EFFICIENT REINFORCEMENT LEARNING IN CONTINUOUS ENVIRONMENTS

A Dissertation

Presented to the Faculty of the Graduate School

of the College of Computer Science

of Northeastern University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mohammad A. Al-Ansari

August 30, 2001

Dedication

This work is dedicated to my parents, Abdulrahman and Huda, and my daughter Abir.

Acknowledgments

This work could not have been completed without the efforts of many individuals who have provided the professional and moral support that I needed.

I thank my advisor, Professor Ron Williams, from whom I have learned so much. His vast expertise in the field of Machine Learning in particular, and AI in general, provided inspiration and insight into many interesting areas of research. His guidance helped me focus on the topic of this dissertation and his advice provided invaluable direction throughout the process.

Professor Rich Sutton has been a very influential teacher, through his seminal writings in Reinforcement Learning and the many talks that I heard him deliver, all of which helped pique my interest in the field and maintain it until today. For that, I thank him. I also deeply appreciate his serving on my dissertation committee and giving the sound advice and direction that helped improve the final outcome of this work.

I thank Professors Robert Futrelle and Bryant York for providing many useful comments about my proposal, which helped in shaping the directions that my work followed. I also thank them for their very useful comments on my dissertation.

I also thank King Saud University and its College of Computer and Information

Sciences for providing me with a considerable part of the financial support that I needed to pursue my research.

On the personal front, my indebtedness to my parents, Huda and Abdulrahman, is eternal. They have provided endless, invaluable support and have never stopped giving. I could not have arrived where I am today without their assistance in countless ways. My daughter, Abir, was always the ultimate inspiration and motivation, especially when the task seemed most daunting. Her light always shone at the end of the tunnel, showing me the way.

I thank my sister Lubna for her sound advice and the strong positive effect she has had on my life in numerous ways. My brothers Asim, Hany and Abdulaziz were always there with encouragement along the way. I especially appreciate the extra efforts that Hany and Asim extended at the end of this process.

I thank my lifelong friend, Sulaiman Mirdad, for his unrelenting, sincere support, since our childhood, in personal and professional matters alike. His intelligence and free spirit made him an invaluable aid, always able to put matters in perspective no matter how bad they seemed.

Finally, I thank my dear friend, Hamdi Tchelepi, for his continuous support and encouragement and for our many unique, mind-opening discussions. His remarkable analytical abilities were indispensable tools that always helped me see things in the proper light, no matter what the circumstances were.

Abstract

Reinforcement Learning (RL) is a machine learning paradigm with which autonomous agents can improve their behavior in unknown environments based on their own experience without an explicit teacher signal. RL algorithms are based on estimating a value function over the state space, and scaling them to large state spaces remains a challenge. One approach, known as *variable resolution*, is to focus representational power in regions of the state space where experience shows it is most needed. One of the most promising variable resolution methods is *parti-game*, which has competitive performance and has shown promising scalability on the class of deterministic, continuous, goal-type RL problems. It performs dynamic, kd-tree-based partitioning of the state space based on a game-theoretic approach to assigning costs to partitions, and it uses an *a priori* local controller to try to navigate through the partitions it creates to reach the goal. Despite its promise, however, parti-game has a number of shortcomings relating to efficiency, consistency, sub-optimality of solutions found and reliance on a designer-supplied local controller.

This work introduces a family of variable resolution algorithms, in the same spirit of parti-game, that addresses each of these drawbacks, thus providing a powerful, *a-priori*-model-independent paradigm for finding higher quality solutions of

RL problems of this class, and doing so more efficiently, more reliably and with better scalability.

The algorithms introduced in this work have shown an average of 71% reduction over parti-game in the number of state space partitions generated and more than 58% improvement in the total steps needed to reach a stable solution. These algorithms also reduce the average per-step computation cost by a factor that increases as the number of partitions and dimensions increases. Furthermore, we show that the algorithms we introduce are far less sensitive to the configuration of the problem they are solving, making their performance far more consistent than parti-game's, in some cases improving the performance by two orders of magnitude.

Finally, this dissertation presents a general scheme for eliminating the need for a designer-supplied local controller, requiring only that a small, discrete set of suitable actions be defined by the designer. This extends the applicability of this family of algorithms to problems for which designing a local controller is not possible.

Contents

Abstract **ix**

List of Figures **xvii**

List of Tables **xxi**

1 Introduction **1**

1.1 The RL Problem 2

1.1.1 Types of Models 4

1.2 Variations in RL problems 5

1.2.1 *A priori* Knowledge 5

1.2.2 Deterministic vs. Stochastic Environments 7

1.2.3 Discrete vs. Continuous State and Action Spaces 8

1.3 Convergence Results 10

1.4 Handling Large State Spaces 10
1.4.1 Efficiency of Dynamic Programming 11
1.4.2 The Need for Using Function Approximators 11
1.4.3 Difficulties with Using Function Approximators 13
1.4.4 Variable Resolution Methods 17
1.5 Dissertation Summary 18

2 The Parti-Game Algorithm 21

2.1 Introduction 21
2.2 Performance 25
2.3 Promise and Limitations 26
2.4 Contributions of this Dissertation 28

3 Efficient On-Line Variable Resolution Algorithms 31

3.1 Partitioning Strategies 32
3.1.1 Effects on Exploration 33
3.2 Partitioning Only Losing Cells 35
3.2.1 Problem Domains 36
3.2.2 Experimental Results 46
3.3 Balanced Partitioning 49

3.3.1 Reason for the Imbalance 51
3.3.2 Algorithm for Balanced Partitioning 53
3.3.3 Experimental Results 54
3.3.4 Acrobot Results 61
3.4 Contributions 67

4 Low Computational Requirements **69**

4.1 Single-Pass Dynamic Programming 70
4.1.1 The Algorithm 71
4.1.2 Proof of Correctness 72
4.2 Incremental Computation 73
4.2.1 The Algorithm 74
4.2.2 Proof of Correctness 76
4.3 Effect on Performance 78
4.4 Contributions 81

5 Optimizing Solutions **83**

5.1 Other Gradients 85
5.2 Algorithm for Optimizing Solutions 87
5.3 Applicability 88

5.4 Other Examples 89

5.5 Contributions 90

6 *A priori* Model-Independence 93

6.1 Introduction 93

6.2 The Role of Local Controllers 94

6.2.1 The Need and Dependence on Local Controllers 96

6.2.2 How Local Controller Quality Affects Solution Quality 97

6.2.3 Learning the Local Controller 98

6.3 Learning Local Controllers Using Tile Coding 100

6.3.1 Tile Coding Function Approximators 100

6.3.2 Configuring the Tile Coding Approximator 103

6.3.3 Choosing the Parameters 105

6.3.4 Exploration Strategy 105

6.3.5 The Algorithm 110

6.3.6 The Curse of Dimensionality Revisited 111

6.4 Simulation Results 117

6.4.1 Mazes 118

6.4.2 How LLPPG+LLC Can Outperform LLPPG 121

6.4.3 Stability of Solutions Under LLPPG+LLC 125

6.4.4 Puck on a Hill 127

6.4.5 Application of the Global Optimization Algorithm to the Puck Problem 131

6.4.6 Nine-Joint Arm 131

6.4.7 The Acrobot 139

6.5 Discussion 141

6.6 Contributions 142

7 Conclusions and Future Directions **145**

7.1 Summary of Contributions 145

7.2 Future Directions 149

7.2.1 Data Efficiency 149

7.2.2 Using Other Variable Resolution Representations 151

7.2.3 Expanding the Class of Problems 152

Bibliography 160

List of Figures

1 A set of continuous mazes. 39

2 An ice puck on a hill. 39

3 A snake-like 9-joint robot arm. 44

4 The Acrobot. 44

5 Results of applying parti-game and loser-partitioning parti-game to maze a, the puck-on-a-hill and the multi-joint arm problems. 47

6 A very simple maze, first introduced in Figure 1(d). 50

7 Partitioning produced by (a) parti-game and (b) parti-game with partitioning only losing cells for the very simple maze introduced in Figure 1(d). 50

8 The partitioning of the maze of Figure 1(d) after 829 steps and 10 rounds of partitioning. 52

9 Results of applying PG, LPPG and LLPPG to maze a, the puck on a hill and the multi-joint arm problems. 56

10 The result of partitioning largest cells on the losing side in the maze introduced in Figure 1(d). 57

11 Graph showing the number of seconds needed to reach a solution by PG, LPPG and LLPPG as a function of trial number when applied to the Acrobot problem. 63

12 Graph showing the number of partitions needed by PG, LPPG and LLPPG as a function of trial number when applied to the Acrobot problem. 64

13 A re-creation of the maze used in (Moore and Atkeson 1995). (a) shows the empty maze and (b) shows the solution to which our implementation of parti-game converged. 80

14 The results of the final trial of applying (a) PG, (b) LPPG and (c) LLPPG to the simple maze of Figure 1(a). 84

15 The solution found by applying the global improvement algorithm on the maze of Figure 1(a). 86

16 (a) A sub-optimal-form solution to the puck-on-a-hill problem. (b) The optimal-form solution reached after applying the global optimization algorithm to the solution in (a). 90

17 A 2-tiling tile coding approximator for a 2D state space. 102

18 The eight trials required to stabilize LLPPG+LLC applied to maze a. 120

19 The results of the five iterations needed by LLPPG+LLC to stabilize at the non-optimal form solution in (e) above. 128

20 The optimal-form solution to the puck-on-a-hill problem reached after applying the global optimization algorithm of Chapter 5 to the solution reached by LLPPG+LLC shown in Figure 19(d). 132

21 Graph showing the number of seconds needed to reach a solution by PG, LPPG, LLPPG and LLPPG+LLC as a function of trial number when applied to the Acrobot problem. 143

22 Graph showing the number of partitions needed by PG, LPPG, LLPPG and LLPPG+LLC as a function of trial number when applied to the Acrobot problem. 144

List of Tables

1 Results of applying parti-game (PG) and loser-partitioning parti-game (LPPG) to four mazes, the puck-on-a-hill and the nine-joint arm problems. Smaller numbers are better and are shown in **bold.** 48

2 Results of applying PG, LPPG and LLPPG on three of the problem domains. 59

3 A comparison between PG and Q-learning, performed by Moore and Atkeson (1995), and our implementation of PG using our Algorithms 4 and 5. 80

4 A repetition of some of the results of Table 2 in addition to the results of applying LLPPG+LLC to the same set of problems. . . . 122

5 The results of applying PG, LPPG, LLPPG and LLPPG+LLC to the nine-joint arm problem. 137

Chapter 1

Introduction

Reinforcement learning (RL) is a machine learning paradigm in which an *autonomous agent* learns from its own experience to improve its behavior in an unknown environment. In contrast to supervised learning, the RL agent learns without explicit teacher input about the exact behavior that is required.

To introduce reinforcement learning, we will start by outlining the general RL problem and the algorithms for dealing with it. We will then discuss the various ways actual problems can differ from the general problem and give a brief outline of current methods for tackling each of these variations.

1.1 The RL Problem

In a typical RL problem, an autonomous agent is required to act in an unknown environment. Based on what the agent can discern about the current state from its sensory input at the current time step, it chooses an action to perform from the set of possible actions it associates with that state.[1] These actions can cause state changes in the dynamic system in which the agent operates. Thus, the agent can be viewed as a *controller* of such a dynamic system. Upon entering a state, the agent receives a scalar input, called *reinforcement* or *reward*. The agent's goal is to maximize the expected value of the *return*, a function of the series of rewards it receives over time that is usually either the simple or discounted sum of the rewards.

The agent maintains a mapping from states to actions, called a *policy*, which dictates what action to take at a given state. To achieve its goal, the agent needs to *learn* a policy that yields the maximum expected return. Such policy is called an *optimal policy*.

The most common approach to solving RL problems is to compute an *optimal value function* whose value at each state is the maximum expected return to be received when starting at that state and following an optimal policy. Once an

[1]For the rest of this discussion, we will assume that complete state information is always available to the agent. Problems in which the agent does not have complete access to state information (called *hidden state* problems) will not be discussed here.

optimal value function is computed, we get an optimal policy just by following a *greedy policy* based on it. A greedy policy is one that, at every state, chooses the immediate action that results in the successor state with the highest value.

The optimal value of a state x, $V^*(x)$, is satisfied by the *Bellman optimality equation*:[2]

$$V^*(x) = \max_{a \in A_x} [r(x,a) + \gamma V^*(x')]$$

where x is a state, $V^*(x)$ is the optimal value of state x, A_x is the set of allowable actions at x, $r(x,a)$ is the reward received after taking action a in state x, γ is a discount factor < 1, and x' is the state entered when action a is taken in state x.

A different form of value function, which was introduced with *Q-Learning* (Watkins 1989), computes values for state-action pairs (called *Q-values*). The Bellman optimality equation for the Q-values is:

$$Q^*(x,a) = r(x,a) + \gamma \max_{a' \in A_{x'}} Q^*(x',a')$$

In order to execute a greedy policy based on regular state values, all allowable actions at the current state have to be tried out to determine which successor state has the highest value before any action can be taken. Therefore, a model of the dynamics has to be used in this case. The advantage of Q-values is that the merits

[2]For the sake of simplicity, we only give the forms of the Bellman equation for deterministic environments here. Stochastic environments are also possible, as will be discussed in section 1.2.2.

of allowable actions from the current state can be assessed without having to first execute each of them. As a result, no model is needed to execute a greedy policy based on Q-values.

1.1.1 Types of Models

What we mean by a model of the dynamics is a function that describes what the effects of taking actions in the environments are. Specifically, the dynamics can be described using a transition function, δ, that maps a pair consisting of a state, s, and an action, a, to the state that the agent would enter as a result of taking action a in state s. Thus, the dynamics is formally defined as $\delta : S \times A \rightarrow S$, where S is the set of states and A is the set of allowable actions. Another function the agent is exposed to is the *reward* function, which determines how much reward to be given to the agent upon entering a particular state, in which case the function would be of the form $S \rightarrow \mathcal{R}$, or upon taking an action in a state, which would be of the form $S \times A \rightarrow \mathcal{R}$.

1.2 Variations in RL problems

The generic problem outlined above was introduced to give a general overview of the types of problems dealt with in the field. The decision of what method to use to find a solution to a particular RL problem is affected by certain properties of this given problem. In what follows we describe some properties that distinguish different types of RL problems and briefly discuss existing methods for handling the various cases.

1.2.1 *A priori* Knowledge

One important aspect in deciding how to handle a RL problem is how much information is available to the agent *a priori*. In its most challenging form, the agent in a RL problem starts with no *a priori* knowledge of the dynamics of the system or its reward structure, and it is expected to base its learning on its own direct *exploration* of the space. However, simpler models in which the dynamics are known *a priori* are also challenging and of interest.

When complete information about the dynamics of the system and the reward structure is available, and when the problem at hand has discrete state and action spaces,[3] an optimal value function can be found *off-line* by solving the appropriate

[3]Handling continuous problems will be discussed in section 1.2.3.

form of the Bellman equation using one of many known *Dynamic Programming* (DP) algorithms, like *value iteration* or *policy iteration* (Bertsekas 1987, 1995). These methods can, at least in theory, arrive at an optimal policy for such problems under certain conditions. In practice, however, these algorithms are often too computationally expensive. Thus, when dealing with complicated problems, we may be forced to settle for finding near-optimal policies or ones that only cover those parts of the space considered "interesting".

Two kinds of approaches exist for dealing with situations where a model of the dynamics is not available. One approach is to start out by exploring the state space, collecting enough data to build a model of the dynamics and reward structure, then proceed to find an optimal policy off-line based on these models. However, in many situations it is desirable that the agent be able to start acting in the environment early on and that it improves its behavior incrementally. This requires a learning approach that gradually uses collected data to build increasingly better policies. Methods that achieve this, usually called *on-line* or *direct* methods, have to balance exploration of new parts of the space with exploiting information already collected in improving its current policy. Barto, Sutton, and Watkins (1990) give a good introductory treatment of these issues, outlining both model-based DP methods and direct methods based on Sutton's (1988) Temporal Differencing (TD) framework.

Goal Problems

A restricted class of RL problems is one in which the agent only receives a reward when it reaches a *goal state*. The reward structure in such problems is very simple and is an example of *a priori* knowledge that can be easily supplied to the agent to save it a lot of exploration. An interesting algorithm that takes advantage of this extra information in these cases is the *parti-game* algorithm which we will talk about in detail in later sections.

1.2.2 Deterministic vs. Stochastic Environments

Problems can also differ in whether the environment is deterministic or stochastic. In the former case, actions taken at a particular state always have the same effect on the environment. If the environment is stochastic, on the other hand, the effect of taking an action at a state is determined by some probability distribution over a *set* of possible next states. As with the rest of the dynamics specifications, the agent is either provided with these probability distributions *a priori* or it has to learn them on its own. The aforementioned DP methods for solving RL problems under complete information apply both to the deterministic and stochastic cases. When these probability distributions are not available before hand, the agent needs

to collect statistics about the results of actions during its exploration and compute estimates of these probability distributions. These estimates are then used in computing the optimal policy.

Most of the current RL theory is specified for the more general stochastic case and is thus applicable to both types of problems.

1.2.3 Discrete vs. Continuous State and Action Spaces

Most early RL research had been focused on problems with discrete state spaces which could typically be framed as *Markov Decision Problems* (MDPs). Consequently, most of the important theory in the field only applies to discrete problems and does not readily extend to problems with continuous dynamics. Continuous problems are usually dealt with by quantizing the state space in a way which facilitates arriving at a suitable solution and then treating the problem as a discrete space one.

This quantization can either be uniform or variable across the space with higher resolution in areas where the designer thinks it is needed. Uniform quantization requires no knowledge of the characteristics of the problem but it could be unnecessarily wasteful of space. Indeed if the state space is large, the space requirements

for such representation can be completely unfeasible. Quantizing at a variable resolution can be much more feasible, but it requires much interference on the part of the designer. Deciding how to properly partition a space can be very difficult especially in high dimensional problems. For this reason, *adaptive partitioning* of the space emerges as a good solution to this problem. These issues will be discussed in more detail in sections 1.4.2 and 1.4.4.

Similarly, most work in the field has concentrated on problems that have discrete action spaces, where the number of allowable actions at each state is finite. Although this is sufficient in some cases, real-world problems often allow continua of actions. Similarly to the continuous state space case, the traditional approach for handling continuous action cases is to quantize the usually bounded action space and treat the problem as a discrete action problem. The drawback of this approach is that it does not guarantee finding optimal policies because the agent does not have the freedom of selecting any action value from the allowable ranges. There has been relatively little work done that considers the full ranges of actions available to the agent. Examples of such research are Luus (1989, 1992) and Baird and Klopf (1993).

1.3 Convergence Results

Major convergence proofs of important RL paradigms have been arrived at in the past several years (e.g., Sutton 1988; Watkins 1989; Dayan 1992; Watkins and Dayan 1992). These results have added theoretical rigor to the already promising empirical results that these methods have achieved by proving convergence to the optimal solution.

However, these proofs rely specifically on the look-up table representation of value functions. Unfortunately, using look-up tables proves to be impractical when the problems we are dealing with have large state spaces. The following section introduces these problems and the solutions currently available for dealing with them.

1.4 Handling Large State Spaces

Having introduced the RL paradigm and the types of problems it deals with, we now focus on its applicability in practice. Arguably the most important obstacle facing RL is scalability of its algorithms to large problems. Since current RL methods mostly focus on building a value function, the amount of work needed to solve an RL problem is a function of the number of states for which values need to be

computed.

1.4.1 Efficiency of Dynamic Programming

DP-based techniques, which sit at the core of most traditional RL methods, generally require time that is polynomial in the number of states and actions. Compared to other methods for solving MDP's, like linear programming and direct search in the policy space, DP is the most efficient (Littman, Dean, and Kaelbling 1995; Sutton and Barto 1998).

The basic RL problem, however, is one in which the number of states can grow exponentially with the number of dimensions. This is the problem that Bellman (1957) called the *curse of dimensionality*, which renders classical, tabular DP-based RL methods all but useless with large high-dimensional problems.

1.4.2 The Need for Using Function Approximators

Since the sizes of state spaces, and consequently the number of states for which a value needs to be computed, grow exponentially with the number of dimensions, we can not expect the look-up table representation to be practical for arbitrary high-dimensional problems. In fact, the size of the state space could be too large for look-up table representation even for some low-dimensional problems.

Furthermore, even assuming a look-up table representation is possible for a given state space, finding a globally optimal value function requires repeatedly updating the value of each cell in such a look-up table a large numbers of times. This means that the time requirements for such computation grows even faster than the exponentially growing space requirements.

This has led researchers to try employing more economical ways of representing value functions, such as parametric function approximators, including multi-layer perceptrons, regression methods, etc. Use of such function approximators addresses multiple problems. First, since the function approximators used typically have far fewer parameters than the functions they are meant to represent, a big reduction in storage requirements is achieved. It also means, however, that such function approximators lack the representational power to exactly represent the values of each individual state of a given arbitrary function. This lack of representational power means that these types of function approximators have to "generalize" values learned at a single state in the space to many other states. This is another way in which the use of function approximators helps in reducing the computational requirements of RL algorithms, because it reduces the number of updates needed to learn the value function, thus making tackling higher dimensional problems more feasible.

The exact way in which generalization is achieved by a function approximator is controlled by learning algorithms that search for parameter vectors that minimize the error between the desired and actual function values. Because generalization is a function of the parameter vector, the goal of this search can also be seen as the parameter vector with which the best generalization is achieved.

1.4.3 Difficulties with Using Function Approximators

While function approximators clearly have the potential to expand the practical applicability of RL methods to larger problems, using them introduces some difficulties. These difficulties relate to issues of convergence, type of solutions arrived at when convergence is guaranteed and performance of the algorithms.

Convergence

The first difficulty with using function approximators in RL is that it affects whether convergence can be achieved. While convergence is guaranteed when a look-up table is used, this is not the case when it is replaced with a function approximator. Many examples have been presented that show how RL algorithms may not converge to a solution when a function approximator is used. Thrun and Schwartz (1993) outlined some of the pitfalls that might be encountered when a function approximator is

used in the place of a look-up table and that it is possible that the algorithm would diverge. Bradtke (1993) showed that while Q-Learning converges to the optimal solution with probability one when used on a problem with discrete state and action spaces and a look-up table representation is used, it only converges locally when used with a certain class of control problems that have continuous state and action spaces. Boyan and Moore (1995) give some negative examples of using function approximators instead of look-up tables for a number of simple control tasks. Sutton (1996), in response to the problems outlined by Boyan and Moore (1995), presented successful empirical results on the same set of problems, using a straightforward RL method that uses *tile coding* (also known as *sparse coarse coding* and *CMAC's*, for Cerebellar Model Articulator Controllers) to represent value functions.

