp program $\mathcal{H}$ iff the following conditions hold for all ω and policies π that are consistent with z_c.*

1. *Equivalent states under z_c have equivalent transition probabilities: $\forall_{\omega',a,a',N}$* $F_{SS}(\omega', N|\omega, a) = F_{SS}(z_c(\omega', a'), N|z_c(\omega, a), a)$ [3]
2. *Equivalent states have equivalent rewards: $\forall_{\omega',a,a',N}$* $r(\omega', N, \omega, a) = r(z_c(\omega', a'), N, z_c(\omega, a), a)$
3. *The variables in z_c are enough to determine the optimal policy: $\forall_a \pi^*(\omega) = \pi^*(z_c(\omega, a))$*

The last condition is the same sort of condition as the last condition of E-equivalence, and is what enables us to maintain hierarchical optimality. Note that SSR-equivalence implies C-equivalence.

The ALispQ learning algorithm

We present a simple model-free state abstracted learning algorithm based on MAXQ (Dietterich 2000) for our three-part value function decomposition. We assume that the user provides the three abstraction functions z_p, z_c, and z_e. We store and update $\hat{Q}_c(z_c(\omega, a), a)$ and $\hat{Q}_e(z_e(\omega, a), a)$ for all $a \in A$, and $\hat{r}(z_p(\omega, a), a)$ for $a \in A_p$. We calculate $\hat{Q}(\omega, a) = \hat{Q}_r(\omega, a) + \hat{Q}_c(z_c(\omega), a) + \hat{Q}_e(z_e(\omega, a), a)$. Note that, as in Dietterich's work, $\hat{Q}_r(\omega, a)$ is recursively calculated as $\hat{r}(z_p(\omega, a), a)$ if $a \in A_p$ for the base case and otherwise as $\hat{Q}_r(\omega, a) = \hat{Q}_r(i_a(\omega), a') + \hat{Q}_c(z_c(i_a(\omega)), a')$, where $a' = \arg\max_b \hat{Q}(i_a(\omega), b)$. This means that that the user need not specify z_r. We assume that the agent uses a GLIE (Greedy in the Limit with Infinite Exploration) exploration policy.

Imagine the situation where the agent transitions to ω' contained in subroutine h, where the most recently visited choice state in h was ω, where we took action a, and it took N primitive steps to reach ω'. Let ω_c be the last choice state visited (it may or may not be equal to ω, see Figure 2 for an example), let $a' = \arg\max_b \hat{Q}(\omega', b)$, and let r_s be the discounted reward accumulated between ω and ω'. Then, ALispQ learning performs the following updates:

```
(defun root () (progn (guess) (wait)  (get) (put)))
(defun guess () (choice guess-choice
                        (nav R)
                        (nav G)
                        (nav B)
                        (nav Y)))
(defun wait () (loop while (not (pass-exists)) do
                        (action  'wait)))
```

Figure 4: New subroutines added for the extended taxi domain.

- if $a \in A_p$, $\hat{r}(z_p(\omega, a), a) \leftarrow (1-\alpha)\hat{r}(z_p(\omega, a), a) + \alpha r_s$
- $\hat{Q}_c(z_c(\omega), a) \leftarrow (1-\alpha)\hat{Q}_c(z_c(\omega), a) + \alpha\beta^N [\hat{Q}_r(z_c(\omega'), a') + \hat{Q}_c(z_c(\omega'), a')]$
- $\hat{Q}_e(z_e(\omega, a), a) \leftarrow (1-\alpha)\hat{Q}_e(z_e(\omega, a), a) + \alpha\beta^N \hat{Q}_e(z_e(\omega', a'), a')$
- if ω_c is an exit state and $z_c(\omega_c) = \omega_c$ (and let a be the sole action there) then $\hat{Q}_e(z_e(\omega_c, a), a) \leftarrow (1-\alpha)\hat{Q}_e(z_e(\omega_c, a), a) + \arg\max_b \hat{Q}(\omega', b)$ [4]