Some more positive convergence results have been achieved recently. Tsitsiklis and Roy (1997) proved convergence of TD prediction (i.e., policy evaluation) with probability one with linear function approximators. They also showed an upper bound of the value function arrived at in terms of the minimum mean squared error, with minimum error achieved only when TD's parameter, λ, is 1. However, this does not apply for the complete RL problem in which the policy continually changes until an optimal one is found.

Baird (1995) defined a class of algorithms he calls *residual algorithms* that can

potentially use any type of function approximator in conjunction with traditional RL algorithms and arrive at a value function for a given policy by performing gradient descent on an error function combining the *Bellman residual* and TD error. One problem with this method is that it relies on the existence of two independent samples of the next state, something that can not be provided when learning on-line without the availability of a model.

The above results only provide convergence guarantees for evaluating a given policy. But the goal of RL is to arrive at an optimal policy. These methods could in principle be used to perform a complete policy evaluation step of a policy iteration algorithm, but this would be extremely slow and thus very impractical.

Two very recent algorithms have been introduced that begin to address solving the general RL problem using function approximators. The VAPS (Value and Policy Search) algorithm (Baird and Moore 1999) performs gradient descent on an error function that combines reinforcement and the value function, thus reducing error in both simultaneously. This algorithm is guaranteed to converge to a local minimum of its error function. However, this means that it does not necessarily converge to a locally optimal policy. In fact, it is only guaranteed to reach a locally optimal policy when the value function component is eliminated from the error function, in which case the algorithm reduces to Williams' (1992) REINFORCE algorithm.

More recently, Sutton et al. (1999) introduced a *Policy Gradient* framework for RL with function approximation, in which the policy is explicitly represented by a function approximator, independent of the value function. They prove that a version of policy iteration using this formulation converges to a locally optimal policy. Williams' (1992) REINFORCE algorithm falls within this class of algorithms as well.

Efficiency

While using function approximators greatly improves the scalability of RL algorithms when compared to using the look-up table representation, performance of the resulting algorithms still leaves a lot to be desired. This is because generalizing function approximators like multi-layer perceptrons, though very powerful learning mechanisms, are not compatible with the goal of on-line learning that RL promises to achieve because they typically require large numbers of iterations over data sets.

Tile coding systems are faster to learn than multi-layer perceptrons because they utilize the faster learning rule—least mean squares (LMS) vs. back propagation—and they lend themselves better to incremental learning. However, it remains the case that learning a value function for the entire state space is a slow process, even with the improvements introduced by the use of function approximators. For

example, Sutton (1996) and Sutton and Barto (1998) show that applying RL with tile coding function approximators (which are among the fastest-learning functions approximators available) on two-dimensional problems can require trials in the order of hundreds to tens of thousands before a good solution is found.

1.4.4 Variable Resolution Methods

There have been other attempts at addressing the limitations of current RL methods that take a rather different approach from what we have described so far. Some of these efforts focused on dividing the state space into relatively large partitions such that the action to take is based on the partition the current state is in, an idea dating back to the *BOXES* program (Michie and Chambers 1968). However, rigid, *a priori* partitioning of the state space requires much interaction on the part of the designer and can be very difficult especially in high dimensional problems.

An appealing way for economically representing a high-dimensional, continuous state space that avoids these problems is to dynamically partition it based on experience, resulting in varying resolution across the space, with high resolution concentrated only where it is needed. Moore (1990, 1991, 1994) presented a number of algorithms that try to achieve that. The most recent of these methods (Moore 1994; Moore and Atkeson 1995) is *parti-game*, which dynamically partitions the

state space using a *kd-tree* (Friedman, Bentley, and Finkel 1977), and associates high-level actions with each partition. It uses a game-theoretic approach in assigning costs to partitions and deciding when a partition needs to be split into smaller ones to increase the resolution. The algorithm was applied to a variety of control problems with varying dimensionality and it exhibited very good performance and scalability. Because of its promising features, we will dedicate the following chapter to introducing the algorithm.

1.5 Dissertation Summary

In Chapter 2, we introduce the parti-game algorithm and explain its functionality in detail. We will also introduce the different problems to which it has been applied and how its performance compares favorably with other RL algorithms. Chapter 3 shows that while parti-game is very efficient in solving a specific class of RL problems, its performance can be inconsistent and too dependent on the characteristics of the problem to which it is being applied. The chapter then introduces two successive enhancements to the algorithm that result in better overall performance and superior consistency in dealing with various problems. This results in a faster algorithm whose performance is more predictable.

Chapter 4 introduces a highly efficient implementation of the algorithm that

computes minimax cell costs, which sits at the core of the algorithms introduced in this dissertation. The ability of this algorithm to compute cell costs in the very efficient manner that it does facilitates the use of these algorithms in on-line settings which may require very fast decision-making ability from the control mechanism being used.

The algorithms introduced up to Chapter 4 do not necessarily find optimal form solutions. Chapter 5 introduces an iterative add-on algorithm which can be used with any member of the family of algorithms described in this dissertation to find better solutions than the solutions the algorithm in question has found. The chapter shows that this algorithm can be used to find optimal-form solutions in cases where one or more of the family of minimax-based algorithms presented here were only able to arrive at a solution of sub-optimal form.

Chapter 6 takes the algorithms introduced in this dissertation further in the direction of being independent of the existence of *a priori* knowledge about the domain to which they are being applied. It does so by eliminating the need for a designer-supplied local controller by its introduction of a fast, scalable, tile-coding-based mechanism with which effective local controllers can be learned on-line. The chapter shows that using this learning mechanism to learn local, greedy controllers does not have a big impact on performance compared to cases where an *a priori* controller is available.

Chapter 2

The Parti-Game Algorithm

Because it forms the basis of much of the work we present in this dissertation, this chapter is dedicated to introducing the parti-game algorithm in detail.

2.1 Introduction

Parti-game (PG) is a reinforcement learning algorithm introduced by Moore (1994) and Moore and Atkeson (1995) that operates on deterministic, continuous goal-type problems by dynamically partitioning the state space into hyper-rectangular cells of varying sizes, represented using a k-d tree data structure. It assumes the existence of a pre-specified greedy local controller that can be commanded to proceed from the current state to a given state. The algorithm uses a game-theoretic approach to

assign costs to cells based on past experiences using a minimax algorithm. A cell's cost can be either a finite non-negative integer or infinity. The former represents the number of cells that have to be traveled through to get to the goal cell and the latter represents the belief that there is no reliable way of getting from that cell to the goal. Cells with a cost of infinity are called *losing* cells while others are called *winning* ones.

The algorithm starts out with one cell representing the entire space and another, contained within it, representing the goal region. In a typical step, the local controller is commanded to proceed to the center of the most promising neighboring cell. Upon entering a neighboring cell (whether the one aimed at or not), or upon failing to leave the current cell within a timeout period, the result of this attempt[1] is added to the database of experiences the algorithm has collected, cell costs are recomputed based on the updated database, and the process repeats. The algorithm terminates upon entering the goal cell.

If at any point the algorithm determines that it can not proceed because the agent is in a losing cell, each cell lying on the boundary between losing and winning cells (termed the *win-lose boundary*) is split across the dimension in which it is

[1]The experience is a triple (x_1, x_2, x_3) representing the cell agent originated in, the cell aimed at and the one it ended up in, respectively.

largest and all experiences involving cells that are split are discarded. Since parti-game assumes, in the absence of evidence to the contrary, that from any given cell every neighboring cell is reachable, discarding experiences in this way encourages exploration of the newly created cells.

Parti-game is shown in Algorithm 1. $J_{WC}(i)$, the minimax cost for cell i, is defined as:

$$J_{WC}(i) = \begin{cases} 0 & \text{if } i = \text{GOAL} \\ 1 + \min\limits_{j \in \text{NEIGHS}(i)} \max\limits_{k \in \text{OUTCOMES}(i,j)} J_{WC}(k) & \text{otherwise} \end{cases} \quad (1)$$

where NEIGHS(i) is the set of cells adjacent to cell i, OUTCOMES(i, j) is all members of NEIGHS(i) which the agent entered as a result of an aiming attempt that started in cell i heading for cell j. If OUTCOMES(i, j) is the empty set, it is assumed to have the value $\{j\}$, which is the source of the algorithm's optimism when no experience exists. $J_{WC}(i)$ takes the cost $+\infty$ if the experiences the agent has show that there is no reliable way of getting from cell i to the goal. In other words, $J_{WC}(x)$ takes the value of ∞ if $x \in \text{MinMaxNeighbor}^+(x)$, the transitive closure of MinMaxNeighbor(x), where MinMaxNeighbor(x) is as defined in Algorithm 1:

$$\text{MinMaxNeighbor}(x) = \arg\min_{j' \in \text{NEIGHS}(x)} \max_{k \in \text{OUTCOMES}(x,j')} J_{WC}(k) \quad (2)$$

Algorithm 1 The Parti-Game algorithm.

Input: A kd-tree representing the current partitioning of the state space.
Input: A database containing some of the experiences seen thus far (used by the OUTCOMES function).
Outputs: The modified input kd-tree and experiences database.
$s \leftarrow$ InitialState
GOAL $\leftarrow$ GoalCell
while $s \notin$ GOAL **do**
  $i \leftarrow$ cell containing s
  Compute J_{WC}, for each cell
  if $J_{WC}(i) = +\infty$ **then**
    Partition those winning and losing cells that fall on the win-lose boundary and **remove** experiences associated with them from the database of experiences.
  else
    $j \leftarrow \text{MinMaxNeighbor}(i) = \arg\min_{j' \in \text{NEIGHS}(i)} \max_{k \in \text{OUTCOMES}(i,j')} J_{WC}(k)$
    $p \leftarrow$ center of cell j
    while Agent is not stuck, s is still in i **and** timeout did not expire **do**
      Actuate local controller towards p
      $s \leftarrow$ current state
    end while
    $k \leftarrow$ cell containing s
    add k to OUTCOMES(i, j)
  end if
end while

2.2 Performance

Moore and Atkeson (1995) compared the performance of parti-game to other algoritms that can solve discrete RL problems. Specifically, they compared PG's performance to Q-learning (Watkins 1989) and prioritized sweeping (Peng and Williams 1992, Moore and Atkeson 1994) as well as modified versions of the two algorithms. To allow these algorithms to operate on continuous environments, the problem was discretized into the coarsest possible uniform discretization that still allowed these algorithms to find a solution. In addition to the normal implementations of the two algorithms, modified versions were also used in which they were given information about where the goal is. This was done by initializing the cost-to-goal of any unexplored state transitions to the "Manhattan distance" to the goal. This is equivalent to PG's initial optimism where it believes that the transition between any two neighboring cells is possible, unless experience proves otherwise, which translates into the belief that the number of cells through which it has to travel from any cell to the goal cell is the Manhattan distance—at the cell level—from that cell to the goal.

Moore and Atkeson (1995) showed that PG outperformed all four algorithms in the number of backups performed and the number of steps taken in the world before the goal is reached, with one exception. The version of Q-learning that was

informed of where the goal is performed fewer backups than Moore and Atkeson (1995)'s implementation of PG, although it took more steps in the world before finding the goal.

In Chapter 4 we will present a minimax evaluation algorithm for the J_{WC} function—which sits at the core of PG and the algorithms introduced in this dissertation—that reduces the computational requirements of this class of algorithms considerably. The chapter will show that the number of backups performed for the same problem on which the comparison was performed in (Moore and Atkeson 1995) is dramatically reduced.

Moore and Atkeson (1995) also showed evidence that parti-game scales well with the number of dimesions of the problem. This was demonstrated by the fact that the number of partitions and the number of steps taken in the world did not grow exponentially with the growth of the number of dimensions.

2.3 Promise and Limitations

It could safely be said that one of the ultimate goals of reinforcement learning is finding a fast algorithm with which a *tabula rasa* agent could learn to behave optimally in any environment which generates a reinforcement signal and from which state information could always be obtained. By behaving optimally we mean that

the algorithm would ultimately be able act in the environment in a way that maximizes the expected reinforcement it receives from the environment over time. Such an algorithm should also have good scalability with high dimensional problems.

Parti-game's appeal stems from the fact that it is a fast RL algorithm that has been shown to scale well with the number of problem dimensions and which does not require a full model of the environment. Its strengths come from the very coarse resolutions at which it is able to find solutions, and the fact that these resolutions are dynamically increased based on need and only in relevant areas of the state space.

Still, parti-game has a number of shortcomings and limitations that stand in the way of its achieving this goal of RL:

1. While parti-game can perform very well against other RL algorithms, we have shown in (Al-Ansari and Williams 1998a, 1998b, 1999) that its performance can be inconsistent from one problem to another and could be very sensitive to the specific problem configuration. This brings into question its applicability to arbitrary problems.

2. It can converge to solutions that are only locally optimal.

3. Although it does not require a full model of the dynamics of the problem

to which it is being applied, it needs to be supplied with an *a priori* local controller. This means that it is not completely independent of *a priori* knowledge of the problem dynamics.

4. It only applies to problems that are:

   (a) goal-type

   (b) deterministic

   (c) continuous

2.4 Contributions of this Dissertation

In the following chapters, this dissertation introduces a family of algorithms, in the same spirit of parti-game, that find optimal-form solutions to deterministic, continuous-space, goal-type RL problems. These algorithms find solutions more quickly than parti-game, have good scalability and do not require an *a priori* local controller. Therefore, the family of algorithms introduced here improves upon items 1, 2 and 3 above, in addition to providing superior performance to parti-game's. Furthermore, this dissertation also presents two successive algorithms for computing minimax costs, which is central to these algorithms, that dramatically reduce the computational overhead of the algorithms and, therefore, make them even better

suited for on-line learning and control and give the better scalability. However, these algorithms are still restricted to goal-type, deterministic problems. Extending them to more general RL problems remains a challenge and will be discussed further in Chapter 7.

Chapter 3

Efficient On-Line Variable Resolution Algorithms

As mentioned in Chapter 2, although parti-game is a very efficient algorithm for solving a specific class of RL problems, the consistency of its performance can be dependent on the particular problem configuration to which it is applied. This chapter introduces two successive algorithms that improve on parti-game. The first has better overall performance than parti-game, but does not specifically address the inconsistent performance problem. The second algorithm directly addresses this inconsistency, making its performance considerably more predictable.

3.1 Partitioning Strategies

Before introducing the modified algorithms, we first take a closer look at parti-game's partitioning strategy. We define a partitioning strategy as follows.

Definition 1 *For our purposes, an algorithm's partitioning strategy is defined by (a) its partitioning-triggering condition which determines when partitioning is deemed necessary, (b) how it selects critical regions, i.e., regions of the state space in which new cells need to be introduced when the triggering condition is satisfied, and (c) which cells from those critical regions to actually partition.*

Under this definition, parti-game's partitioning strategy can be defined as follows.

a. **Triggering condition: when the agent enters a losing cell.** Thus parti-game actually postpones partitioning until it becomes absolutely necessary, because the agent can not make any informed decisions in a losing cell without performing further partitioning.

b. **Critical regions: all cells on the win-lose boundary.** The win-lose boundary defines barriers the agent perceives that prevent it from reaching the goal. Defining the critical region as all cells that fall on that boundary guarantees covering the whole region in which partitioning could have any

hope of finding an opening in those barriers by searching at the higher resolutions. At the same time, this set of cells is the smallest subset of the state space (representable in cells) that contains all states that fall on either side of every perceived barrier.

c. **Cells to partition: every cell in a critical region.** With this, parti-game hopes to maximize the chances of finding an opening through perceived barriers, if one could be found at the resulting resolution. By partitioning on both sides of the win-lose boundary, parti-game guarantees that neighboring cells along the boundary remain close in size. Along with the strategy of aiming towards centers of neighboring cells, this produces pairings of winner-loser cells that form proposed "corridors" for the agent to try to go through to penetrate the barrier it perceives.

3.1.1 Effects on Exploration

The algorithms that will be introduced in this chapter alter the partitioning strategy of parti-game in different ways. Before we introduce these algorithms and study how their performance differs from that of parti-game, we first examine the basic, low-level implications of changing partitioning strategies.

The manner in which parti-game explores the state space is closely tied to the

partitioning strategy it employs. Since partitioning some regions in the state space more finely facilitates finding openings in perceived domain barriers in those regions, employing different partitioning strategies may lead to finding different openings or following different trajectories in the process of searching for ones. The two following points show specific ways in which this could happen.

- **Same region, different partitionings.** Two algorithms employing different partitioning strategies may select the same region of the state space for further partitioning but partition it in two different ways. Since the agent moves in the space by choosing states to aim for in the cells it wants to enter, partitioning a region of a state space in different ways implies that the agent may select different cells to aim at, which would clearly result in following different trajectories, i.e., the space will be explored differently.

- **Different regions partitioned.** Two algorithms employing different partitioning strategies may select different regions of the state space for further partitioning when the agent finds itself in a losing cell. Since re-exploration occurs only in newly partitioned regions, this means that different regions would be explored by the two algorithms.

Thus, altering parti-game's partitioning strategy has a direct effect on the exploration strategy. In what follows we will introduce two algorithms that change PG's partitioning strategy—and, thus, explore the state space differently—and study their performance and how it compares with PG's.

3.2 Partitioning Only Losing Cells

Consider parti-game's partitioning strategy mentioned above. The triggering condition for partitioning is when the agent finds itself in a losing cell, and the critical region selected is comprised of all cells that fall on the win-lose boundary. This means that partitioning can only be triggered with the agent on the losing side of the win-lose boundary. That being the case, reducing the subset of critical region cells to partition to only those that are losing at the time partitioning is triggered would still give the agent the same kind of access to the boundary through the newly formed cells. However, this would result in a size disparity between winner- and loser-side cells and, thus, would not produce the winner side of the pairings mentioned above. To produce a similar effect to the pairings of parti-game, we change the aiming strategy of the algorithm. Under the new strategy, when the agent decides to go from the cell it currently occupies to a neighboring one, it aims towards a state in this neighboring cell that is just inside the center point of the common

surface between the two cells. While this does not reproduce the trajectories of the original aiming strategy exactly, it achieves a very similar objective.

Parti-game's good scalability stems from its variable resolution strategy, which partitions finely only in regions where it is needed. By limiting partitioning to losing cells only, we try to increase the resolution in even fewer parts of the state space and thereby make the algorithm even more efficient.

Algorithm 2 shows the loser-partitioning parti-game algorithm (LPPG). Notice that the only change between this algorithm and parti-game is that only losing cells are split in step 10. Thus the partitioning strategy of LPPG changes part (c) of parti-games's partitioning strategy: which of the critical region's cells to partition.

3.2.1 Problem Domains

Before we can introduce experimental results comparing LPPG and parti-game, we first introduce the set of problem domains that the algorithms were applied to. This set of problems will be used as a benchmark throughout this dissertation.

Continuous Mazes

The first and simplest domain used is that of continuous, two-dimensional mazes, in which the agent can move freely, at a constant speed and in any direction, except

Algorithm 2 loser-partitioning parti-game.

1: Input: A kd-tree representing the current partitioning of the state space.
2: Input: A database containing some of the experiences seen thus far (used by the OUTCOMES function).
3: Outputs: The modified input kd-tree and experiences database.
4: $s \leftarrow$ InitialState
5: GOAL $\leftarrow$ GoalCell
6: **while** $s \notin$ GOAL **do**
7: $i \leftarrow$ cell containing s
8: Compute J_{WC}, for each cell
9: **if** $J_{WC}(i) = +\infty$ **then**
10: Partition the losing cells that fall on the win-lose boundary and discard experiences associated with them
11: **else**
12: $j \leftarrow \text{MinMaxNeighbor}(i) = \arg\min_{j' \in \text{NEIGHS}(i)} \max_{k \in \text{OUTCOMES}(i,j')} J_{WC}(k)$
13: $p \leftarrow$ a state in j that is at a distance $\alpha \times \text{MinDimension}(j)$ from both the shared hyper-surface between i and j and the center of this hyper-surface, where $\alpha \ll 1$ and MinDimension(j) is the smallest hyper-surface width of cell j
14: **while** not stuck, s is still in i **and** timeout did not expire **do**
15: Actuate local controller towards p
16: $s \leftarrow$ current state
17: **end while**
18: $k \leftarrow$ cell containing s
19: add k to OUTCOMES(i, j)
20: **end if**
21: **end while**

when a barrier blocks its path. A set of four such mazes, shown in Figure 1, was used. The agent's task in each of these is to start at the point marked Start and reach the square goal region.

The actions available to the agent are the continuous set $[0, 2\pi)$, which enable the agent to move in any direction in the maze. The state change of the agent given an action, u, and a time step, δt, is described by:

$$x_t = x_{t-1} + \delta t \cos u$$

$$y_t = y_{t-1} + \delta t \sin u$$

When a barrier blocks the agent's path, its motion is halted at the point of intersection between the line of motion it has been following and the barrier. The mazes we use have the size of 10×10 units and the value of δt used is 0.1. Therefore, it would take the agent 100 steps to go from one end of the maze to the opposite one, when traveling parallel to one of the axes.

The local controller we used simply determines the angle between the horizontal and the line from the current state to the target state, at every time step, and applies it as its action.

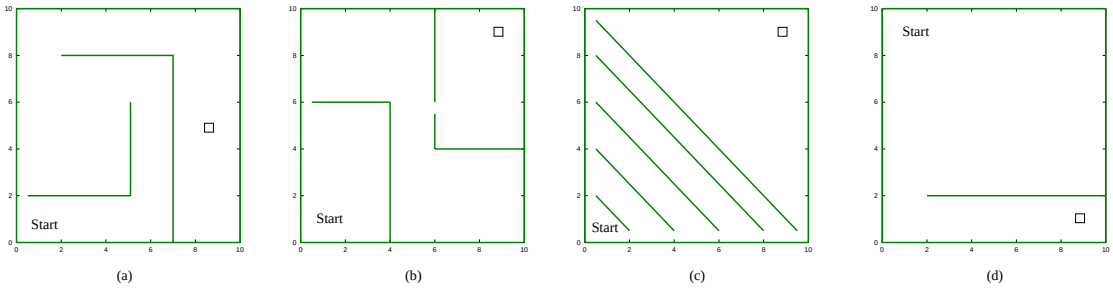


Figure 1: A set of continuous mazes.

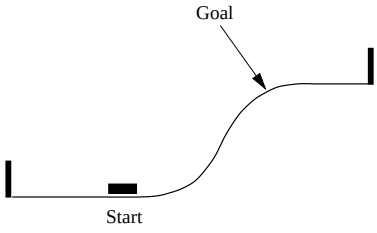


Figure 2: An ice puck on a hill.

Puck on a Hill

The puck-on-a-hill, shown in Figure 2, is a puck on a frictionless surface. It can thrust horizontally to the left and to the right with a maximum force of 1 Newton. The state space is two-dimensional, consisting of the horizontal position and velocity. The agent starts at the position marked Start at velocity zero and its goal is to reach the position marked Goal at velocity zero. Maximum thrust is not adequate to get the puck up the ramp from the Start state, so it has to learn to move to the left far enough such that when it heads back to the right with full thrust, it would be able to build enough momentum to allow it go up the hill. The puck is reset to the Start position whenever it hits either of the barriers at the top and bottom of the slope.

The puck weighs 1kg. The horizontal dimension, position x, has a range of $[-20, 20]$ and the vertical dimension, velocity v, ranges within $[-13, 7]$. The start state is $(0, 0)$ and the goal is $(16, 0)$. The surface the agent moves on is flat for $x \leq 0$ with height $h = 0$, but for $x > 0$, $h = 1 - \cos(\frac{\pi}{20} x)$. The equations used to update the state of the puck given a thrust at time t, T_t, and a time step, δt, are:

$$x_t = x_{t-1} + \delta t \, v_t$$

$$v_t = v_{t-1} + \delta t \, a_t$$

Where,

$$a_t = \frac{T_t - 9.8\, m_{t-1}}{1 + m_{t-1}^2}$$

and

$$m_t = \begin{cases} \frac{\pi}{20} \sin\left(\frac{\pi}{20}\, x_{t-1}\right) & \text{if } x_{t-1} > 0 \\ 0 & \text{otherwise} \end{cases}$$

We used a δt of 0.1 in our simulations.

The local controller we used determines the thrust to be used at time t, T_t, as follows:

$$T_t = \begin{cases} \frac{\theta}{\theta_c}\, T_m & \text{if } |\theta| \leq \theta_c \\ \frac{\pi - \theta}{\theta_c}\, T_m & \text{if } |\pi - \theta| \leq \theta_c \\ \text{sign}\left(v_{\text{target}} - v_{\text{current}}\right) T_m & \text{otherwise} \end{cases}$$

where θ is the angle of the vector from the current state to the target state ($s_{\text{target}} - s_{\text{current}}$), $\theta_c = \frac{\pi}{12}$ is a constant *cruising* angle, $T_m = 1$ is the absolute value of the two bang-bang thrust values (-1 and 1) and v_{current} and v_{target} are the current and target velocities, respectively.

This defines a bang-bang controller, but with a special case. In the general case, a thrust of +1 is used if the target velocity is higher than the current one, and a thrust of -1 is used if the target velocity is lower than the current velocity. However, if $|\theta|$ or $|\pi - \theta|$ are within the cruising angle of $\frac{\pi}{12}$, we use the appropriate fraction of the limit thrust that would slowly move the agent in the direction of the target

state.

Nine-Joint Arm

A nine degree of freedom, snake-like arm that moves in a plane and is fixed at one tip, as depicted in Figure 3. The objective is to move the arm from the start configuration to the goal one, which requires curling and uncurling to avoid the barrier and the wall.

The dynamics of this system and the local controller used are described in detail in Moore and Atkeson (1995). The system is kinematic with the state taking the form $s = (\theta_1, \ldots, \theta_9)$, denoting the set of joint angles. θ_1 is the angle between the first joint and the horizontal axis at the fixed tip. For θ_i where $1 < i \leq 9$, the angle denoted is that between joints $i-1$ and i. The actions used are of the form $a = (\delta_{\theta_1}, \ldots, \delta_{\theta_9})$, where each δ_{θ_i} represents the desired displacement in the i^{th} dimension. The state transition function given the action a is:

$$s_{t+1} = \begin{cases} s_t + a & \text{If the motion of the arm from } s_t \text{ to } s_t + a \text{ would not result} \\ & \text{in an intersection with any barriers or any of its parts.} \\ s_t & \text{Otherwise.} \end{cases}$$

Actions have the restriction that $|a| \leq \frac{\pi}{15}$.

The action the local controller uses at time step t is $\lambda(s_{\text{target}} - s_{t-1})$, where

$\lambda = \min(\frac{\pi}{15|(s_{\text{target}} - s_{t-1})|}, 1)$. The start state we used is

$$s_0 = (-\frac{3}{4}\pi, 0.02, -0.02, \ldots, -0.02)$$

and the goal state is $s_G = -s_0$, that is, $(\frac{3}{4}\pi, -0.02, 0.02, \ldots, 0.02)$. We chose the goal region to have the goal state at its center, with 0.075 added on either side. That is, if $s_G = (\theta_{G_1}, \ldots, \theta_{G_9})$, the goal region is defined by the nine intervals, $[\theta_{G_i} - 0.075, \theta_{G_i} + 0.075]$, $i = 1 \ldots 9$.

The Acrobot

Shown in Figure 4, a two-link robotic arm that is hinged at the top tip and under-actuated at the joint between the two links. The state is four-dimensional, $(\theta_1, \dot{\theta_1}, \theta_2, \dot{\theta_2})$. The task, which was introduced by Sutton (1996), is to get the bottom tip to cross the goal line in the shortest time possible, starting from the resting position. Boone (1997) also studies this problem in some detail and addresses minimum-time control of the acrobot.

The equations of motion of the acrobot are as follows (Sutton 1996):

$$\ddot{\theta}_1 = -d_1^{-1}(d_2\ddot{\theta}_2 + \phi_1)$$

$$\ddot{\theta}_2 = \left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1} \left(\tau + \frac{d_2}{d_1}\phi_1 - \phi_2\right)$$

$$d_1 = m_1 l_{c1}^2 + m_2(l_1^2 + l_{c2}^2 + 2 l_1 l_{c2} \cos\theta_2) + I_1 + I_2$$

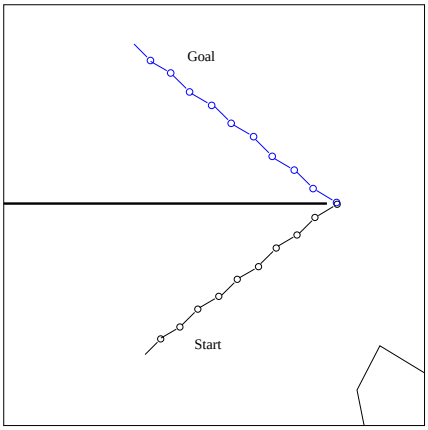


Figure 3: A snake-like 9-joint robot arm.

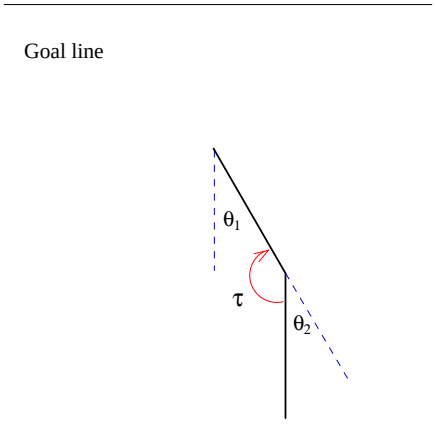


Figure 4: The Acrobot.

$$d_2 = m_2(l_{c2}^2 + l_1 l_{c2} \cos\theta_2) + I_2$$

$$\phi_1 = -m_2 l_1 l_{c2} \dot{\theta_2}^2 \sin\theta_2 - 2m_2 l_1 l_{c2} \dot{\theta}_2 \dot{\theta}_1 \sin\theta_2 + (m_1 l_{c1} + m_2 l_1)\, g \cos(\theta_1 - \pi/2) + \phi_2$$

$$\phi_2 = m_2 l_{c2}\, g \cos(\theta_1 + \theta_2 - \pi/2)$$

where $\tau \in \{+1, 0, -1\}$ is the torque applied in the middle joint, $\dot{\theta}_1$ and $\dot{\theta}_2$ are the angular velocities bounded by $\dot{\theta}_1 \in [-4\pi, 4\pi]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$, $m_1 = m_2 = 1$ are the masses of the two links, $l_1 = l_2 = 1$ are the lengths of the two links, $l_{c1} = l_{c2} = 0.5$ are the lengths to the center of mass of the links, $I_1 = I_2 = 1$ are the moments of inertia of the links and g is the gravity constant ($9.8\,\mathrm{m/sec^2}$). Following Sutton (1996), we used a time increment of 0.05 seconds but torques were applied for four time increments.

Because the goal region in this problem can not be completely specified as a hyper-rectangular region, we used only a subset of the goal region that can be so specified. The goal region was defined by $\theta_1 \in [2\pi/3, \pi]$, $\dot{\theta}_1 \in [-4\pi, 4\pi]$, $\theta_2 \in [0, \pi/2]$ and $\dot{\theta}_2 \in [-9\pi, 9\pi]$. This makes this problem slightly more difficult than the version studied by Sutton (1996) and Boone (1997). We will discuss this in further detail when presenting the experimental results. For our test purposes, we used a greedy local controller that performs one-step lookahead at every time step using the actual model to pick the action that would take it closest to the target in Euclidean distance. The allowable torque values are $\{+1, -1, 0\}$.

In Chapter 6 we will see how the *a priori* controllers defined here could be replaced by learning controllers so that there would be much less dependence on domain knowledge.

3.2.2 Experimental Results

We now present experiments comparing PG and LPPG on the problem domains described above. An experiment starts with exactly two partitions, one comprising the whole state space and another, contained within the first, representing the goal region. Then, one or more trials are performed. Each trial starts by placing the agent in the start state and applying the given algorithm until it terminates, i.e., until the agent enters the goal cell. The partitioning, experiences and (therefore) cell costs are preserved from each trial to the next. The experiment terminates when the solution stabilizes, that is, when the latest trial does not yield any new experiences. Not receiving new experiences means that all agent actions at every step will be identical from one trial to the next since the agent would be acting on identical information.

Figure 5 depicts the partitioning and final trajectories after a stable solution is found when applying PG and LPPG Maze a of Figure 1(a), the puck on a hill and the nine-joint arm problems. The figure shows that the effect of partitioning

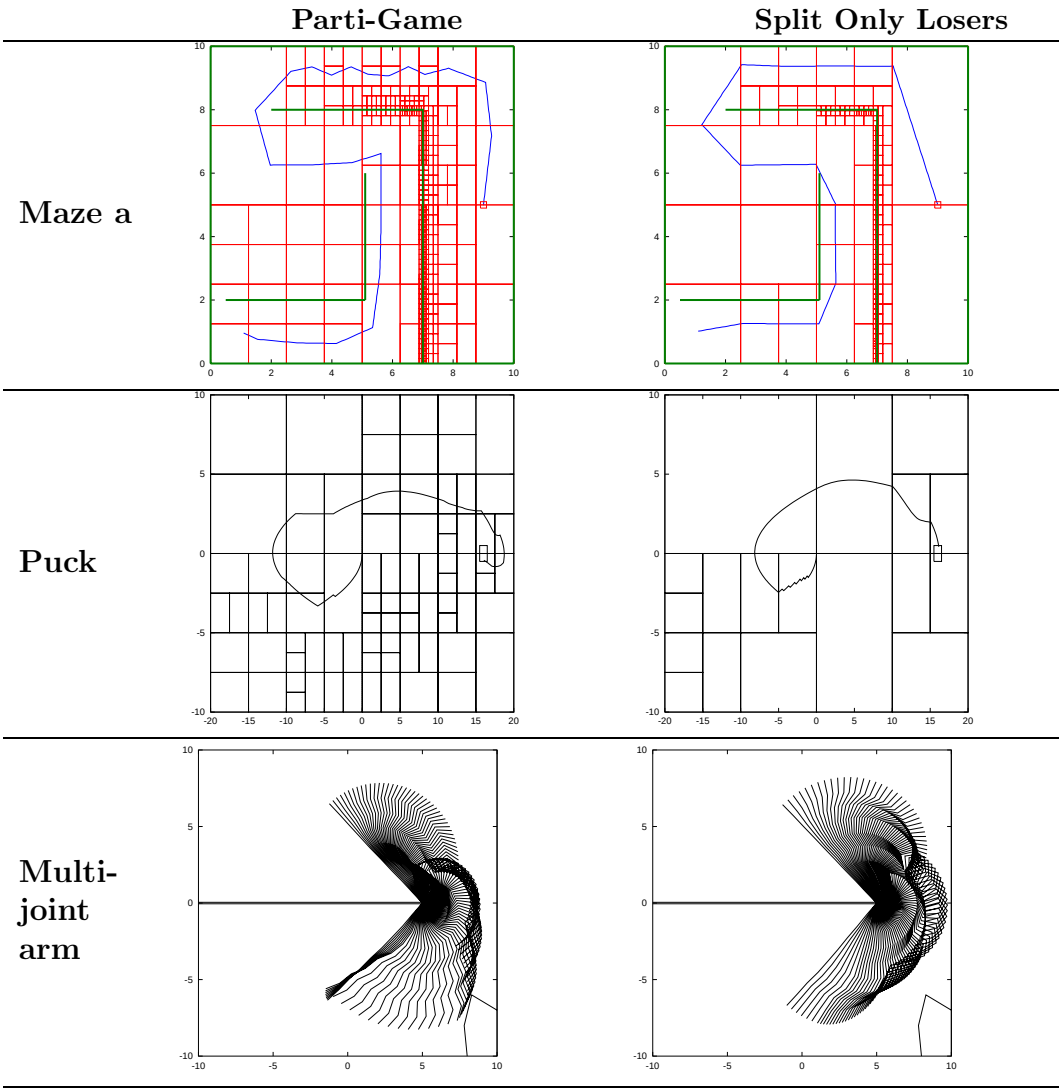


Figure 5: Results of applying parti-game and loser-partitioning parti-game to maze a, the puck-on-a-hill and the multi-joint arm problems.

Problem	Algorithm	Trials	Partitions	Total Steps	Final Trajectory Steps
maze a	PG	**3**	444	35,131	279
	LPPG	**3**	**239**	**16,652**	**256**
maze b	PG	6	98	**5,180**	183
	LPPG	**5**	**76**	7,187	**175**
maze c	PG	3	176	**7,768**	416
	LPPG	**2**	**120**	10,429	**165**
maze d	PG	**2**	1,194	553,340	**149**
	LPPG	**2**	**350**	**18,639**	155
puck	PG	6	80	6,764	240
	LPPG	**2**	**18**	**3,237**	**151**
nine-joint arm	PG	36	117	7,112	103
	LPPG	**13**	**52**	**5,499**	**101**

Table 1: Results of applying parti-game (PG) and loser-partitioning parti-game (LPPG) to four mazes, the puck-on-a-hill and the nine-joint arm problems. Smaller numbers are better and are shown in **bold.**

only on the losing side is that partitioning becomes even more localized than that produced under parti-game. Table 1 shows complete results for applying the two algorithms to all four maze problems, the puck on a hill and the nine-joint arm. Because of problems in reaching stable solutions, the Acrobot results can not be compared in this table and, therefore, will be presented separately in section 3.3.4. The table compares the number of trials needed, the number of partitions, total number of steps taken in the world (by the local controller) and the length of the final trajectory measured in the number of such steps. The table shows that the

new algorithm indeed resulted in fewer total partitions in all problems. It also improved in all but one of the problems in the length of the final trajectory. It improved on parti-game in the number of trials needed to stabilize in four out of the six problems, while it resulted in the same number of trials in the remaining two. It resulted in fewer total steps taken in the world in the four of the six problems, however, that number increased in the remaining two.

3.3 Balanced Partitioning

Upon close observation of PG's performance on Maze a in Figure 5, we see that parti-game partitioned too finely along the right wall of the upside-down-L-shaped barrier compared to its top wall. Although it generated fewer overall partitions, the figure shows that LPPG produced the same type of behavior when it only partitioned on the losing side of the win-lose boundary.

To help illustrate this sort of behavior more clearly, we shift our attention to the even simpler maze first introduced in Figure 1(d), which is repeated in Figure 6 for convenience. Figures 7(a) and (b) show the partitioning required by PG and LPPG, respectively. Figure 7(a) shows that parti-game has an extremely hard time reaching the goal in this maze, requiring 1,194 partitions and 553,191 steps in the first iteration before it ever reaches the goal! After it finds the goal once, it can

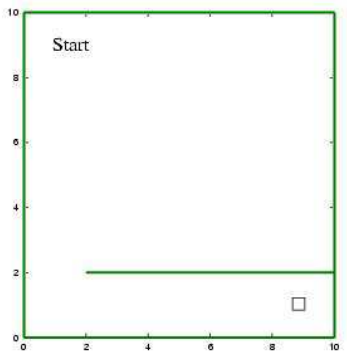
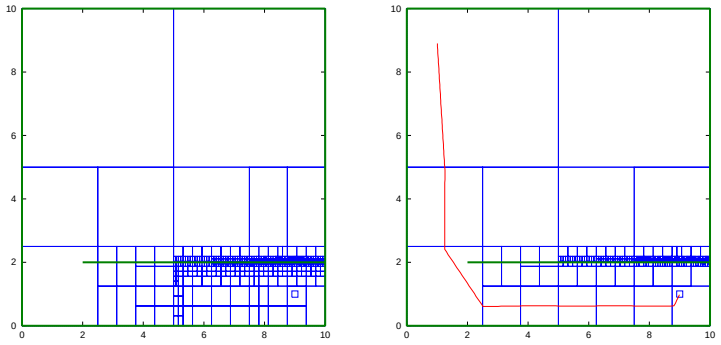


Figure 6: A very simple maze, first introduced in Figure 1(d).



(a) (b)

Figure 7: Partitioning produced by (a) parti-game and (b) parti-game with partitioning only losing cells for the very simple maze introduced in Figure 1(d).

get there in 149 steps, for a total of 553,340 steps needed before converging to the solution. Figure 7(b) shows the solution arrived at by partitioning only on the losing side of the win-lose boundary. While it improves the solution greatly, requiring only 350 partitions and 18,639 total steps, this still is far too much work for such a simple task.

3.3.1 Reason for the Imbalance

As we have seen in Figures 7(a) and (b), partitioning along the one barrier in this maze is very uneven, being extremely fine near the goal and growing coarser as the distance from the goal increases.

The reason behind this is that the agent under PG and LPPG effectively orders its exploration of the frontier cells along a barrier based on the distance from where it currently is to the goal, giving higher priority to cells with lowest cost. When an attempt to go through the lowest cost cell fails, the next best cell is tried. It often happens, however, that in trying to go through the most promising cell and discovering that it is blocked and it is thus given an infinite cost, other cells that had good costs now also look bad. This prevents the agent from trying these cells and results in it not moving toward the areas where these cells lie. Consequently, the agent searches only a limited part of the frontier actually made available after

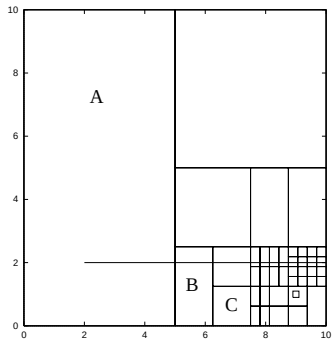


Figure 8: The partitioning of the maze of Figure 1(d) after 829 steps and 10 rounds of partitioning.

each partitioning episode. In addition, this means that these untried cells, which are now losing cells, do not fall on the win-lose boundary and thus do not get partitioned immediately.

Figure 8 shows an example of just how slowly parti-game's partitioning reaches the left half of the maze, which is vital to reaching the goal. After 829 steps in the world and 10 rounds of partitioning, the space is partitioned as seen in the figure. The agent, which is currently in cell B, still has not re-entered cell A, since it left it at the very beginning of the trial when it headed directly from the start state toward the goal region. Partitioning along the barrier is very unbalanced in favor of the area near the goal. Under this partitioning, the agent will try to go from cell B to cell C and, when it fails, A along with B will be considered losing and therefore the agent will see no need to enter A. Cell B will now be on the win-lose

boundary, but A will not, so A will still not get partitioned in the next partitioning phase, although the agent will enter it for the first time.

Putting higher focus on places where the highest gain could be attained if an opening is found can be a desirable feature, however what happens in cases like this one is obviously excessive.

3.3.2 Algorithm for Balanced Partitioning

One of the factors contributing to this problem of continuing to search for a shortcut at ever-higher resolutions in the part of the barrier nearest the goal is that both PG and LPPG search for solutions using an implicit trade-off between the shortness of a potential solution path and the resolution required to find this path. Only when the resolution becomes so fine that the number of cells through which the agent would have to travel as it uses this potential shortcut exceeds the number of cells to be traversed when traveling around the barrier does the algorithm finally go toward the actual opening.

A conceptually appealing alternative to bias this search is to maintain a more explicit coarse-to-fine search strategy. One way to do this is to try to keep the smallest cell size the algorithm generates as large as possible. In addition to achieving the balance we are seeking, this would tend to lower the total number of partitions

and result in shallower tree structures needed to represent the state space, which, in turn, results in higher efficiency.

To achieve these goals, LPPG was modified such that whenever partitioning is required, instead of partitioning all losing cells, we only partition those among them that are of maximum size. This has the effect of postponing splits that would lower the minimum cell size as long as possible. Algorithm 3 shows this largest loser-partitioning parti-game algorithm (LLPPG) where only the largest of the losing cells on the win-lose boundary are split when the agent finds itself in a losing cell and splitting is required.

3.3.3 Experimental Results

We now present experimental results for applying LLPPG to the our set of benchmark problems. Figure 9 is an expanded version of Figure 5 that adds the results of LLPPG to the previous results which are repeated for comparison. The first row in the figure compares results of applying all algorithms to the maze of Figure 1(a). In contrast to the two other algorithms depicted in the same figure, we can see that the LLPPG partitions very uniformly around the barrier. In addition, it requires the fewest number of partitions and total steps out of the three algorithms. Figure 10 shows the results of the two iterations needed by LLPPG when applied to

Algorithm 3 largest loser-partitioning parti-game.