Theorem 5 (Convergence of ALispQ-learning) *If z_r, z_s, and z_e are R-,SSP-, and E- Equivalent, respectively, then the above learning algorithm will converge (with appropriately decaying learning rates and exploration method) to a hierarchically optimal policy.*

Experiments

Figure 5 shows the performance of five different learning methods on Dietterich's taxi-world problem. The learning rates and Boltzmann exploration constants were tuned for each method. Note that standard Q-learning performs better than "ALisp program w/o SA" – this is because the problem is episodic, and the ALisp program has joint states that are only visited once per episode, whereas Q learning can visit states multiple times per run. Performing better than Q learning is the "Better ALisp program w/o SA", which is an ALisp program where extra constraints have been expressed, namely that the `load` (`unload`) action should only be applied when the taxi is co-located with the passenger (destination). Second best is the "ALisp program w/ SA" method, and best is the "Better ALisp program w/SA" method. We also tried running our algorithm with recursive optimality on this problem, and found that the performance was essentially unchanged, although the hierarchically optimal methods used 745 parameters, while recursive optimality used 632. The similarity in performance on this problem is due to the fact that every subroutine has a deterministic exit state given the input state.

We also tested our methods on an extension of the taxi problem where the taxi initially doesn't know where the passenger will show up. The taxi must guess which of the primary destinations to go to and wait for the passenger. We also add the concept of time to the world: the passenger will show up at one of the four distinguished destinations with a distribution depending on what time of day it is (morning or afternoon). We modify the root subroutine for the domain and add two new subroutines, as shown in Figure 4.

[3] We can actually use a weaker condition: Dietterich's (2000) factored condition for subtask irrelevance

[4] Note that this only updates the Q_e values when the exit state is the distinguished state in the equivalence class. Two algorithmic improvements are possible: using all exits states and thus basing Q_e on an average of the exit states, and modifying the distinguished state so that it is one of the most likely to be visited.

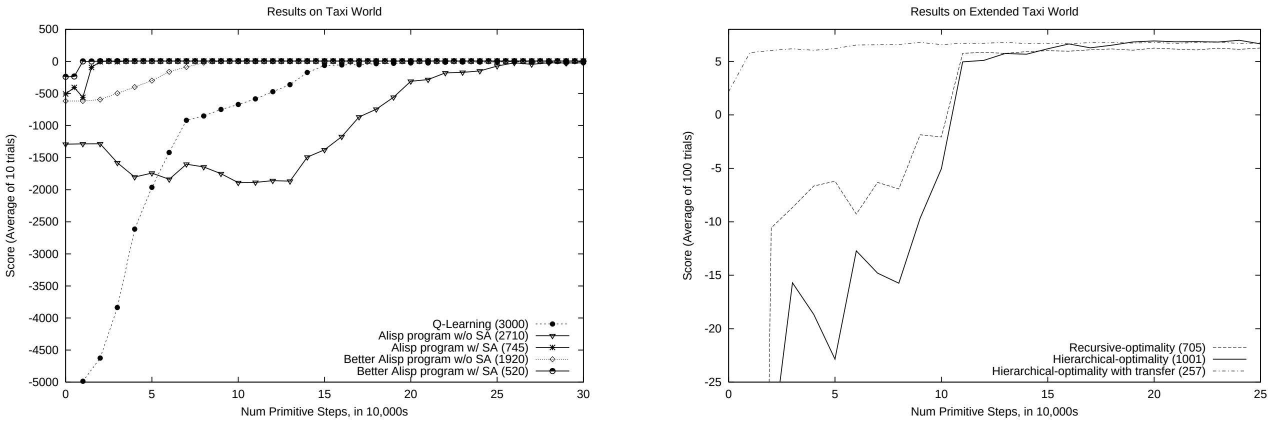


Figure 5: Learning curves for the taxi domain (left) and the extended taxi domain with time and arrival distribution (right), averaged over 50 training runs. Every 10000 primitive steps (x-axis), the greedy policy was evaluated for 10 trials, and the score (y-axis) was averaged. The number of parameters for each method is shown in parentheses after its name.