Input: A kd-tree representing the current partitioning of the state space.
Input: A database containing some of the experiences seen thus far (used by the OUTCOMES function).
Outputs: The modified input kd-tree and experiences database.
$s \leftarrow$ InitialState
GOAL $\leftarrow$ GoalCell
while $s \notin$ GOAL **do**
  $i \leftarrow$ cell containing s
  Compute J_{WC}, for each cell
  if $J_{WC}(i) = +\infty$ **then**
    Partition the largest losing cells that fall on the win-lose boundary and discard experiences associated with them
  else
    $j \leftarrow \text{MinMaxNeighbor}(i) = \underset{j' \in \text{NEIGHS}(i)}{\arg\min} \underset{k \in \text{OUTCOMES}(i,j')}{\max} J_{WC}(k)$
    $p \leftarrow$ a state in j that is at a distance $\alpha \times \text{MinDimension}(j)$ from both the shared hyper-surface between i and j and the center of this hyper-surface, where $\alpha \ll 1$ and MinDimension(j) is the smallest hyper-surface width of cell j
    while not stuck, s is still in i **and** timeout did not expire **do**
      Actuate local controller towards p
      $s \leftarrow$ current state
    end while
    $k \leftarrow$ cell containing s
    add k to OUTCOMES(i, j)
  end if
end while

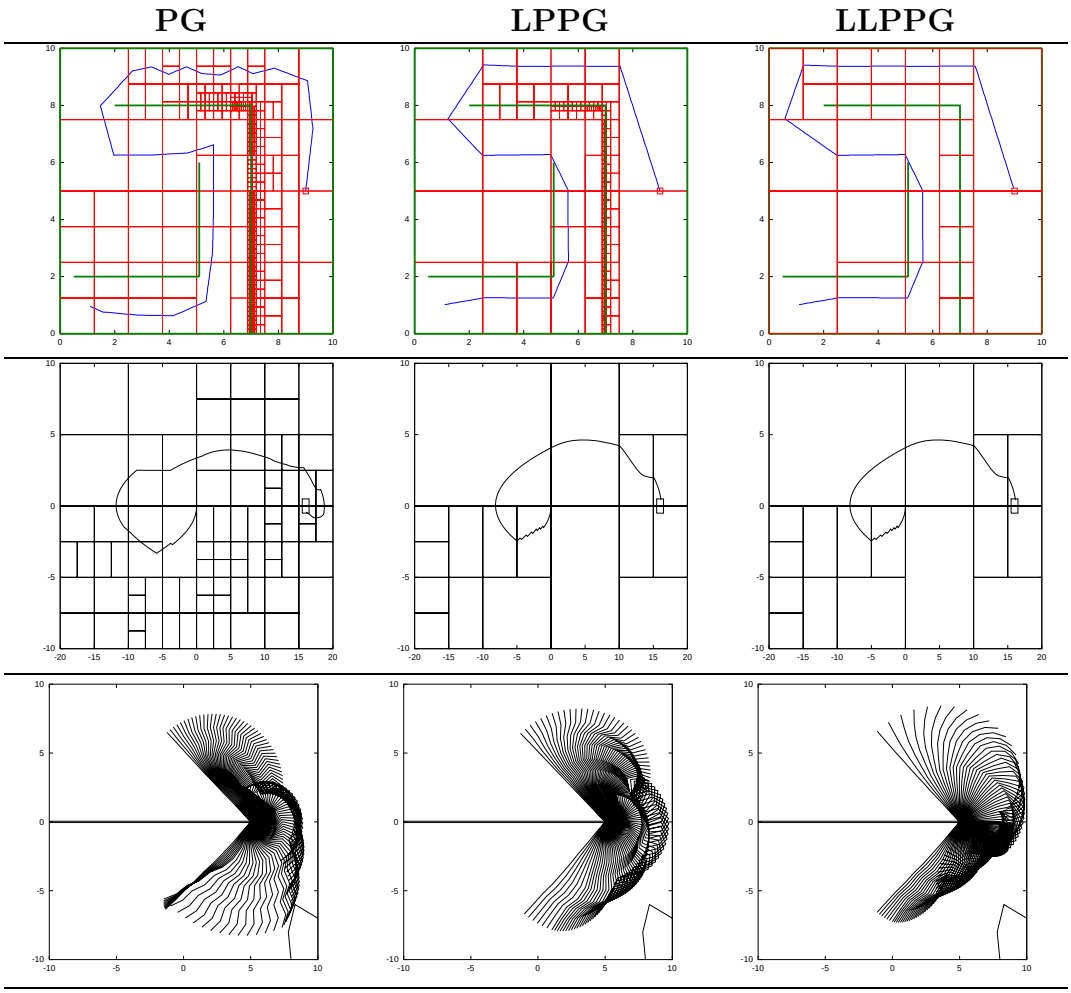


Figure 9: Results of applying PG, LPPG and LLPPG to maze a, the puck on a hill and the multi-joint arm problems.

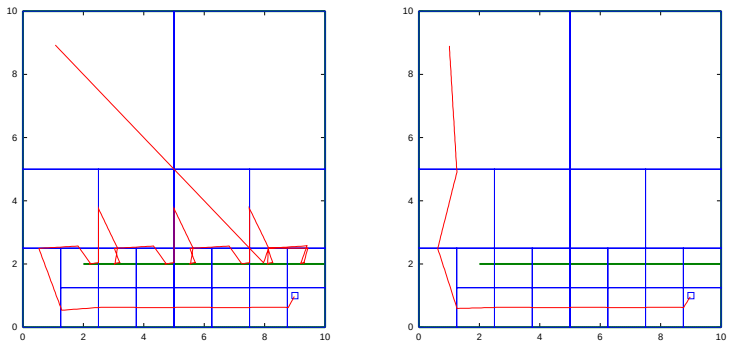


Figure 10: The result of partitioning largest cells on the losing side in the maze introduced in Figure 1(d). Only two trials are required to stabilize. The first requires 1304 steps and 21 partitions. The second trial adds no new partitions and produces a path of only 165 steps.

the especially problematic maze introduced in Figure 7. The agent traverses the barrier in a systematic way, uniformly partitioning around it until the opening is found. This algorithm vastly outperforms PG and LPPG in this case, requiring far fewer steps and partitions.

Table 2 also summarizes all the results of PG, LPPG and LLPPG. Again, it duplicates the previous results for comparison reasons. The table shows that LLPPG improves on parti-game's performance even further. It outperforms LPPG in four problems in the total number of partitions required, while it ties it in the remaining two. It outperforms LPPG in total steps taken in five problems and ties it in one—although PG still beats both of our new algorithms in this criterion for maze b. It

improves on the number of trials needed to stabilize in one problem, ties LPPG in four cases and ties PG in the remaining one. In the length of the final trajectory, LLPPG does better in one case, ties LPPG in two cases and does worse in three. This latter result is due to the generally larger partition sizes that result from the lower resolution that this algorithm produces. However, the increase in the number of steps is very minimal in all problems.

PG's *vs.* LLPPG's Exploration Strategies

As we saw in the preceding discussion, LLPPG performs far better than PG on Maze d. Because this maze is so simple yet elicits drastically differing behaviors from PG and LLPPG, we will use it as a basis for analyzing the behavior of the two algorithms. Let us imagine a parameterized version of this maze in which the size and location of one opening in an otherwise completely solid barrier can be varied such that it can be placed anywhere along the barrier and its width can be set to any value between zero and the width of the entire barrier.

Obviously, the worst case problem for any of the two algorithms would be one in which the width of the opening in the barrier is very small. Although the width of the opening can, in theory, be arbitrarily small, practical aspects of the problem and the implementation (e.g., the size of the agent and the accuracy of the computer

Problem	Algorithm	Trials	Partitions	Total Steps	Final Trajectory Steps
maze a	PG	**3**	444	35,131	279
	LPPG	**3**	239	16,652	**256**
	LLPPG	**3**	**27**	**1,977**	270
maze b	PG	6	98	**5,180**	183
	LPPG	**5**	**76**	7,187	175
	LLPPG	6	**76**	5,635	**174**
maze c	PG	3	176	7,768	416
	LPPG	**2**	120	10,429	**165**
	LLPPG	**2**	**96**	**6,803**	**165**
maze d	PG	**2**	1,194	553,340	**149**
	LPPG	**2**	350	18,639	155
	LLPPG	**2**	**21**	**1,469**	165
puck	PG	6	80	6,764	240
	LPPG	**2**	**18**	**3,237**	**151**
	LLPPG	**2**	**18**	**3,237**	**151**
nine-joint arm	PG	36	117	7,112	103
	LPPG	13	52	5,499	**101**
	LLPPG	**4**	**39**	**3,229**	110

Table 2: Results of applying PG, LPPG and LLPPG on three of the problem domains. Smaller numbers are better. Best numbers are shown in **bold.**

simulation) will dictate a minimum width for the opening. Having fixed such a minimum, finding the worst case problem for each algorithm becomes a matter of determining the location of this smallest opening along the barrier.

Based on the exploration pattern PG followed in Maze d, we can predict that the worst case for PG is one in which the very small opening is positioned at the left end of the barrier. Because of the extreme bias PG has toward finding would-be openings close to the goal, and for the reasons explained in section 3.3.1, locating the tiny opening at the left end of the wall would increase the difficulty of the problem for PG by orders of magnitude. By the same token, however, if the tiny opening is located near the goal region, PG would find it very quickly *because* of its bias.

As for LLPPG, since it searches the barrier systematically and uniformly at gradually increasing resolutions until the opening is found, its worst case behavior is much more dependent on the size of the opening than its location. Having fixed the size of the worst case opening, as soon as the uniform partitioning along the barrier reaches a resolution that allows the agent to reliably pass through the opening, the agent is guaranteed to have found the opening by the end of its sweep of the wall. Thus, while LLPPG would vastly outperform PG in any setup in which the opening is on the left side of the barrier (at a rate that increases very quickly

as the opening situated at the left edge of the maze gets smaller), it would perform much worse than PG in cases where the opening is near the goal, because it would search along the whole barrier at successively higher resolutions until the proper resolution across the whole barrier is reached before it can find the opening.

3.3.4 Acrobot Results

We found that all three algorithms discussed so far were not able to reach a stable solution when applied to the Acrobot problem. Although good solutions are found within only one or two trials, the algorithms continued to add more partitions and, consequently, did not stabilize at any of the good solutions they found. Since old experiences are discarded when partitioning takes place, the algorithms keep fluctuating between good and bad solutions. Figures 11 and 12 respectively show the number of seconds needed to reach the goal and the total number of partitions at the end of every trial by each of the three algorithms.

A good reference point for assessing the quality of the solutions arrived at by these algorithms is found in (Boone 1997), which studied minimum-time control for this acrobot task. The best solution obtained by Boone (1997) has a length of 61 steps, each 0.2 seconds long, which translates to a solution 12.2 seconds long. Although, as we mentioned earlier, our task is a bit more challenging than the

task he studied, and despite the relatively very coarse resolutions they generate, all three algorithms were repeatedly able to arrive at solutions that are around 20 seconds or less, even within the first few trials, when the partitioning is especially coarse. Therefore, interestingly, if the algorithms were made to stop collecting experiences after the first time a good solution is reached (effectively fixing their policies), they would stop producing new partitions and, thus, repeatedly achieve that good solution on every subsequent trial.

An interesting point Figures 12 shows is that although none of the algorithms are able to stabilize on a solution, the number of partitions increases more slowly when using LPPG, and even slower with LLPPG. Although this did not seem to have an effect on the stability of the solution, it is encouraging to see that LLPPG and, to a lesser extent, LPPG are able to cut down so dramatically on the growth of resource requirements, which suggests that they would be better able to scale up as the number of dimensions and/or partitions generated rises.

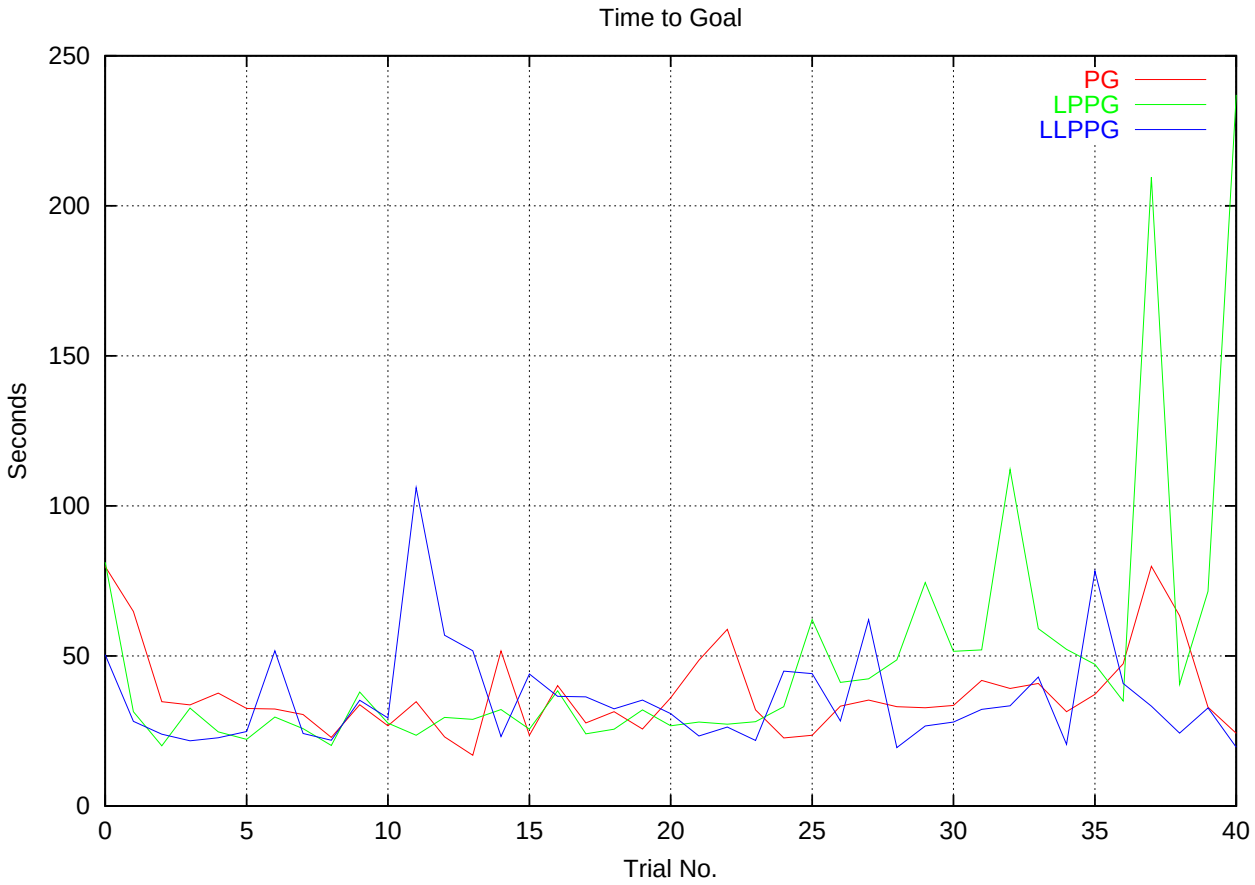


Figure 11: Graph showing the number of seconds needed to reach a solution by each of the algorithms as a function of trial number when applied to the Acrobot problem.

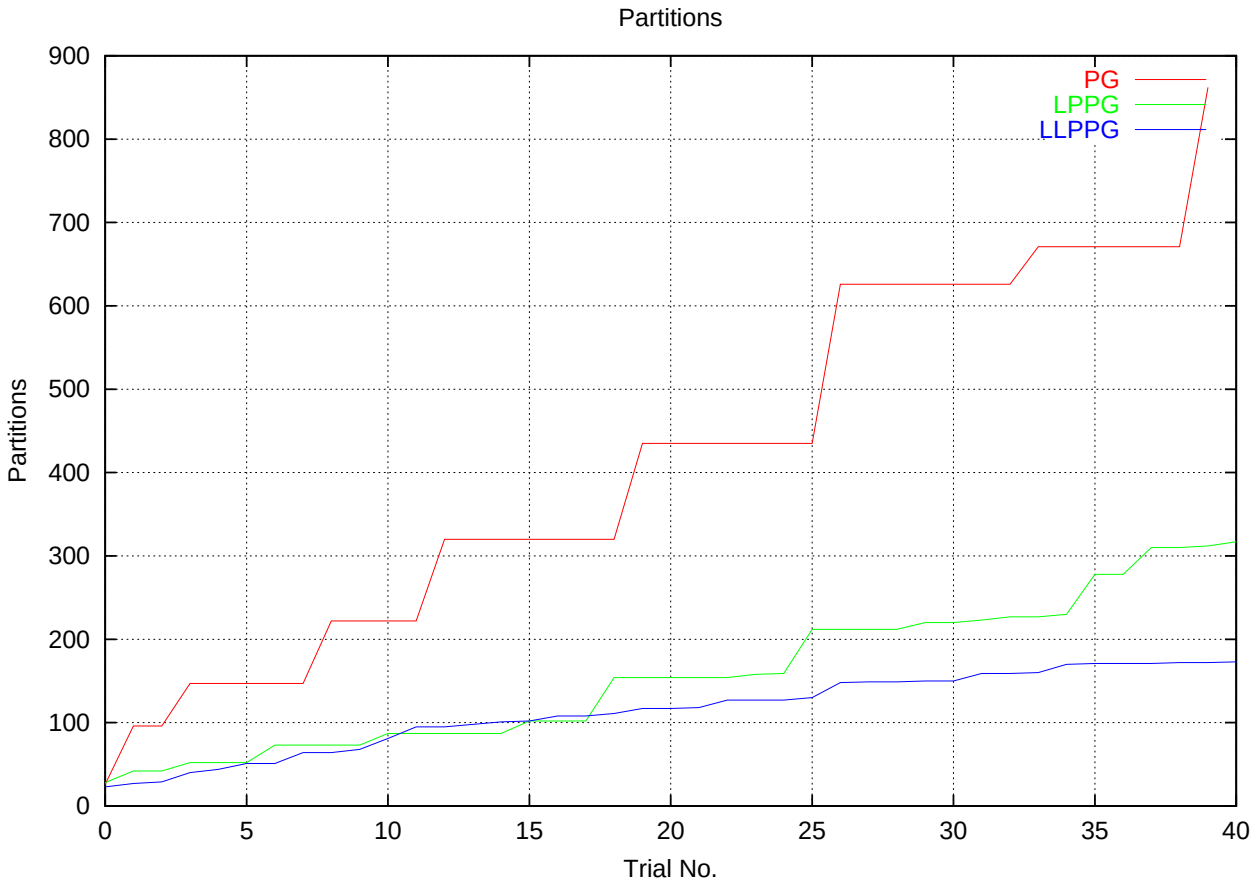


Figure 12: Graph showing the number of partitions needed by each of the algorithms as a function of trial number when applied to the Acrobot problem.

Reason for the Acrobot Instability

This instability is arguably caused by the chaotic nature of the Acrobot problem, which is an inverted pendulum-type-problem. The algorithms introduced in this dissertation partition the state space into highly regular, hyper-rectangular-shaped cells. Such partitioning of a state space (together with the algorithm for assigning cell costs, choosing neighbors to aim for and the local controller used) represents the algorithms' approximation of regions in which the system being controlled behaves more or less uniformly with respect to the policy being followed.

While the shapes of these regions will almost always be only a rough approximation of the proper control regions that could perfectly describe the problem at hand, the consequences of the divergence from being optimal regions has the potential of being most noticeable when the system being controlled is chaotic. A chaotic system is one in which it is likely that two states that are extremely close in Euclidean distance can yield wildly different trajectories through the space under the same policy. To see why the chaotic nature of the Acrobot problem is a plausible explanation for the continual partitioning that we experience, we present the following argument.

The algorithms introduced in this dissertation re-compute cell costs every time a previously unknown cell-level experience is observed. Every time cell costs are

re-computed, the cell-to-cell policy can change as a result. Thus, if the agent under the control of one of our algorithms produces any new experiences during its last trajectory to the goal, it is possible that the policy used in the next trial will be different from the one just used. A new policy will likely lead to the agent following a different trajectory. Since all successful trajectories have to ultimately converge to the goal, they will likely have to pass through some of the same cells through which other trajectories to the goal have passed. However, different trajectories will likely enter cells common between them at different entry states. Assuming that the same policy will be followed from such a common cell on, when the system is a chaotic one, it is more likely than usual that starting from two different states in this cell, and aiming towards the same state in the same neighbor would result in two different cell-level outcomes, i.e., two different cells would result from the two aiming attempts.[1] The cell-level experience resulting from the latter of the two entry points into the common cell will be a new one, thus triggering a re-computation of cell costs—possibly leading to a change of cell costs—and the cycle repeats.

[1]Another factor that adds to the likelihood of this happening is that the local controllers we use only have a few of the allowable range of actions available to them.

3.4 Contributions

This chapter presented two successive algorithms that are derived from parti-game with differing partitioning strategies. The first, LPPG, improves on PG's performance in general, resulting in fewer partitions generated and fewer iterations needed to reach stable solutions. It also often took fewer steps in the world before it settled on a solution and almost always found a solution that is shorter than those found by PG.

The second algorithm, LLPPG, specifically addressed consistency problems that PG has in which it performed a disproportionately high amount of exploration on a very simple problem. We provided a general characterization of the conditions under which this kind of behavior occurs and presented a solution that completely eliminated the problem. The resulting algorithm is one that searches the state space along domain barriers much more uniformly than PG (and even LPPG) does and is thus not susceptible to the same type of disproportionate exploration that PG exhibited. This results in dramatic improvement in performance in those types of cases. More generally, LLPPG also resulted in better overall performance than LPPG—and, thus, usually PG as well—outperforming it or at least tying its performance in most cases.

Although they are able to find very good solutions in a very short amount of

time, neither PG nor our two algorithms were able to converge to a good solution in the acrobot task. We believe that this is due to the chaotic nature of the problem and the fact that it may not be suited for the type of rigid partitioning of the state space these algorithms generate. However, LPPG and LLPPG exhibited much slower growth in resource requirements than parti-game on this problem, an attribute that is very desirable in the face of the curse of dimensionality.

Chapter 4

Low Computational Requirements

Since the algorithms introduced in this dissertation are intended for on-line goal finding, the computational requirements of the action selection components of these algorithms need to be low enough to facilitate timely decision making. However, as has been explained in previous chapters, these algorithms require that minimax costs of the complete cell-space be recomputed every time a new experience is encountered. This means that the success of these algorithms hinges on the use of algorithms that can compute cell costs very efficiently.

One of the contributions of this dissertation is introducing a highly efficient algorithm for computing minimax cell costs. Two important speedup features contribute to the good performance of this algorithm, namely, single-pass computation

of the cell costs and the incremental computation of these costs. The following two sections will explain these features in detail and present the algorithms that implement them.

This chapter will also compare the performance of parti-game as presented in (Moore and Atkeson 1995) to our implementation of PG that uses our version of the minimax evaluation algorithm. We will also compare our results to the Q-learning results reported by Moore and Atkeson (1995).

4.1 Single-Pass Dynamic Programming

Classical dynamic programming algorithms typically require multiple passes over the sets of states, repeatedly performing backup operations until convergence is attained. Low resolution partitioning of the state space can greatly reduce the amount of time needed to find solutions. However, since the algorithms introduced in this dissertation, as well as parti-game, need to perform dynamic programming whenever a new experience is encountered, the low resolution partitioning of the space does not go far enough in making the algorithms suitable for on-line control.

Fortunately, it turns out that, by exploiting the nature of the problems that the algorithms introduced in this dissertation apply to, these computational requirements can be dramatically reduced without sacrificing the quality or correctness of

the results. The first of two significant features that facilitate this efficient computation is performing the dynamic programming to compute cell costs in a single pass over the cell space. The following subsection introduces the algorithm and provides a proof that it computes the correct function.

4.1.1 The Algorithm

Algorithm 4 performs the minimax cell cost computation used by parti-game, loser-partitioning parti-game and largest loser-partitioning parti-game. That is, it computes J_{WC} of Equation 1 as required by step 8 of Algorithms 1, 2 and 3. The outline of the algorithm is as follows. All cells are initially assigned a cost of $+\infty$, except for the goal cell which is given a cost of zero. The goal cell is also made the only member of the *frontier set*. In each iteration, every cell that neighbors a frontier cell is tested for eligibility for cost assignment. A cell is considered eligible if there exists an action from that cell for which all outcome cells have already been assigned finite costs. All cells found to be eligible are then assigned a cost that is equal to the iteration number, with iteration numbers starting from one. The algorithm terminates when a given iteration is not able to find any eligible cells to which to assign costs. All remaining cells retain their cost of $+\infty$ and are thus considered losing cells.

Algorithm 4 Single-pass minimax cost computation. Computes minimax cell costs, J_{SP}, for all cells in the space.

```
1: Input: A kd-tree representing the current partitioning of the state space.
2: Output: The input kd-tree with costs assigned.
3: IterationCount ← 0
4: J_SP(GOAL) ← 0
5: J_SP(c) ← +∞, ∀c ≠ GOAL
6: FrontierSet ← {GOAL}
7: while FrontierSet ≠ ∅ do
8:    fs ← ∅
9:    IterationCount ← IterationCount + 1
10:   for each cell, c, neighboring a cell in FrontierSet do
11:      if c has an action, a_c, for which all outcome cells' costs are ≠ +∞ then
12:         J_SP(c) ← IterationCount
13:         fs ← fs ∪ {c}
14:      end if
15:   end for
16:   FrontierSet ← fs
17: end while
```

4.1.2 Proof of Correctness

The following is a proof that Algorithm 4 indeed computes J_{WC}.

Lemma 1 *Algorithm 4 computes J_{WC} of Equation 1.*

Proof. In the following discussion we will refer to the cost computed by Algorithm 4 as J_{SP}. We prove that $J_{SP} = J_{WC}$ by induction.

Initial step. For the goal cell, $J_{SP} = J_{WC} = 0$. Furthermore, $J_{SP} \neq 0$ for all other cells.

Assumption. Assume $J_{SP}(c) = J_{WC}(c)$ for all c for which $J_{WC}(c) \leq n$.

Induction Step. Here we need to prove that $J_{SP}(c) = J_{WC}(c)$ for all c for which $J_{WC}(c) = n+1$. If $J_{WC}(c) = n+1$ for some cell c, the best worst-case cost among available actions at c is $n+1$. Any action with a worst-case cost of $n+1$ necessarily has at least one outcome that is a cell with a cost of n and no outcomes with a cost higher than n. In iteration $n+1$, a cell is found eligible by Algorithm 4 if and only if there exists an action for which all outcomes have been assigned costs. Since, by the **Assumption** step, all cells with costs $\leq n$ are assigned costs before iteration $n+1$, cell c will be found eligible by Algorithm 4 and, therefore, $J_{SP}(c) = n+1$.

If $J_{WC}(c) > n+1$, the best worst case action from c leads to a cell with cost $> n$ and, thus, will not be found eligible by Algorithm 4. Therefore, c will not be assigned a cost in iteration $n+1$, which means that $J_{SP}(c) \neq n+1$. □

4.2 Incremental Computation

Although the improvement of the previous section goes a long way in reducing the computational requirements of the dynamic programming step of the algorithms dealt with in this dissertation, it turns out that we can do better. The reason DP is performed after encountering a new experience is to facilitate proper cell-cost-based action selection at the next time step. However, these minimax-based algorithms perform greedy action selection based only on the immediate neighbors of the cell

the agent occupies at the time an action needs to be selected. As is the typical relationship between value/cost functions and greedy policies based on them, many cell cost assignments can result in the same policy in which a cell cost assignment using J_{WC} (of Equation 1) would result.

This section introduces an enhancement to algorithm 4 that results in identical agent behavior to that achieved under full J_{WC} computation, while limiting cost updates to only a subset of the complete cell space, thus requiring far fewer backups. This algorithm guarantees that whenever the agent needs to make a decision, the cost of the neighbors of the cell it currently occupies are up to date and equal to J_{WC} of those cells. This, in turn, guarantees that an agent that uses this enhancement will behave identically to one that utilizes a full cell cost computation algorithm for making decisions.

4.2.1 The Algorithm

Algorithm 5 implements this modification by changing steps 3, 5, 6 and 7 of Algorithm 4. In step 3, instead of always setting IterationCount to zero, it is only set to zero when the agent has just been reset to the starting state. In all other cases, it is set to the minimum of one less than the cost of c_i, and the cost of c_e, where c_i is the cell the agents started its aiming attempt from, and c_e is the cell the agent

Algorithm 5 Incremental minimax cost computation. Computes minimax cell costs, J_{IC}, after an aiming attempt from cell c_i that resulted in the agent entering cell c_e. Only costs of cells for which cost computation is necessary for the next step to be taken by the agent are computed.

1: Input: A kd-tree representing the current partitioning of the state space.
2: Output: The input kd-tree with costs assigned.
3: **if** Agent was just reset to start state **then**
4: IterationCount $\leftarrow 0$
5: **else**
6: IterationCount $\leftarrow \min(J_{IC}(c_i) - 1, J_{IC}(c_e))$
7: **end if**
8: $J_{IC}(\text{GOAL}) \leftarrow 0$
9: $J_{IC}(c) \leftarrow +\infty, \forall c \mid J_{IC}(c) > \text{IterationCount}$
10: FrontierSet $\leftarrow \{c \mid J_{IC}(c) = \text{IterationCount}\}$
11: **while** FrontierSet $\neq \emptyset$ **and** $J_{WC}(c_e) = +\infty$ **do**
12: fs $\leftarrow \emptyset$
13: IterationCount $\leftarrow$ IterationCount $+ 1$
14: **for** each cell, c, neighboring a cell in FrontierSet **do**
15: **if** c has an action, a_c, for which all outcome cells' costs are $\neq +\infty$ **then**
16: $J_{IC}(c) \leftarrow$ IterationCount
17: fs $\leftarrow$ fs $\cup \{c\}$
18: **end if**
19: **end for**
20: FrontierSet $\leftarrow$ fs
21: **end while**

actually entered. This value of IterationCount is the maximum cost of a cell that could not possibly be affected by the experience the agent has just received. In other words, any cell with a current cost less than or equal to this value will retain that cost under this new experience. In step 5, instead of setting the costs of all non-goal cells to $+\infty$, only those cells with current costs greater than the initial IterationCount are set to $+\infty$. In step 6, the frontier set is initialized to contain all

cells that have a cost equal to the initial value of IterationCount, whereas it was set to contain only the goal cell in Algorithm 4. In step 7, the condition of the loop is changed such that it is terminated when the cell the agent currently occupies is assigned a cost, as opposed to quitting only when no more cells could be assigned costs as in Algorithm 4.

4.2.2 Proof of Correctness

Here we prove that propagating costs only from the cell from which the agent started an experience up to the cell in which the experience ends results in identical behavior to that achieved when complete backup sweeps are performed.

Lemma 2 *Assume that at time t the agent was in cell c_i, aimed towards cell c_a and entered cell c_e, forming the experience triple (c_i, c_a, c_e). Assume further that the triple (c_i, c_a, c_e) is a new experience, which measns that cell costs need to be recomputed to take it into account. Let $M = \min\left(J^t_{WC}(c_i) - 1, J^t_{WC}(c_e)\right)$. If all cells, i, with $J^t_{WC}(i) > M$ are set to $+\infty$, then backups are started from all cells j with $J^t_{WC}(j) = M$ and stopped when c_e is assigned a non-infinity cost, $MinMaxNeighbor(c_e)$ at time $t + 1$ would be the same as if a full backup sweep was performed.*

Proof. Equation 1 for J_{WC} is repeated below.

$$J_{WC}(i) = \begin{cases} 0 & \text{if } i = \text{GOAL} \\ 1 + \min\limits_{j \in \text{NEIGHS}(i)} \max\limits_{k \in \text{OUTCOMES}(i,j)} J_{WC}(k) & \text{otherwise} \end{cases}$$

When the new experience triple (c_i, c_a, c_e) is received, the only value that changes in the OUTCOMES function is OUTCOMES (c_i, c_a). The change of OUTCOMES (c_i, c_a) means that $J^{t+1}_{WC}(c_i)$ might change. Since J_{WC} starts out with complete optimism (i.e., assuming every neighbor is reachable from any given cell) the fact that an experience is new means that it is a negative experience, which can only serve to move the costs upward. Thus, the change in OUTCOMES (c_i, c_a) means that $J^{t+1}_{WC}(c_i) \geq J^{t}_{WC}(c_i)$.

If $J^{t+1}_{WC}(c_i) = J^{t}_{WC}(c_i)$, then no backups, full or otherwise, will change any costs and, thus, performing partial backups would result in identical $MinMaxNeighbor(i)$ for all cells i.

If $J^{t+1}_{WC}(c_i) > J^{t}_{WC}(c_i)$, any cell i whose cost, $J^{t}_{WC}(i)$, was dependant on $J^{t}_{WC}(c_i)$ may potintially go up in cost. However, any cell i with $J^{t}_{WC}(i) \leq J^{t}_{WC}(c_i)$, by definition, can not be dependent on c_i's cost and, thus, its cost would not go up as a result of c_i's cost going up. Furthermore, since $J^{t}_{WC}(c_i) \leq M$, we can also say that any cell i with $J^{t}_{WC}(i) \leq M$ will not go up in cost as a result of the new experience, (c_i, c_a, c_e). Therefore, starting backups from cells whose cost is M would not leave

behind any cells whose cost might change as a result of the new experience.

If backups are halted when the current cell, c_e, recieves a cost, the agent is able to take its next step from c_e with information equivalent to when full backups have been performed. Since all the uncomputed cells whose $J^t_{WC}(i) > M$ will be left with a cost of $+\infty$, i.e., are losing cells, if the agent enters any of them as a result of a subsequent step, recomputation would immediately be forced, as parti-game and the algorithms introduced in this dissertation dictate. □

4.3 Effect on Performance

To assess the effect of using Algorithm 5 as the minimax cost computation algorithm compared to performing full backup sweeps, we compared two versions of parti-game, one using each of Algorithms 4 and 5. To provide a good reference point for the comparison, we reconstructed the continuous maze that (Moore and Atkeson 1995) used to compare PG with Q-learning and prioritized-sweeping.

Figure 13(a) shows the empty maze and 13(b) shows the maze with the partitioning and final trajectory after our implementation of parti-game converged to a solution from the start state shown. As we proved in the previous sections, our two versions of parti-game produce exactly the same solution, after exploring the state

space in exactly the same way and producing identical partitionings. The difference between the algorithms, however, can be seen in the number of backups they perform. The bottom half of Table 3 lists the number of steps taken in the world and the number of backups that were performed before convergence was attained. We can clearly see that Algorithm 5 vastly outperforms Algorithm 4, performing only about 16% of the number of backups that the latter performed!

The top half of Table 3 shows results reported by Moore and Atkeson (1995). We chose to include the results of parti-game and the version of Q-learning that was informed of the location of the goal, since this is the only algorithm that performed better than parti-game and only in the number of backups needed to converge.

Although differences in the two implementations probably exist, it is reassuring to see that Figure 13(b) is almost identical to the corresponding figure in (Moore and Atkeson 1995). Despite that, we can see a stark difference in that the number of steps taken by our implementation is almost exactly twice the number reported by Moore and Atkeson (1995). This does not appear to be due to a difference in step size. The step size we use is one hundredth of the width of the square state space, while (Moore and Atkeson 1995) report that they converted the number of steps taken by PG such that all results are expressed in steps with length that is one fiftieth of the with of the space. Therefore, the numbers we show here are

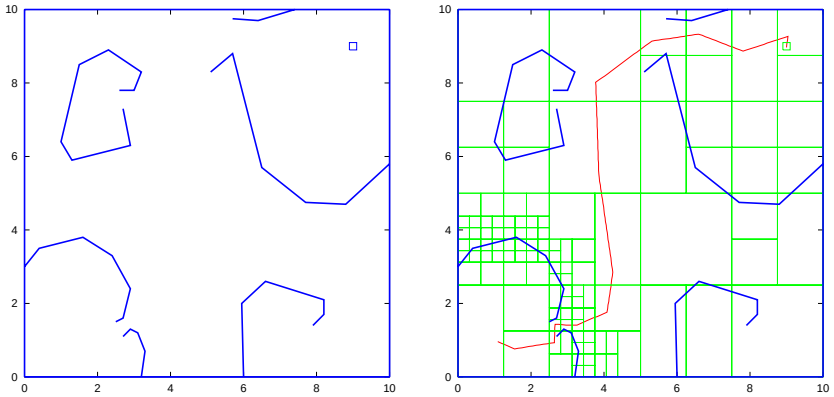


Figure 13: A re-creation of the maze used in (Moore and Atkeson 1995). (a) shows the empty maze and (b) shows the solution to which our implementation of parti-game converged.

Algorithm	Steps up to Convergence	Backups up to Convergence
PG (Moore & Atkeson '95)	954	20,682
Q-learning, goal known (Moore & Atkeson '95)	13,526	13,526
PG w/ Alg. 4	1,919	24,937
PG w/ Alg. 5	1,919	3,900

Table 3: The top half of this table shows the results reported by (Moore and Atkeson 1995), showing their comparison of parti-game against a version of Q-learning that was given information about where the goal is. The bottom half shows the performance of our implementation of parti-game using Algorithms 4 and 5.

already divided by two to compensate for that.

However, even though our implementation of PG takes almost twice as many steps as Moore and Atkeson (1995)'s, the number of backups they report is only 17% lower than what we achieved from our implementation of PG with Algorithm 4. Furthermore, the more than six-fold improvement in the number of backups that we achieved when we switched to Algorithm 5 for computing minimax costs is so significant that it overshadows the small differences we see here, indicating that using Algorithm 5 for cell cost computation results in PG outperforming Q-learning even in the case where Q-learning is informed of the location of the goal.

4.4 Contributions

This chapter presented two very efficient algorithms that compute minimax cell costs for any of the algorithms dealt with in this dissertation. The first allows computation of cell costs in only one pass, making it a very efficient dynamic programming algorithm for use with this class of algorithms. The second leads to even further improvement since only a subset of the number of backups that need to take place to compute all cell costs are performed when the agent encounters a new experience. We also proved that the two algorithms presented here do not affect the external behavior of the algorithm that is utilizing them to compute minimax

cell costs. Finally, we showed that the second algorithm introduced in this chapter requires far less backups than PG required for a particular maze introduced in Moore and Atkeson (1995).

The low computational requirements of this algorithm and the reduction in the number of partitions and steps required to reach stable solutions achieved by LLPPG (and, to a lesser degree, LPPG) of Chapter 3 combine to provide a fast, efficient algorithm for solving continuous, deterministic, goal-type problems. In light with the evidence of good scalability of the original parti-game algorithm shown by Moore and Atkeson (1995), the superior performance of the algorithms we present here holds great promise for even better scalability.

Chapter 5

Optimizing Solutions

As was stated in Section 2.3, parti-game does not claim to be able to find optimal solutions to the reinforcement learning problems it is designed to solve. The algorithms introduced in this dissertation thus far have not addressed this problem directly and, therefore, inherit this limitation from PG. A clear example of this can be seen in the results of maze a in Figure 9, which are repeated here in Figure 14. We saw there how all these algorithms settle on the longer of the two possible routes to the goal in this maze. In this chapter, we investigate ways we could improve upon the performance of these algorithms so that they could find paths of *optimal form*.

It is important to note that we are not seeking paths that are optimal, since

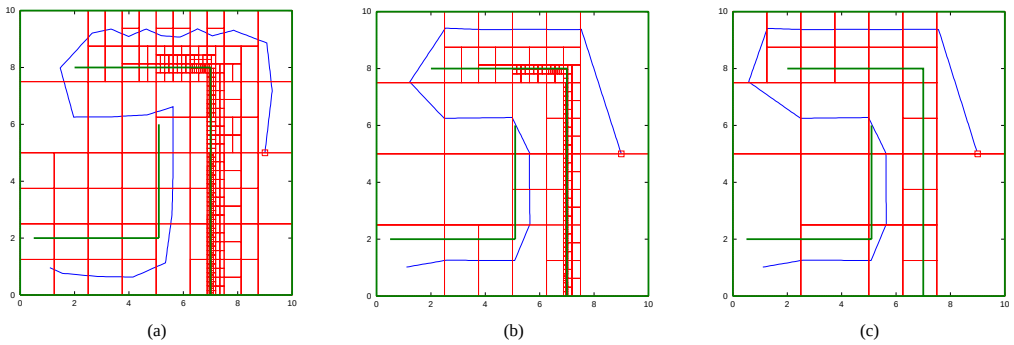


Figure 14: The results of the final trial of applying (a) PG, (b) LPPG and (c) LLPPG to the simple maze of Figure 1(a).

that is not generally possible to achieve using the cell shapes and aiming strategies we are using here. By a path of optimal form we mean a path from which a true optimal path can be obtained by performing continuous deformations on such path. Intuitively, a path of optimal form is one that makes use of all the shortcuts in the state space. For example, in the puck-on-a-hill problem, an optimal form path is one that starts out going to the left far enough, then thrusts to the right and is able to climb up the hill and reach the goal. Such a path may not be truly optimal because it does not switch actions at the right moment in time. However, we say that it is of optimal form because it does not, for example, cycle a few times between going left and right before it reaches the goal.

5.1 Other Gradients

As explained in previous chapters, parti-game partitions only when the agent has no winning cells to aim for and the only cells partitioned are those that lie on the win-lose boundary. The win-lose boundary falls on the gradient between finite- and infinite-cost cells and it comes to exist when the algorithm knows of no reliable way to get to the goal from some part of the state space based on the current partitioning and the experiences gathered so far. Consistently partitioning along this gradient guarantees that the algorithm will eventually find a path to the goal, if one exists.

However, gradients across which the difference in cost is finite also exist in a state space partitioned by parti-game (or any of the algorithms introduced in this dissertation). Like the win-lose boundary, these gradients are boundaries which the agent does not believe it has a reliable way of penetrating. Although finding an opening in such a boundary is not essential to reaching the goal, these boundaries potentially have shortcuts that might improve the agent's policy. Any gradient with a difference in cost of two or more is a potential location of such a useful shortcut and parti-game can be made to explore those areas if we partition along those boundaries.

Because such gradients exist throughout the space, we need to be selective

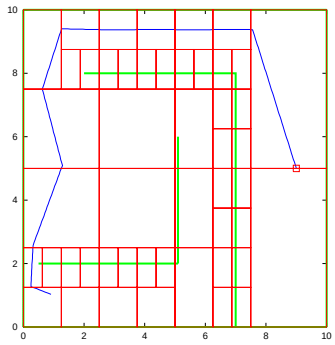


Figure 15: The solution found by applying the global improvement algorithm on the maze of Figure 1(a). The solution proceeded exactly like that of the algorithm of section 3.3 until the solution in the rightmost column of Figure 9(a) was reached. After that, eight additional iterations were needed to find the better trajectory, resulting in 22 additional partitions, for a total of 49.

about which ones to partition along. There are many possible strategies one might consider using to incorporate this additional partitioning. For example, since parti-game focuses on the highest cost difference gradients only, the first thing that comes to mind is to follow in parti-game's footsteps and assign partitioning priorities to cells along gradients based on the differences in costs across those gradients. However, since the true cost function typically has discontinuities, it is clear that the effect of such a strategy would be to continue refining the partitioning indefinitely along such a discontinuity in a vain search for a nonexistent shortcut.

5.2 Algorithm for Optimizing Solutions

A much better idea is to try to pick cells to partition in a way that would achieve balanced partitioning, following the rationale introduced in section 3.3. Again, such a strategy would result in a uniform coarse-to-fine search for better paths along those other gradients.

The following discussion could in principle apply to any of the algorithms we studied up to this point. However, because of the superior behavior of LLPPG, we will use it as the basis for the improvements studied in this chapter.

The way we incorporated partitioning along other gradients is as follows. At the end of any trial in which the agent is able to go from the start state to the goal without any unexpected results of any of its aiming attempts, we partition the largest "losing cells" (i.e., higher-cost cells) that fall on any gradient across which costs differ by more than one. Because data about experiences involving cells that are partitioned is discarded, the next time LLPPG is run, the agent will try to go through the newly formed cells in search of a shortcut.

This algorithm amounts to simply running LLPPG until a stable solution is reached. At that point, new cells along some of the other gradients are introduced and, when it is subsequently run, LLPPG is applied again until stabilization is achieved, and so on. The results of applying this algorithm to the maze of Figure 14

is shown in Figure 15. As we can see, the algorithm finds the better solution by increasing the resolution around the relevant part of the barrier above the start state.

In the absence of information about the form of the optimal trajectory, there is no natural termination criterion for this algorithm. It is designed to be run continually in search of better solutions. If, however, the form of the optimal solution is known in advance, the extra partitioning could be turned off after such a solution is found.

5.3 Applicability

When is this algorithm applicable, and what type of solutions can it find? The minimax, partitioning-based algorithms we have studied in this dissertation rate solutions based on J_{WC}, which is the number of cells through which the agent believes it needs to travel to get to the goal. The algorithm we presented in this chapter is capable of finding a shortcut in the state space if the partitioning that is required to expose such shortcut results in a path that is shorter, in terms of cell count, than any other solution. Therefore, to the extent that a shorter path in the Euclidean sense (or any other measure of interest) is also representable as a shorter path in the cell count sense, this algorithm will be able to find it.

Naturally, an ordering of solutions based on Euclidean distance can be different from one based on cell count. Nevertheless, a solution that is far superior to another in Euclidean length is likely to also be superior when measured using cell counts. It is safe to say that the likelihood that this algorithm finds a shorter path is directly related to the superiority of this shorter path. If a solution is far superior to the current one, this algorithm is much more likely to find it than one that is only marginally better. Thus this algorithm does not guarantee finding every shorter path (in the Euclidean sense), but it has a higher likelihood of finding those that are *significantly* better than the solutions found by the minimax, partitioning-based algorithm being used.

5.4 Other Examples

Another application of this algorithm will be seen in Section 6.4.5, when the solution found by a version of the LLPPG algorithm that learns its own local controller from experience, which will be introduced in Chapter 6, arrives at sub-optimal form solutions to the puck problem presented in Section 3.2.1. As a preview, Figures 16(a) and 16(b) respectively show the sub-optimal-form solution originally found and optimal-form solution found by the global optimization algorithm presented here.