The right side of Figure 5 shows the results of running our algorithm with hierarchical versus recursive optimality. Because the arrival distribution of the passengers is not known in advance, and the effects of this distribution on reward are delayed until after the guess subroutine finishes, the recursively optimal solution cannot take advantage of the different distributions in the morning and afternoon to choose the best place to wait for the arrival, and thus cannot achieve the optimal score. None of the recursively optimal solutions achieved a policy having value higher than 6.3, whereas every run with hierarchical optimality found a solution with value higher than 6.9. In general, hierarchical optimality frees the programmer from having to specify pseudo-reward values at the exit states of subroutines. Secondly, Figure 5 also shows the results of transferring the `navigate` subroutine from the simple version of the problem to the more complex version. By analyzing the domain, we were able to determine that an optimal policy for `navigate` from the simple taxi domain would have the correct local sub-policy in the new domain, and thus we were able to guarantee that transferring it would be safe.

Discussion and Future Work

This paper has presented ALisp, shown how to achieve safe state abstraction for ALisp programs while maintaining hierarchical optimality, and demonstrated that doing so speeds learning considerably.

Several questions might arise for the reader when reading this work. First, one might wonder if the greater expressiveness of ALisp is necessary in fully-observable worlds, given that optimal policies are guaranteed to be implementable with lookup tables. In addition to providing a more concise method for partial programming, ALisp provides the programmer with all the expressiveness of a full programming language, enabling abstraction that would be difficult or impossible to create otherwise. We are also presently investigating using our three-part decomposition for partially-observable domains. Our plan is to start with safe state abstraction, then do function approximation for each component of our three-part abstracted and decomposed value function. (Makar, Mahadevan, & Ghavamzadeh 2001) have examined this issue using MAXQ and recursive optimality.

Secondly, the reader might wonder if, by rearranging the hierarchy of a recursively optimal program to have "morning" and "afternoon"`guess` functions, one could avoid the deficiencies of recursive optimality. Although possible in this case, in general, this method can result in adding an exponential number of subroutines (essentially, one for each possible subroutine policy). Also, in recursive optimality, the programmer must precisely choose the values for the pseudo rewards in each subroutine. In the taxi domain, this is fairly straightforward, but in cases with more complex dynamics it can be be prohibitive.

Finally, our system requires that the user specify the set of state abstractions to use. As previously mentioned, it would be preferable to automatically identify those state abstractions warranted by the environment's dynamics. Combining our three-part value function decomposition with Boutillier's (Boutilier *et al.* 2000) off-line inferential approach to finding state abstractions looks promising.

References

[1] Amarel, S. 1968. On representations of problems of reasoning about actions. In *Machine Intelligence 3*, volume 3. 131–171.

[2] Andre, D., and Russell, S. 2001. Programmatic reinforcement learning agents. In *NIPS 13*.

[3] Bellman, R. E. 1957. *Dynamic Programming*. Princeton, New Jersey: Princeton University Press.

[4] Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI-2000*.

[5] Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *AAAI-95*.

[6] Dietterich, T. G. 2000. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR* 13:227–303.

[7] Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *JAIR* 4:237–285.

[8] Makar, R.; Mahadevan, S.; and Ghavamzadeh, M. 2001. Hierarchical multi-agent reinforcement learning. In *Agents-2001*.

[9] Parr, R., and Russell, S. 1998. Reinforcement learning with hierarchies of machines. In *NIPS 10*.

[10] Precup, D., and Sutton, R. 1998. Multi-time models for temporally abstract planning. In *NIPS 10*.

[11] XXXX, X. 2002. State abstraction in hierarchical RL tech report: name and url removed for blind review.