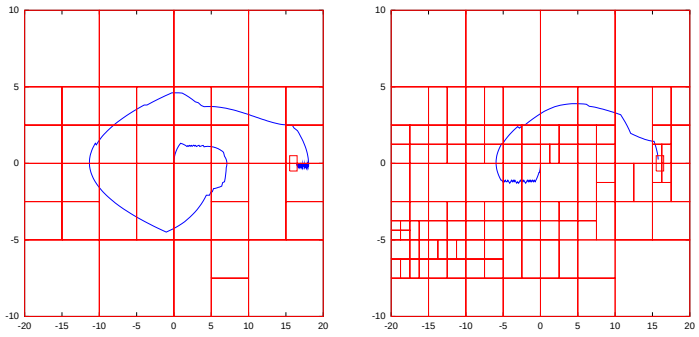


Figure 16: (a) A sub-optimal-form solution to the puck-on-a-hill problem. (b) The optimal-form solution reached after applying the global optimization algorithm to the solution in (a).

These figures will be shown again in Section 6.4.5 along with a complete description of the algorithm used and the circumstances under which the sub-optimal solution was found and how the global optimization algorithm was used to find the optimal-form solution.

5.5 Contributions

The algorithm presented in this chapter is useful when better solutions than those found by the minimax, partitioning-based algorithm being used are sought. This algorithm is capable of finding solutions that are qualitatively (and quantitatively) better than the ones arrived at by the main algorithms. What is meant by a

qualitatively better solution is one that can not be reached by deforming the current path by making local, low-level perturbations in the stationary policy arrived at by the main algorithm.

Furthermore, this algorithm is inherently more efficient than any algorithm that tries to find better policies by searching directly in the space of low-level policies because it uses the low-resolution partitioning structures the companion minimax, partitioning-based algorithm produces to perform a more informed search based on the experience gained while exploring the space.

Coupled with any of the efficient algorithms introduced in this dissertation, this algorithm adds to the attractive features of this family by facilitating finding solutions that are closer to the optimal ones.

Chapter 6

A priori Model-Independence

6.1 Introduction

Since the goal of a learning agent is to improve its behavior over time by utilizing the experience it gains from the environment, *a priori* domain knowledge, if available, can be very helpful in providing a head start on the agent's learning. However, learning algorithms that can operate without *a priori* domain knowledge are very desirable for a number of reasons.

First, many problems provide little or no *a priori* domain knowledge and we would like a learning algorithm to be able to function under such initial conditions and be able to learn and improve its performance without requiring initial domain

knowledge to be incorporated in its design. Second, existing domain knowledge might be available in forms that are not suitable for the learning algorithm at hand. For example, we might have a model of the dynamics of the problem the agent is trying to solve that maps state/action pairs to states while the learning algorithm requires a very different type of model (good examples of this can be found in (Moore 1991) or (Moore and Atkeson 1992). Third, the class of learning algorithms being used might not be able to make use of *a priori* domain knowledge (Maclin and Shavlik 1994). While finding ways of facilitating the incorporation of existing domain knowledge into a learning paradigm is a very important area of research, trying to make such a learning paradigm able to operate under little or no domain knowledge is an orthogonal direction of pursuit that has intrinsic value. In general, *a priori* model independence is a very desirable feature of any RL algorithm because it brings us closer to the ultimate goal of being able to place a learning agent in a completely unknown environment and expect it to exhibit intelligent behavior over time.

6.2 The Role of Local Controllers

One important feature of the algorithms discussed and introduced in this dissertation thus far is that they do not require a formal *a priori* model of the dynamics

of the environments in which they are to operate. However, these algorithms are not completely model-free, since they rely on a pre-specified local controller which, by definition, encompasses some form of domain knowledge that allows it to move the agent locally.

This chapter attempts to eliminate these algorithms' requirement of *a priori* local controllers while preserving their attractive features that enable them to find good solutions very efficiently. This is done by replacing the *a priori* controller with one that is learned from the experience the agent gains as it explores the environment. That is, these algorithms will now utilize experiences gained from the environment at two distinct levels. The first is the cell-to-cell experience level, which they already use, and the second is at the lowest level available, which will be used to build and improve the local controller being learned.

This dual use of experiences and basing partitioning on evolving local controllers, as opposed to ones that are fixed *a priori*, might have a significant effect on the behavior of the agent and the partitionings generated by these algorithms. For these reasons, before we delve into the details of how we will incorporate learning local controllers into these algorithms, we will first study the requirement for local controllers further. We will examine the need for the local controllers, the extent to which these algorithms rely on them, and how their quality affects the solutions

found by these algorithms. This is done to understand and anticipate the possible effects that replacing the *a priori*, hand-crafted controllers with learned ones might have on the performance of these algorithms.

6.2.1 The Need and Dependence on Local Controllers

The local controllers are needed to perform the low-level actions of moving the agent in the state space. They are commanded by the agent to move from its current state to a neighboring state, based on a decision made by the associated high-level, partitioning-based algorithm. Partitioning ensues when the local controller fails to take the agent to the target and the agent thinks there is no way it can reliably reach the goal.

The partitioning found by these algorithms can basically be seen as an attempt to get around the inability of these local controllers to get the agent to an area of the state space it views as crucial to reaching its goal. A partitioning also represents a hypothesis about the dynamics of the environment based on the given local controller and the experiences gained thus far during exploration. When more experiences are received, the hypothesis might need to be changed to be consistent with the new, larger set of experiences. Because of the agent's optimistic view of its cell space with which it considers every neighbor to be accessible from a given

cell unless proven otherwise, the more the agent explores, the more it is likely to encounter experiences that do not fit with its current hypothesis (partitioning). These experiences are considered "negative" experiences with respect to the current hypothesis (partitioning). When enough negative experiences have been encountered, a new hypothesis is formed by performing additional partitioning, resulting in higher resolution representations of the state space.

6.2.2 How Local Controller Quality Affects Solution Quality

So, how does having one local controller as opposed to another manifest itself in the behavior of these algorithms? That is, how much does it affect their exploration, partitioning and, ultimately, the final solutions they find? In the most simplistic case, one for which only the simplest hypothesis is needed, the *a priori* local controller would be able to take the agent from the start state to the goal region, as it is commanded to do in the initial step the algorithm takes, and no partitioning would be required. However, if the local controller fails to do that, because it runs into domain barriers that conflict with its simplistic notions of how to move in the space, partitioning takes place, i.e., the hypothesis gets more complex. The final partitioning (hypothesis) arrived at is one with which the local controller can

reliably move, cell to cell, from the start state to the goal.

This clearly means that the better the local controller, the more useful exploration and the less partitioning the given algorithm will need. At one extreme, under a perfect controller, no exploration, and, thus, no partitioning will take place. At the other extreme, continual, hopeless exploration, and, consequently, infinite partitioning will take place. Of course, if such a perfect controller existed, there would be no need for any of these algorithms, because that controller would be one that is single-handedly able to solve the goal-finding problem we are trying to tackle. In reality, a controller devised by the designer will be one that falls somewhere in between: certainly not perfect, but, on the other hand, knowledgeable enough to facilitate finding the goal without requiring too much exploration and overly fine partitioning.

6.2.3 Learning the Local Controller

As mentioned above, the goal of this chapter is to get away from the requirement of specifying a local controller *a priori* by the designer. However, a learned controller would most likely be inferior to a hand-crafted one, at least initially, when starting with absolutely no knowledge of the environment. In such cases, the agent will likely do more exploration, and, as explained above, this is likely to lead to more

partitioning by the algorithm using this controller. Although such learned local controller will improve over time as it incorporates more data into its model, the effects of its earlier inaccuracy will remain in the kd-tree model representing the partitioning of the algorithm using it. This is because its inaccuracy would have likely contributed to high exploration and the consequent fine partitioning.

Thus, a price that we are likely to pay for complete independence of *a priori* domain knowledge is that the agent might have to perform more intensive exploration and the final partitionings arrived at might be of higher resolutions. Of course, this does not come as a surprise and the important question that remains to be answered is exactly how much worse solutions found under learned local controllers would be compared to ones based on pre-designed, hand-crafted ones, and whether these algorithms will continue to have superior performance compared to other counterparts under similar initial conditions.

The rest of this chapter will be dedicated to answering this question. We will start by describing a specific algorithm for online learning of local controllers that integrates well with the algorithms introduced in this dissertation. We will then apply these algorithms to our set of bench mark problems and compare their performance to when hand-crafted controllers are used.

6.3 Learning Local Controllers Using Tile Coding

We now shift our attention to the implementation of a local controller learner that integrates with the algorithms introduced in this dissertation. For the sake of brevity, we will refer to the learned local controller as the LLC for the rest of this discussion. We will use this term to refer to the module that performs the two distinct tasks of learning an approximate model of the dynamics of the system and using this learned model to perform the local control tasks the agent requests.

As mentioned above, the most important goal in designing a local controller learner that integrates with these algorithms is for it not to cause an excessive reduction in the original superior performance of these algorithms. Furthermore, to preserve these algorithms' on-line nature, the learning method chosen should be both fast and suitable for use with the incremental flow of training data that occurs with these algorithms.

6.3.1 Tile Coding Function Approximators

Tile coding methods (Sutton and Barto 1998), also known as CMAC's (for Cerebellar Model Articulator Controller) (Miller, Glanz, and Kraft 1990), are a good

candidate to use for this task because of their fast learning using the LMS rule and their allowing incremental learning. In what follows we give a brief introduction to tile coding approximators. Miller, Glanz, and Kraft (1990) give a detailed description of CMAC's and Sutton and Barto (1998) provide a good introduction to tile coding techniques and how they are used in reinforcement learning.

A tile coding function approximator is typically used to learn a function that maps states from a continuous state space to values.[1] Thus, a typical function to be learned by a tile coding approximator might have the form $\mathrm{F} : \mathcal{R}^n \rightarrow \mathcal{R}$. The approximator represents the state space using one or more *tilings*, each of which overlaying the entire state space, where a tiling is a hyper-rectangular-shaped structure that has the same dimensionality as the state space. Tilings are usually slightly larger than the state space and multiple tilings used by an approximator are laid over the state space in a staggered manner using displacements that are either uniform or chosen randomly. Each tiling is divided into one or more *tiles*, where a tile is the basic unit that is assigned a value in the tile coding setup. The approximated function value at a state is equal to the sum of the values of all the tiles (one from each tiling) in which the state falls. Figure 17 is a depiction of how a tile coding approximator can be set up for 2D state space.

[1] Although this introduction is specific to state value functions, it straightforwardly extends to state/action value functions, as will be shown in the following section.

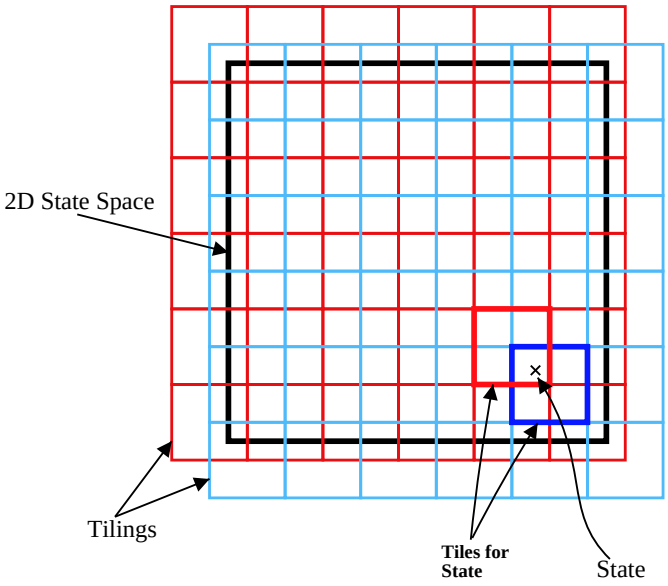


Figure 17: A 2-tiling tile coding approximator for a 2D state space.

A tile coding approximator is trained using the LMS update rule:

$$\mathrm{F}^{t+1}(s) \leftarrow \mathrm{F}^{t}(s) + \eta\, \mathrm{E}^{t}(s),$$

where η is the learning rate and $\mathrm{E}^{t}(s)$ is the observed error at time t (i.e., the difference between the actual, observed value of the function at time t and the its approximation, $\mathrm{F}^{t}(s)$). The update of the function value at a state is distributed equally among all tiles in which the state falls. Thus, if there are n tilings, one tile in each of the tilings will receive an additive update of $\eta/n\, \mathrm{E}^{t}(s)$.

6.3.2 Configuring the Tile Coding Approximator

The function of any local controller is to try to move the agent to the target state to which the agent commands it to go. The central component of the LLC we will use is a tile coding function approximator that approximates a function of the form $\mathrm{F} : S \times A \rightarrow \mathcal{R}^n$, where S is the set of states, A is the set of actions and n is the dimensionality of the underlying state space. F maps state/action pairs to a displacement vector representing an approximation of the state change that would occur if that action is taken at that state. Since the state spaces of the problems our algorithms handle are continuous, and assuming that our actions always come from a subset of $\mathcal{R}^m$, where m is the dimensionality of the action space, we will write F as $\mathrm{F} : \mathcal{R}^{(n+m)} \rightarrow \mathcal{R}^n$. When the LLC is commanded to head toward a target state, s, from current state, r, it executes action a determined by

$$a = \arg\min_{a \in A} \|(\vec{s} - (\vec{r} + \mathrm{F}(r, a))\| \tag{3}$$

where A is the set of allowable actions. That is, the action taken by the LLC at each time step is the one which the learned function, F, predicts would take the agent closest to the target state, in Euclidean distance, based on a one-step look-ahead. In short, the LLC is a greedy controller based on F.

The mapping above will be learned using n tile coding function approximators,

one for each of the components of the $\mathcal{R}^n$ vector to be learned. We will call the functions being learned by the tile coding approximators $\mathrm{F}_1, \mathrm{F}_2, \ldots, \mathrm{F}_n$, each of the form $\mathrm{F}_i : \mathcal{R}^{(n+m)} \rightarrow \mathcal{R}$. In the remainder of this discussion, F will be used to denote the group of functions F_i, where $i = 1 \ldots n$, and we will call n the *cardinality* of F. We will use $\mathrm{F}(s, a)$ to denote the n-dimensional vector $(\mathrm{F}_1(s, a), \mathrm{F}_2(s, a), \ldots, \mathrm{F}_n(s, a))$. Thus, each of the tile coding function approximators will use tilings that have a dimensionality of $n + m$. To guarantee proper representation of actions, each of the action dimensions will have an adequate number of splits such that each allowable value in each dimension is represented by a sub-interval into which no other allowable values fall.

F is updated after every step is taken using the LMS rule. If applying action a while in state s at time t resulted in the agent entering state r, then $\vec{\theta}^t = \vec{r} - \vec{s}$ is the vector representing the actual state change that results from applying action a in state s. F will be updated with the update rule

$$\mathrm{F}^{t+1}(s, a) \leftarrow \mathrm{F}^t(s, a) + \eta \, \mathrm{E}^t(s, a) \tag{4}$$

where $\mathrm{F}^t(s, a)$ is the value of $\mathrm{F}(s, a)$ at time t, $\mathrm{E}^t(s, a) = \vec{\theta}^t - \mathrm{F}^t(s, a)$, which is the error in $\mathrm{F}(s, a)$ at time t, and $\eta < 1$ is the *learning rate*. Thus, the update for each $\mathrm{F}_i(s, a)$ is $\eta \, \mathrm{E}_i^t(s, a) = \eta \left(\vec{\theta}_i^t - \mathrm{F}_i^t(s, a) \right)$.

6.3.3 Choosing the Parameters

The most important parameters that affect a tile coding function approximator are the number of tilings and the layout and sizes of tiles within each tiling. These parameters are usually set and changed in a variety of ways in order to arrive at settings that are suitable for the particular problem at hand. In keeping with our goal of complete independence from *a priori* knowledge about the system, our choices of tilings will intentionally not take into consideration any special characteristics of the problem being solved, except for its general, relative complexity. To that end, we will always use the same, *uniform* tile distribution in *all* tilings of any given dimension. We will only vary the total number of tilings and the density of the *uniform* tiling distribution of all the tilings in a given dimension.

6.3.4 Exploration Strategy

An important decision to make about the LLC is how to address the classical exploration vs. exploitation trade off. While the way in which to balance these usually competing objectives is already established for the algorithms introduced in this dissertation, the LLC needs to address this trade-off at the much lower level of choosing actions at each time step while it tries to learn the dynamics of the environment and evolve into a suitable local controller.

Our algorithm uses a principled exploration strategy in which the probability of exploration is based on an approximation of the normalized exponential average squared error observed for each state/action pair. To achieve this, we use the same tile coding structure used to approximate F to also approximate three additional quantities. Specifically, we will store approximations of the exponential average squared error at a given state, action and dimension, $\mathrm{MSE}_i(s,a)$, and the maximum squared error ever experienced at a state, action and dimension, $\mathrm{MaxSE}_i(s,a)$. We will also store the approximate number of times an action is taken at a state, $\mathrm{n}(s,a)$.[2]

By using the same tile coding structure of F, we mean that we will approximate MSE and MaxSE and store $\mathrm{n}(s,a)$ at the same resolution at which F is being approximated and we will use the same tilings to store values for all four functions. Thus, for a pair of triples (s_1,a_1,i_1) and (s_2,a_2,i_2), where $s_1 \neq s_2 \vee a_1 \neq a_2 \vee i_1 \neq i_2$, if the same combination of tiles is used to represent $\mathrm{F}_{i_1}(s_1,a_1)$ and $\mathrm{F}_{i_2}(s_2,a_2)$ (and therefore both have the same value) then all the members of the pairs $(\mathrm{MSE}_i(s_1,a_1),\mathrm{MSE}_i(s_2,a_2))$, $(\mathrm{MaxSE}_i(s_1,a_1),\mathrm{MaxSE}_i(s_2,a_2))$ and $(\mathrm{n}(s_1,a_1),\mathrm{n}(s_2,a_2))$ will also be represented by the same combination of tiles and thus be guaranteed to

[2]We define n as a function of the state and action only, independent of state space dimensions. However, the underlying representation equally divides the value of $\mathrm{n}(s,a)$ among all tile coding approximators designated for the different dimensions. This means that we represent $\mathrm{n}(s,a)$ as $\sum_{i=1}^{n} \mathrm{n}_i(s,a)$. However, this representational detail has no bearing on the material presented here.

have the same value. Of course, the same results could be achieved by using four independent tile coding structures. The choice to use one structure to represent all four functions is just an implementation decision that was made because it makes more efficient use of memory due its lower total overhead.

Each of the three other quantities has a different update rule than that of F. $\mathrm{MSE}_i(s,a)$ is updated as follows:

$$\mathrm{MSE_i^{t+1}}(s,a) \longleftarrow \begin{cases} \gamma\,\mathrm{MSE_i^t}(s,a) + (1-\gamma)\,\mathrm{E}_i^t(s,a)^2 & \text{if } \mathrm{E}_i^t(s,a)^2 \leq \mathrm{MaxSE}_i^t(s,a) \\ \mathrm{E}_i^t(s,a)^2 & \text{if } \mathrm{E}_i^t(s,a)^2 > \mathrm{MaxSE}_i^t(s,a) \end{cases} \tag{5}$$

where γ is the exponential decay rate and $\mathrm{E}_i^t(s,a)$ is the error at time t in dimension i, that is, $\mathrm{E}_i^t(s,a) = \vec{\theta}_i^t - \mathrm{F}_i^t(s,a)$. $\mathrm{MaxSE}_i(s,a)$ is the maximum squared error ever experienced at state s, action a and dimension i. Thus it is updated as follows:

$$\mathrm{MaxSE}_i^{t+1}(s,a) \leftarrow \max\left(\mathrm{MaxSE}_i^t(s,a), \mathrm{E}_i^t(s,a)^2\right) \tag{6}$$

where $\mathrm{E}_i^t(s,a)$ is as defined above. Finally, $\mathrm{n}(s,a)$ is updated as follows:

$$\mathrm{n}^{t+1}(s,a) \longleftarrow \begin{cases} \mathrm{n}^t(s,a) + 1 & \text{if } a \text{ is the action taken at time } t \\ \mathrm{n}^t(s,a) & \text{otherwise} \end{cases} \tag{7}$$

Based on these quantities we now successively define the *normalized* exponential average squared error for a state/action pair in a given dimension, $\overline{\mathrm{MSE}}_i(s,a)$, for a state/action pair over all dimensions, $\overline{\mathrm{MSE}}(s,a)$, and for a state over all actions

and dimensions, $\overline{\text{MSE}}(s)$. These are defined as follows:

$$\overline{\text{MSE}}_i(s,a) = \frac{\text{MSE}_i(s,a)}{\text{MaxSE}_i(s,a)} \tag{8}$$

$$\overline{\text{MSE}}(s,a) = \frac{1}{n}\sum_{i=1}^{n} \overline{\text{MSE}}_i(s,a) \tag{9}$$

$$\overline{\text{MSE}}(s) = \frac{1}{|A|}\sum_{a\in A} \overline{\text{MSE}}(s,a). \tag{10}$$

Note that we have omitted the superscript t from the definitions in the above three equations. From this point forward, when the the time superscript is omitted, it should be assumed to be t. In all other cases, the time superscript will be shown.

Obviously, each of the quantities in equations (8), (9) and (10) falls in the closed interval $[0,1]$. We would like to treat $\overline{\text{MSE}}(s)$ as the probability of taking an exploratory action in state s, and we want this probability to approach zero over time. However, $\overline{\text{MSE}}(s)$ as defined above might never reach zero in some situations. For example, if two or more states that the agent continues to visit share at least one tile and have target effects for some action that are different enough, the approximation, F, may continue to oscillate under some patterns in which the agent visits these states. To guarantee that exploration will ultimately stop, we define $\text{PE}^t(s)$, the probability of taking an exploratory action in state s at time t, as:

$$\text{PE}^t(s) = (1+\epsilon)^{-\,\text{n}(s)}\,\overline{\text{MSE}}^{\,t}(s) = \frac{(1+\epsilon)^{-\,\text{n}(s)}}{|A|}\sum_{a\in A}\overline{\text{MSE}}^{\,t}(s,a), \tag{11}$$

where $\mathrm{n}(s) = \sum_{a \in A} \mathrm{n}(s, a)$ and $\epsilon \ll 1$ is a constant that determines the decay rate of PE. Furthermore, we define $\mathrm{PE}(s, a')$ as the probability that action a' is chosen as the exploratory action in s, if it is determined that some exploratory action is to be taken in s, as:

$$\mathrm{PE}(s, a') = \frac{(1+\epsilon)^{-\mathrm{n}(s,a')} \, \overline{\mathrm{MSE}}(s, a')}{\sum_{a \in A} \left[(1+\epsilon)^{-\mathrm{n}(s,a)} \, \overline{\mathrm{MSE}}(s, a) \right]}. \tag{12}$$

Thus, our exploration strategy is $\mathrm{PE}(s)$-*greedy*, where s is the current state. That is, we take the greedy action determined by Equation (3) above with probability $1 - \mathrm{PE}(s)$, but take an exploratory action with probability $\mathrm{PE}(s)$. If an exploratory action is to be taken, we choose that action based on the probability distribution defined by Equation 12.

Under this exploration strategy, the likelihood that an exploratory action is taken at a state is directly related to the average MSE experienced at that state relative to the maximum squared error possible. At one extreme, when the exponentially decaying mean squared error is the highest it could be, every action taken will be an exploratory one. However, as more experiences are gained and the function F improves, the mean squared error will decrease and, thus, the probability of exploration will decrease, reaching zero at the extreme, in the ideal case of F becoming a perfect predictor of state change. Because of the coarse resolutions at which we represent F, although F is expected to come reasonably close to the target

function, reaching zero error is very unlikely. For this reason, the factor $(1+\epsilon)^{-\mathrm{n}(s)}$ adds a guarantee that exploration probabilities will converge to zero as time goes by.

Furthermore, the likelihood of choosing a particular exploratory action among allowable actions at a state is directly related to the exponentially decaying mean squared error encountered when that action has been taken at that state relative to the sum of the exponentially decaying average error encountered when taking each of the actions at that state. Thus, actions with the highest error will be taken with the highest probability compared to the probability with which other actions are taken. The probability of taking a particular action approaches zero as the average error encountered when taking that action approaches zero. Similarly to what we explained above, using the factor $(1+\epsilon)^{-\mathrm{n}(s,a')}$ in computing $\mathrm{PE}(s,a)$ guarantees it reaching zero, in the limit, in the face of representational inadequacies.

6.3.5 The Algorithm

Having defined all aspects of the LLC, we now show it in Algorithm 6 incorporated into our best-performing algorithm, largest loser-partitioning parti-game (LLPPG). Section 6.4 will show the results of applying this algorithm to our benchmark problems, compare the results to those obtained previously and assess the effects of

having to learn the local controller on the performance of these algorithms. Naturally, the LLC can be incorporated into any of the algorithms introduced in this dissertation in a similar manner.

6.3.6 The Curse of Dimensionality Revisited

Tile coding function approximators use look-up tables to represent tile values. Although tile coding techniques go a long way in addressing the curse of dimensionality by using very coarse representations of the functions they learn compared to direct look-up table techniques (for discrete state spaces or those quantized to a fine enough resolution for RL methods to be applicable), we might still run into the physical limits of the computer system if the dimensionality of the state and/or action spaces are high enough.

Let n and m be the cardinalities of the state and action spaces, respectively. Assuming, without loss of generality, that each of the state and action dimensions is a closed interval from $\mathcal{R}$, the function to be learned by the LLC is $\mathrm{F} : \mathcal{R}^{n+m} \rightarrow \mathcal{R}^{n}$. For each tiling used in approximating this function, we will require np^{n+m} table cells, where p is the number of sub-intervals in which we divide each dimension, assuming we divide each dimension into an equal number of sub-intervals. Furthermore, tile coding approximators usually utilize more than one tiling that span the state space

Algorithm 6 largest loser-partitioning parti-game (LLPPG) with a Learned Local Controller (LLC).

Input: A kd-tree representing the current partitioning of the state space.
Input: A database containing some of the experiences seen thus far (used by the OUTCOMES function).
Outputs: The modified input kd-tree and experiences database.
Parameter A: finite set of allowable actions
$s \leftarrow$ InitialState
GOAL $\leftarrow$ GoalCell
while $s \notin$ GOAL **do**
  $i \leftarrow$ cell containing s
  Compute J_{WC}, for each cell
  if $J_{WC}(i) = +\infty$ **then**
    Partition the largest losing cells that fall on the win-lose boundary and discard experiences associated with them
  else
    $j \leftarrow \text{MinMaxNeighbor}(i) = \underset{j' \in \text{NEIGHS}(i)}{\arg\min} \underset{k \in \text{OUTCOMES}(i,j')}{\max} J_{WC}(k)$
    $p \leftarrow$ a state in j that is at a distance $\alpha \ll 1$ from both the shared hyper-surface between i and j and the center of this hyper-surface.
    while not stuck, s is still in i **and** timeout did not expire **do**
      if random(1.0) $\leq$ PE(s) **then**
        $a' \leftarrow$ action $a \in A$ selected with probability PE(s, a)
      else
        $a' \leftarrow \underset{a \in A}{\arg\min} \| \vec{p} - (\vec{s} + \text{F}(s, a)) \|$
      end if
      Execute action a' for the predefined time constant Δt, ending in state r
      Update F with the low-level experience pair $((s, a), (\vec{r} - \vec{s}))$.
      Update both MSE and MaxSE with the low-level experience pair $((s, a), ([\text{F}(s, a) - (\vec{r} - \vec{s})]^2))$, according to update rules (5) and (6), respectively.
      $s \leftarrow r$
    end while
    $k \leftarrow$ cell containing s
    add k to OUTCOMES(i, j)
  end if
end while

in a cascading manner. If t tilings are used, the total number of table cells needed will be tnp^{m+n}. And since we represent four distinct functions, F, MSE, MaxSE and n, we will require four times that amount, $4tnp^{m+n}$

Thus, although tile coding techniques greatly reduce the space requirements needed to represent the functions being approximated, the space required is still exponential in the number of dimensions of the state and action spaces. For example, for our nine-joint robotic arm problem, both the state and action spaces are nine-dimensional. If we use only four tilings with only one split in each dimension, representing E, MSE, MaxSE and n will require $4 \times 4 \times 9 \times 2^{18} = 37{,}748{,}736$ table cells. If four bytes are used to represent a floating point number, the tile coding approximator will require about 150MB of storage.

While this amount of memory could be manageable by today's computers and is much less than what would be required by an algorithm that requires fine-grained discretization of the state space, it is still high given the very low resolution into which the tilings are divided. Furthermore, the growth of space requirements is still exponential. Even with such very low resolutions, each additional dimension under a similar setup would double the memory requirements, quickly approaching the practical limits of today's computers.

In reality, the trajectories covered by the agent while searching for the goal

might only span a small fraction of the state space. If *a priori* information is available about the dynamics of the system and which regions of the space will be important in finding the final trajectory, the function approximator can be designed in such a way that focuses representational capacity in important parts of the space while providing much less capacity to others, thus leading to less overall memory use. However, for problems for which it is difficult to design a local controller, it is likely that such domain knowledge is unavailable or at least inadequate for these purposes. We would still like to be able to use the algorithms introduced in this dissertation with a LLC in higher dimensional problems for which little or no knowledge about the dynamics is available.

Tile coding literature (e.g., Miller, Glanz, and Kraft (1990) and Sutton and Barto (1998)) addresses the problem of the still exponential growth of the number of table cells needed by using tables of limited, smaller sizes than required by a given problem and using a hashing function that maps coordinates from the space of the ideal table size to those that fall within the ranges of the tables actually allocated. These techniques handle collisions by simply ignoring them, thus forcing different areas of the state space (as determined by the hashing function) to share some of the same cells in representing their value. This is regarded as another mechanism through which generalization is achieved across non-spatially related areas of the

state space, although, at face value, the unprincipled coupling of values from different areas of the state space one would question the effectiveness of this technique as a generalization mechanism given that it would necessarily introduce somewhat arbitrary bias in the learned function and thus may adversely affect performance. While this technique has not been closely studied, it has been effectively used to reduce the storage requirements of tile coding approximators. However, we did not utilize it in the work described in this dissertation.

Instead, we address the problem of the exponential growth of the space requirements needed to represent the model of the dynamics used by the LLC using a different method that does not force the introduction of such bias and guarantees allocating the desired amount of representational power to every part of the space if it is ever needed. We achieve this by calling on a particular feature of the problems to which our algorithms apply.

Only a small fraction the state space is ever visited while trying to solve one of the problems we address using the algorithms introduced in this dissertation. Furthermore, since the LLC only needs to test actions in its immediate locale, it could achieve the same results if we only represent information about the dynamics in those areas that have ever been visited by the agent.

The problem of storing only a small fraction of the cells of a large, high-dimensional tables, which is what is needed in this case, lends itself very cleanly to using sparse table allocation techniques. We chose to implement the tile coding table cells as hash tables, with the indices into the tables as keys, but using an implementation that handles collisions properly, i.e., one that does not result in sharing of cells.

Specifically, we used the hash table implementation that is built into Allegro Common Lisp, which we used exclusively as the implementation platform for our algorithms and for producing all the results of this dissertation. For the nine-joint robotic arm benchmark problem and using the setup described above, this technique resulted in using only 30,596 hash table entries before a stable solution was found. Since each table entry holds a list of four numbers, if we assume four bytes per number, this would mean that a total of 489,536 bytes were needed to solve the nine-joint-arm problem. This is a very small fraction (3.1×10^{-3}) of the 150MB that would be needed if memory is allocated for all tiles, regardless of whether the agent will ever visit the parts of the state space they represent.

6.4 Simulation Results

We now present the experimental results of Algorithm 6, which from this point on we will refer to as LLPPG+LLC.

Each of the following subsections is dedicated to describing the results of applying LLPPG+LLC to one of our benchmark problems and comparing the results to those obtained by the algorithms that used a pre-defined local controller. Each will start by introducing the design of the tile coding function approximator used in the particular LLC by specifying the tiling parameters used. As mentioned earlier in this chapter, and as we will see in each of the following subsections, we intentionally chose the parameters in each case without paying close attention to the characteristics of the problem at hand, and only took into account each problem's relative complexity. This will be seen in the choice of the number and configuration of each tiling used to approximate F, MSE and MaxSE and n. The adjustable parameters of the algorithm were set the to same values for all problems, with F's learning rate $\eta = 0.1$, the MSE exponential decay rate $\gamma = 0.5$ and $\epsilon = 0.05$. The initial values for all functions learned by the tile-coding approximator (F, MSE, MaxSE and n) will be zero. Since our algorithms choose actions based on a prespecified order in the case of ties, this will cause actions at a particular cell to be taken in order The remaining aspects of the design of each LLC used were identical in all problems

and are as specified in Section 6.3.2.

After describing the problem-specific LLC parameters, each section will present the results of applying LLPPG+LLC to the given problem and compare them to the results obtained when an engineered local controller was used. Since learning the LLC involves probabilistic choice of actions, the solutions LLPPG+LLC achieve can be different from trial to trial. Therefore, in order to assess the performance of this algorithm on the benchmark problems relative to the previously presented algorithms, we will present the results averaged over 40 complete iterations, each involving running the algorithm repeatedly until a stable solution is reached.[3] Among all previous results, of particular interest to us will be the results of LLPPG, because this algorithm only differs from LLPPG+LLC in the local controller used, so it will most clearly show the effects of using an LLC instead of an engineered controller.

6.4.1 Mazes

We begin with our simplest set of benchmark problems, the continuous mazes presented in section 3.2.1. Since the mazes are two-dimensional, the functions F, MSE, MaxSE each have a cardinality of two, which means that each of these

[3] We will elaborate on the stability of LLPPG+LLC's solutions in Section 6.4.3.

functions is bi-valued and thus we need to use two tile coding approximators (two sets of tilings) to represent them.

Since the action space is one-dimensional, each of these functions will take three inputs, and thus the tilings to use have to be three-dimensional. Because of the simplicity of the local controller needed for navigating these mazes, we used only one three-dimensional tiling per tile coding approximator. The first two dimensions, each spanning $[0, 10]$, represent the two-dimensional state and the third, $[0, 2\pi]$, represents the one-dimensional action space. The first two dimensions of a tiling were not sub-divided at all, using one large tile to cover the whole space in both of these dimensions. The third dimension was divided into eight sub-intervals representing the eight actions into which we discretized the action space. These eight actions are $\{\frac{\pi}{4}i, i = 0, \ldots, 7\}$.

Results

To see how LLPPG+LLC behaves in a maze, we present Figure 18, which shows the trajectory taken by the agent and the partitioning at the end of each of the eight trials needed to reach a stable solution in a typical iteration of LLPPG+LLC. The effect of taking exploratory actions is especially visible in Figure 18(a), where we can see the many clearly "off-track" steps taken by the agent when it takes

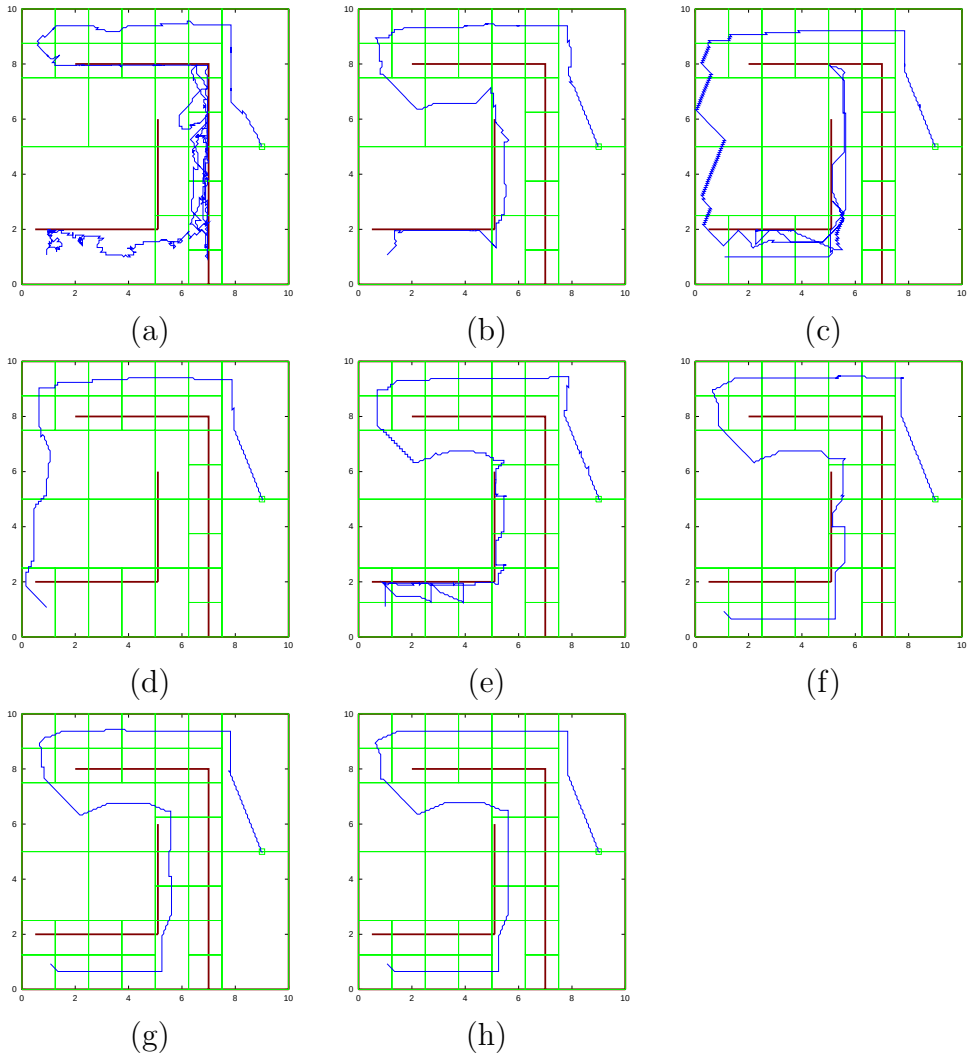


Figure 18: The eight trials required to stabilize LLPPG+LLC applied to maze a. Each figure shows the trajectory taken to the goal and the partitioning at the end of the trial.

exploratory actions. We can also see that the frequency of these off-track steps gets smaller as time goes by, as the agent gains more confidence in the effects of actions and the probability of taking exploratory actions gets smaller.

Table 4 shows the results of applying LLPPG+LLC to the set of maze problems and the puck-on-a-hill problem, along with the results of applying the algorithms from previous chapters, for ease of comparison. For the four maze problems, we can see that LLPPG+LLC did very well in comparison to the prior algorithms even though the algorithm started with absolutely no domain knowledge and it had to learn its own local controller. In most cases, LLPPG+LLC outperformed parti-game in the criteria used to measure the performance of the various algorithms. The only measure in which LLPPG+LLC never outperformed PG was the number of trials needed to reach a stable solution, which is not at all surprising given the exploration LLPPG+LLC performs.

6.4.2 How LLPPG+LLC Can Outperform LLPPG

Table 4 also shows that LLPPG+LLC even outperformed LPPG and LLPPG in Maze b in the number of partitions generated and total steps taken in the environment. While this looks surprising at first glance, it is, nonetheless, explainable, and can be expected in some cases. Given the minimax nature of these algorithms,

Problem	Algorithm	Trials	Partitions	Total Steps	Final Trajectory Steps
maze a	PG	**3**	444	35,131	279
	LPPG	**3**	239	16,652	256
	LLPPG	**3**	**27**	**1,977**	**270**
	LLPPG+LLC	9	34	3,536	298
maze b	PG	6	98	5,180	183
	LPPG	5	76	7,187	175
	LLPPG	6	76	5,635	**174**
	LLPPG+LLC	6	**52**	**3,256**	193
maze c	PG	3	176	7,768	416
	LPPG	**2**	120	10,429	**165**
	LLPPG	**2**	**96**	**6,803**	**165**
	LLPPG+LLC	6	117	10,693	443
maze d	PG	**2**	1,194	553,340	**149**
	LPPG	**2**	350	18,639	155
	LLPPG	**2**	**21**	**1,469**	165
	LLPPG+LLC	7	23	1,693	186
puck	PG	6	80	6,764	240
	LPPG	**2**	**18**	**3,237**	**151**
	LLPPG	**2**	**18**	**3,237**	**151**
	LLPPG+LLC	13	97	20,173	229

Table 4: A repetition of some of the results of Table 2 in addition to the results of applying LLPPG+LLC to the same set of problems. Smaller numbers are better. The results of LLPPG+LLC are averaged over 40 experiments, each run up to the first instance of stability. The averages are rounded to the nearest integer. Best numbers for each problem in each column are shown in **bold**.

one algorithm can outperform another (and even the same algorithm can outperform itself from one iteration to another) based on whether or not some particular negative experience was encountered.

Consider two minimax-based algorithms, A_1 and A_2, that use the same methods for computing cell costs, choosing target cells to aim for and the state to which to aim within a chosen target cell. Now consider running these two algorithms on a continuous maze problem of the type we presented above, starting from the same initial conditions, i.e., the same partitioning, set of past experiences and current state (and, therefore, cell). Let us call this state s_0. Let the two algorithms use almost idential local controllers, i.e., both choose low level actions in exactly the same way and are capable of reaching the target state, barring the existance of any barriers that might be in the way. The only difference between the two local controllers is that A_2's local controller uses a random action with some small probablility, p.

Let us assume, without the loss of generality, that s_0 is the start state of the problem. Let the current cell (the cell that contains s_0) be cell a and assume that it has only two neighbors, b and c. Assume further that the optimal path to the goal, which starts in a, entails aiming for and successfully entring cell c. However, a path through b currently seems more promising because b has a lower cost than

c, however a barrier completely blocks the agent from entring b from a. Both A_1 and A_2 will aim toward b. Let us assume that both local controllers will use exactly the same set of low-level actions until they hit the a-b barrier. At this point, both algorithms will discover that they can not get to b from a, recompute cell costs and then decide to aim for c. Assume further that a barrier partially blocks entry from a to c, and that if the agent aims directly from its current state (the state at which it was blocked while aiming for b) to the default aiming state in c that it will hit this a-c barrier and will conclude that it can not go from a to c. However, assume that the barrier between a and c would not be encountered had the agent aimed directly from s_0 to the default aiming state in c. Now, if A_2 takes a random action on its way to c, it is possible that it could miss the barrier and enter cell c.

The previous scenario will cause A_1 to partition a because it believes that it is a losing cell, while A_2 will not. That is, A_2's exploratory action caused it to be ignorent of the existence of the a-c barrier and, therefore, it did not have to do anything to circumvent it, resulting in it requiring less partitioning (and, most likely, less steps taken in the environment) than A_1, up to the point of entering c.

Since LLPPG+LLC has a random component in the way it chooses actions compared to LLPPG, the previous argument can explain the case where LLPPG+LLC outperformed LLPPG. Of course, taking exploratory actions does not always cause

such "lucky breaks" and, indeed, that is why the performance of LLPPG+LLC is not expected (and in reality usually does not) exceed that of LLPPG.

6.4.3 Stability of Solutions Under LLPPG+LLC

The stability of a solution using any of the algorithms introduced in this dissertation, without the use of an LLC, is attained when no new cell-level experiences are encountered by the agent on its way to the goal. The addition of the LLC adds another condition for attaining a stable solution: exploration probabilities have to be equal to zero at all states in which the agent passes on its way to the goal. For the maze results described above, it was not the case that exploration probabilities had reached zero. Therefore, the solution shown in Figure 18(h) is not a truly stable solution. It is only a *temporarily* or *relatively stable* solution in the sense that the agent was observed to have taken the same actions and thus followed the same trajectory twice, and it would likely repeat that a number of times before it decides to take an exploratory action. When an exploratory action is finally taken, the agent might then deviate from that temporarily stable solution. The number of steps taken between exploratory actions will continue to grow as exploration probabilities get smaller, which means that the amount of time during which a

solution is stable will continue to grow as time goes by. Since exploration probabilities are guaranteed to converge to zero given the update rule in Equation (5) and the way we compute exploration probabilities, the agent will ultimately find a truly stable solution. For the remainder of this dissertation, whenever we refer to an LLC-based algorithm having stabilized at a solution, we will be referring to this type of temporary or relative stability.

The reason for our choice of this type of stability for comparing this algorithm's performance to the performance of its predecessors is that it is the most natural and reasonable point of comparison. This is because running the algorithm for more iterations until real convergence is reached (or approached) will most probably not change the form of the final solution. Furthermore, although running the algorithm for more iterations will increase the number of trials, partitions and total steps, it will also most likely cause the average number of steps per trial to decrease as time goes by. This holds true for the previous versions of the algorithm as well; we can keep running more trials after stabilization to improve the average number of steps per iteration. Therefore, running the algorithm past its initial temporary convergence could actually bias the results in favor of the LLC-based algorithm.

6.4.4 Puck on a Hill

The puck-on-a-hill problem was introduced in section 3.2.1. Like the continuous mazes, this problem is two-dimensional, therefore each of the functions we need to learn has a cardinality of two.

The general aspects of the dynamics of this problem is more complex than that of our continuous maze problems. That is, although the barriers in the maze problems make the dynamics non-linear and discontinuous, the overall trends of the maze dynamics (disregarding barriers) are linear and the effects of actions are uniform throughout the state space. The dynamics of the puck problem, however, are non-linear and the effects of actions vary considerably from one region to another.

For that reason, we chose to use a more complex structure for the tile coding approximators used in the LLC for this problem. To start, we used four three-dimensional tilings per tile coding approximator, versus using only one for maze problems. The three dimensions represent the two-dimensional state space and the one-dimensional action space. While we did not make any splits in any of the state dimensions in the maze problems, all dimensions were split in this problem. The first dimension, the horizontal position of the puck, which spans the interval $[-20, 20]$, was split into two equal length sub-intervals. The second, the velocity

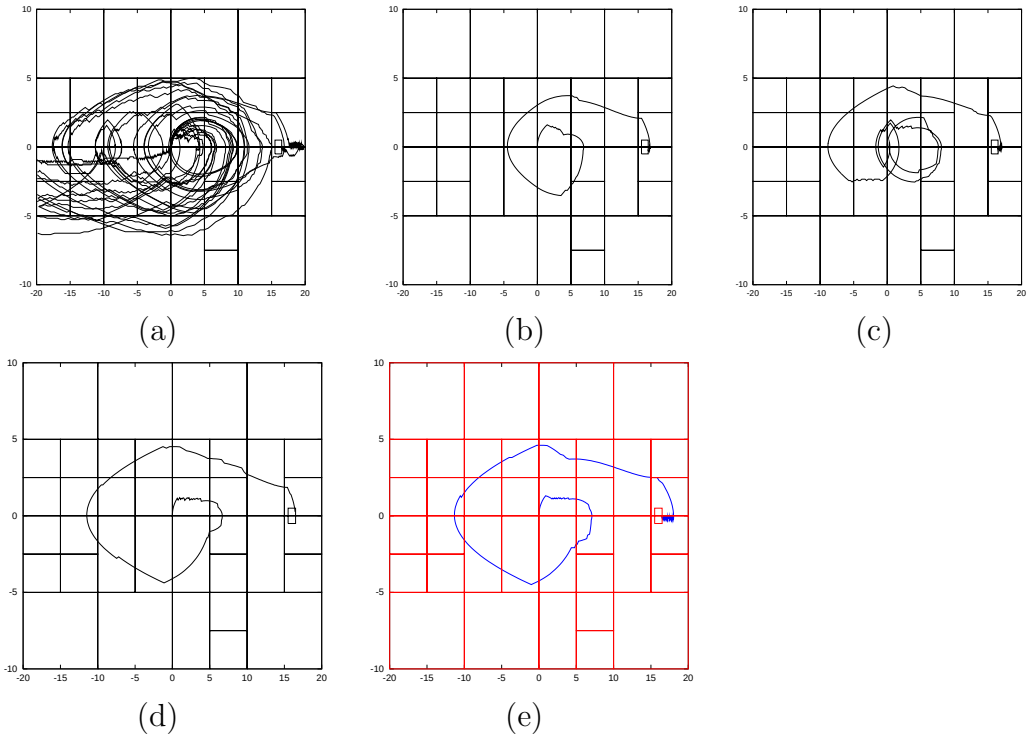


Figure 19: The results of the five iterations needed by LLPPG+LLC to stabilize at the non-optimal form solution in (e) above.

dimension, which spans $[-10, 10]$, was split into eight equal length sub-intervals. Finally, the action dimension (the thrust exerted by the puck), which ranges from -1 to 1, was split into three equal length sub-intervals to represent the three allowable actions, -1, 0 and 1.

Results

Figure 19 shows the five trials that comprise one iteration of applying LLPPG+LLC to the puck problem until the first instance of stabilization. Table 4 shows the average performance of the algorithm over 40 iterations. It is clear from Figure 19 that the algorithm did not find an optimal-form solution in this iteration, but rather found a sub-optimal form solution in which the puck starts out going to the right, then heads back left far enough to be able to head right again and go up the hill with enough velocity to reach the goal.

Because sub-optimal solutions were sometimes reached by LLPPG+LLC on this problem, the average final trajectory length of 229 steps achieved by this algorithm is higher than the optimal-form final trajectory length of 151 reached by LLPPG (and LPPG). The final solution in Figure 19(e), which is a typical sub-optimal solution, has a length of 470. Since the average final solution length over 40 iterations is 229, it is clear that optimal-form solutions were reached in the majority of the cases.

Interestingly, despite the fact that sub-optimal-form solutions were sometimes reached by LLPPG+LLC, its average final solution length on this problem is better than the final solution length that was reached by PG.

However, the algorithm did not outperform any of the previous algorithms in

any other criterion. The average number of iterations needed for initial stabilization is almost twice as big as that of PG, which required the most iterations among the previous algorithms. The number of partitions needed by this algorithm is also larger than all other algorithms, surpassing the 80 partitions needed by the previous worst performer, PG, by 17, requiring a total of 97 partitions. Finally, the average total steps taken in the world is much higher than all other algorithms. Of course, the high number of trials run before stabilizing at a solution, and, consequently, the high total steps taken in the world, are not surprising given the fact that this is an exploratory algorithm while the others are not.

The fact that LLPPG+LLC required more trials before it stabilizes does not necessarily mean that the algorithm was finding bad solutions up to that point. Therefore, a fair way to compare its performance to the previous algorithms' is to look at the average number of steps per trial. We find that LLPPG+LLC averaged 1,552 steps per trial, while PG averaged 1,127 and both LPPG and LLPPG averaged 1,619. Comparing these numbers we can see that LLPPG+LLC actually performed quite well on this problem given that it had to explore and learn an LLC. However, these numbers may also make it seem like PG is the best performer among the group. However, we should take into account the fact that if LPPG and LLPPG+LLC were run as many times as PG was (which will cause them to repeat

the very good solution they settled on), their average steps per trial would drop considerably.

6.4.5 Application of the Global Optimization Algorithm to the Puck Problem

The solution shown in Figure 19(d) is a natural candidate for using the global optimization algorithm presented in Chapter 5. Thus, we applied that algorithm to this problem after obtaining the solution in Figure 19(d). With LLPPG+LLC as the underlying minimax-based algorithm, and after 16 additional iterations, for a total of 22, the global optimization algorithm arrived at the optimal-form solution and the partitioning shown both in Figure 20. Since the global optimization algorithm was applied using LLPPG+LLC as the underlying minimax-based algorithm, the path improvement seen here was achieved while using the learned local controller.

6.4.6 Nine-Joint Arm

The nine-joint arm is a nine-dimensional state kinematics problem introduced in Section 3.2.1. The action space is also nine-dimensional in this problem, with each action consisting of the nine angle changes that should take place in the next time step. We used only one eighteen-dimensional tiling in this problem. We

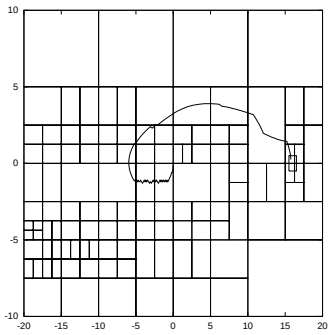


Figure 20: The optimal-form solution to the puck-on-a-hill problem reached after applying the global optimization algorithm of Chapter 5 to the solution reached by LLPPG+LLC shown in Figure 19(d).

did not introduce any splits in any of the state space dimensions, each of which spans the closed interval $[-\pi, \pi]$. That is because the dynamics of this problem are in fact very simple and generally uniform throughout the space, except for the existence of barriers and illegal states that involve intersections between arm segments. Disregarding barriers and illegal states, the delta function, F, that needs to be learned by the tile coding approximator is simply an identity function in the action used.

We defined the set of eighteen actions the agent can choose from, which is the minimum number of actions needed to allow the agent to move in both directions of each of the state space dimensions. Each nine-dimensional action has only one none-zero component that is equal to $\pm c_a$, where $c_a = \frac{1}{32}$. This set of actions allows

the agent reversible movement in the state space, which is important for exploration and maneuverability, and the small step size guarantees the agent's ability to enter the small goal region, which, as we saw in Section 3.2.1, is a hyper-rectangle defined by nine intervals in $\mathcal{R}$, each centered around a goal coordinate and spans 0.075 radians on each side. This set of 18 actions is clearly more primitive than the array of actions the *a priori* controller has access to. The *a priori* controller can change the state in more than one dimension at a time and it can also vary the step size (although there is a maximum of $\frac{\pi}{15}$ imposed on the size of the action vector used). The superiority of the set of actions available to the *a priori* controller is expected to give it an advantage against algorithms that use this fixed set of 18 actions. This will be further discussed below.

Results

The results for the nine-joint arm problem which were shown in Chapter 3 used the greedy controller introduced in the same chapter. The criteria we have been using to assess the performance of the different algorithms included the total number of steps taken in the state space and the number of steps constituting the final, stable trajectory. While, as we said above, we used a set of eighteen constant actions with the LLC—with each of these actions having only one non-zero component—the

engineered controller introduced in Chapter 3 uses actions in which the length of each component can vary in size. Because of this, comparing the number of actions taken by the two different types of controllers would not be meaningful.

To facilitate the comparison, we make the following assumption about the dynamics of this problem. We assume that the rate at which each of the nine angle dimensions changes in response to taking an action is constant throughout the state space, regardless of whether or how other coordinates are being changed at the same time. This would mean that the time needed to complete an action is proportional to the component of that action that is largest in absolute value.

This also means that all LLC-type actions take an equal (constant) amount of time to be performed. For ease of comparison, we will take this constant amount of time to be the basic time unit for comparison in this problem and we will express the time taken by the variable-sized actions of the *a priori* controller in terms of this unit as well. If the *a priori* controller takes an action when the agent is in state $s = (\theta_{s_1}, \ldots, \theta_{s_9})$ that ends with the agent entering state $s' = (\theta_{s'_1}, \ldots, \theta_{s'_9})$, then we will say that this state transition lasted $\frac{1}{c_a} \max_i(|\theta_{s'_i} - \theta_{s_i}|)$ time units, where $i \in \{1, \ldots, 9\}$ and c_a is $\frac{1}{32}$, as mentioned above. We chose to make the assumption explained above and to compute the time taken by each of the *a priori* controller actions in this fashion because it puts the *a priori* controller-based results in the

best light. This is done with the aim of not biasing the comparison in favor of the learned controllers we are presenting. Other assumptions about the time it takes to execute one of those actions, e.g., assuming that it is proportional to the size of the action vector, would make the LLC results look better.

Table 5 shows the results of applying LLPPG+LLC to the nine-joint arm problem along with the results of applying the algorithms from previous chapters, for ease of comparison. The two columns titled "Total" and "Final Trajectory" are each split into a pair of columns, with the left sub-column showing the number of steps taken by the *a priori* controller and the right one containing the number of time units used, rounded to the nearest integer. In cases in which the *a priori* controller was used, the number of steps taken by this controller are shown in the Steps column and the times, calculated as shown above, are listed in the Time columns enclosed in square brackets to indicate that they are calculated as opposed to measured. Conversely, when the set of simple 18 actions were used, the Steps columns show a "–" while the Time columns contain the measured time units spent. The first three rows of Table 5 repeat the results of PG, LPPG and LLPPG, but add the calculated total time units used until stabilization, and those used by the final solution settled on by the algorithm.

As mentioned above, the set of actions available to the *a priori* controller is more

versatile and flexible than the eighteen constant actions available to LLPPG+LLC, and, therefore, algorithms using the *a priori* controller are expected to have an edge when comparing the lengths (or elapsed times) of paths taken. In order to provide a better reference point to assess LLPPG+LLC's performance, especially the impact of having to learn its own local controller on the performance, we also ran LLPPG with a local controller that simply performs one-step lookahead in the real model used to simulate the environment, using the same set of 18 simple actions that the LLC uses. The results of this implementation are shown in the fourth row of Table 5. Finally, the last row of the table shows the performance of LLPPG+LLC on this problem, averaged over 40 iterations. These last two rows of the table do not have entries for the number of *a priori* controller-type (variable size) steps, because the simple, LLC-type actions were used. Therefore, only the Time sub-columns are populated.

Analysis of the Results

For this analysis, we will base our comparisons of total and final trajectory lengths on the Time sub-columns of these respective criteria, since it is the only measure that uses the same units for both types of action sets employed by the three different

	Algorithm	Trials	Partitions	Total		Final Trajectory	
				Steps	Time	Steps	Time
1	PG	36	117	7,112	[16,489]	103	[209]
2	LPPG	13	52	5,499	[12,045]	**101**	**[173]**
3	LLPPG	**4**	**39**	**3,229**	**[7,549]**	110	[223]
4	LLPPG w/ 1-step lookahead in true model	9	40	–	7,600	–	584
5	LLPPG+LLC	14	59	–	15,525	–	717

Table 5: The results of applying the various algorithms to the nine-joint arm problem. The results of LLPPG+LLC are averaged over 40 experiments, each run up to the first instance of stability. The averages are rounded to the nearest integer. Smaller numbers are better and the best numbers are shown in **bold**. The numbers in rows 1-3 were obtained using the engineered controller (described in Section 3.2.1) which uses variable size actions, as opposed to the fixed size actions used by the LLC-based results in rows 4 and 5. To facilitate comparing the two sets of results, we computed the time taken by the engineered controller (shown in brackets). See the text for a complete explanation.

local controllers used.[4]

As expected, the results of the algorithm in row 4, which uses one-step lookahead, using the LLC-type actions, into the true model as its local controller, was outperformed in the length of the final trajectory by all previous algorithms that use the more effective, variable size, multi-dimensional actions. LLPPG with the perfect look-ahead controller found the slowest solution seen thus far, lasting about 2.4 times longer than the best time, which was found by LPPG. This is clearly

[4]It is interesting to note that when the calculated time criteria are used in comparing the three algorithms studied in previous chapters (in rows 1–3), these algorithms retain the same ordering they received when the number of steps taken by the *a priori* controller was used as the comparison criterion.

caused by the fact that the set of actions available to this controller only change one dimension at a time compared to the *a priori* controller's ability to change the state in all dimensions at once. We can easily choose a different set of actions for use by look-ahead-type controllers that would have better performance and could probably outperform the *a priori* controller's set of actions. For example, a relatively small set of specially designed overall curling and uncurling actions could be used that would reach the goal in far fewer time steps. However, such a set would be chosen with the help of our knowledge of the characteristics of this domain and the problem we are solving. Therefore, it would conflict with our goal of introducing an algorithm that can be used when little or no domain knowledge is available to help in designing a greedy controller.

The algorithm does not perform too badly based on the other criteria. It stabilizes faster than PG and LPPG and requires less partitions to stabilize than they do. Furthermore, it beats both algorithms in the total time spent and almost ties LLPPG in this criterion as well.

Finally, we look at the performance of LLPPG+LLC in row 5 of Table 5. Compared to LLPPG with the perfect one-step look-ahead controller, LLPPG+LLC arrives at a solution that is about 23% longer, taking a time of 717 compared to 584, which is not a bad outcome considering that it had to learn the local controller

from scratch. The algorithm outperforms PG in the number of trials, number of partitions and total time required to stabilize. Furthermore, its performance is not too far off that of LPPG in the number of trials, number of partitions and total time needed to stabilize. Thus, even though this algorithm needed to learn its own local controller, the added cost of doing that, as measured by its effect on the number of trials, partitions and total steps needed before stabilization, still left the algorithm with better performance than PG and just slightly lower performance than LPPG, both of which are algorithms that did not need to explore the space in order to build their own local controller.

6.4.7 The Acrobot

We saw in Section 3.3.4 how neither the algorithms introduced in this dissertation nor PG were able to stabilize at a solution in the acrobot problem and we presented a theory about why that is the case. Of course, we would not expect LLPPG+LLC to do better than any of these algorithms. The question is: how much worse would the behavior of LLPPG+LLC be, since it would have to learn the local controller from scratch, and with which the agent would start with a completely ignorant controller?

Since the acrobot has a four-dimensional state space and a one-dimensional

action space, we used four tile coding approximators, each having five inputs. We used four tilings per approximator, and, in each tiling, state dimensions were each split into four sub-intervals and the action dimension was split into three sub-intervals, to allow for the three actions -1, 0 and 1.

Surprisingly, as Figure 21 shows, the performance of LLPPG+LLC was not discernibly worse than the other algorithms that started with the engineered local controller, which performs one-step look-ahead using the true model of the dynamics. The first iteration of LLPPG+LLC takes the longest amount of time to reach the goal among all algorithms. However, from that point on, we can see that LLPPG+LLC starts to reach the goal in times that are often competitive with the other algorithms, although, as is the case with the rest of the algorithms, its performance oscillates between good and bad solutions. This suggests that LLPPG+LLC was quickly able to learn an adequate local controller that allowed it to perform favorably compared to the more knowledgeable competition.

Furthermore, we can see in Figure 22 that the growth in the number of cells when using LLPPG+LLC is very close to that of LLPPG. This is further evidence that the algorithm quickly learned an LLC that enabled it to perform as well as the local controller that performs one-step look-ahead in the true model.

6.5 Discussion

The resolutions of the tile coding approximators we used in the above experiments are noticeably lower than what is typically used by RL researchers for representing approximations of optimal value functions for the same (or similar) problems. The reason we are able to use such low resolutions is the fact that we are using tile coding approximators for representing the dynamics of the problems being solved (specifically, state change functions) as opposed to value functions. While target optimal value functions usually have discontinuities that need to be adequately represented by the tile coding approximator in order for the RL algorithm to converge to a solution, the dynamics of the systems involved, represented as state change functions, may have little or no discontinuities, and can generally be smoother functions. Having to sufficiently represent discontinuities in value functions dictates using high resolution tile coding, at least in the regions in which the discontinuities fall.

The algorithms discussed in this dissertation compute cost-to-goal functions, which obviously have discontinuities in the same regions of the space where a corresponding value function does. The representations of cost-to-goal functions are dynamically computed, and at a higher level, using a coarse kd-tree-based partitioning and a minimax algorithm. The two levels—the tile coding approximator and the kd-tree-based partitioning—combine to define the solution to the problem being tackled.

6.6 Contributions

The contribution of this chapter is introducing a method for online learning of local controllers that fully integrates with the the algorithms introduced in this dissertation, thus allowing them to be used without having to specify *a priori* controllers. This brings this class of algorithms closer to the ultimate objective of being able to operate in completely unknown environments. The human designer of a solution need only choose a set of representative actions for the LLC to use and specify the number of tilings and the resolutions to which each dimension is split for the purposes of learning a local controller. This usually requires much less effort and domain knowledge on the part of the designer than what is required to design a local controller.

The added space requirements for using an LLC are relatively quite small and should not have any significant impact on the scalability of the algorithm with which the LLC is used. This is due to the fact that we are able to use tile coding approximators that have relatively coarse resolutions and that we only allocate memory for the subset of tiles in which states visited by the agent fall.

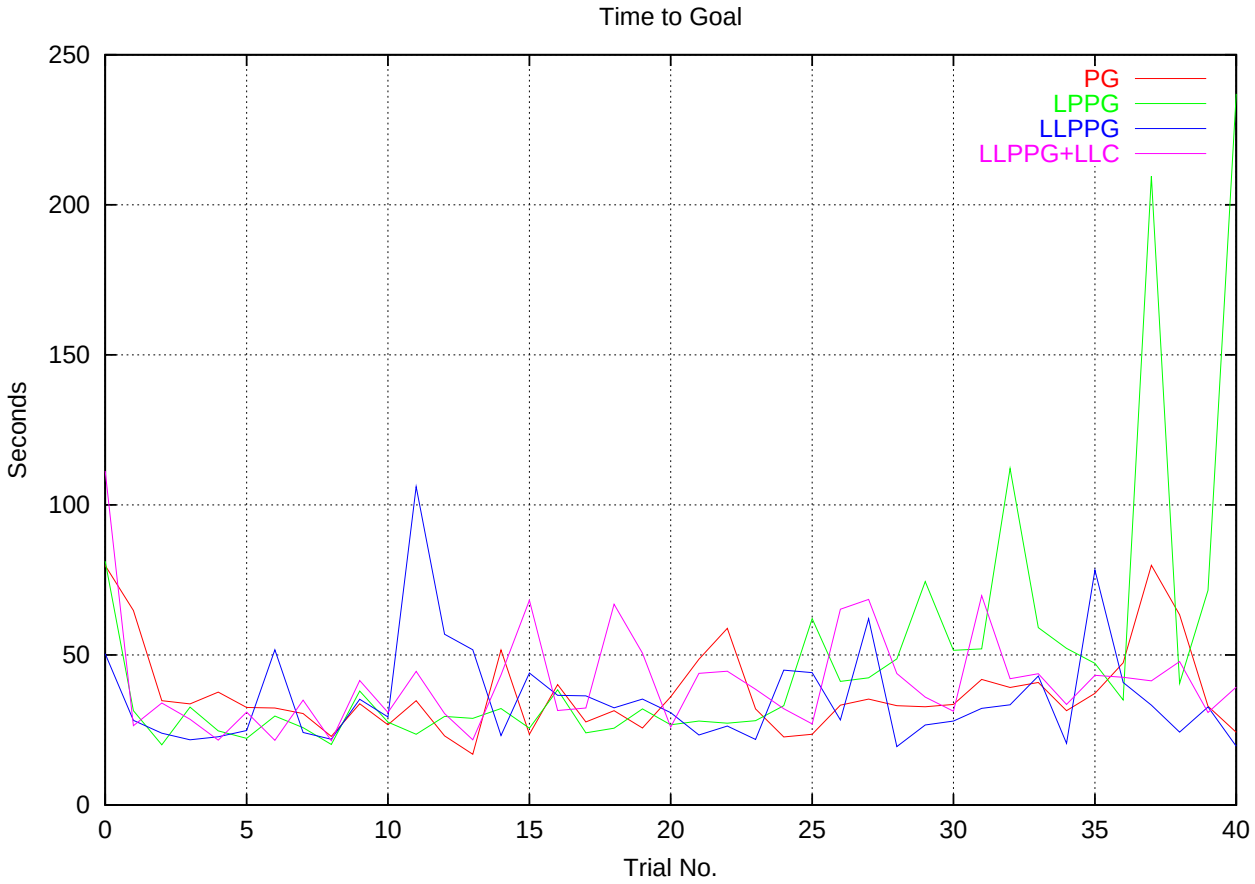


Figure 21: Graph showing the number of seconds needed to reach a solution by PG, LPPG, LLPPG and LLPPG+LLC as a function of trial number when applied to the Acrobot problem.

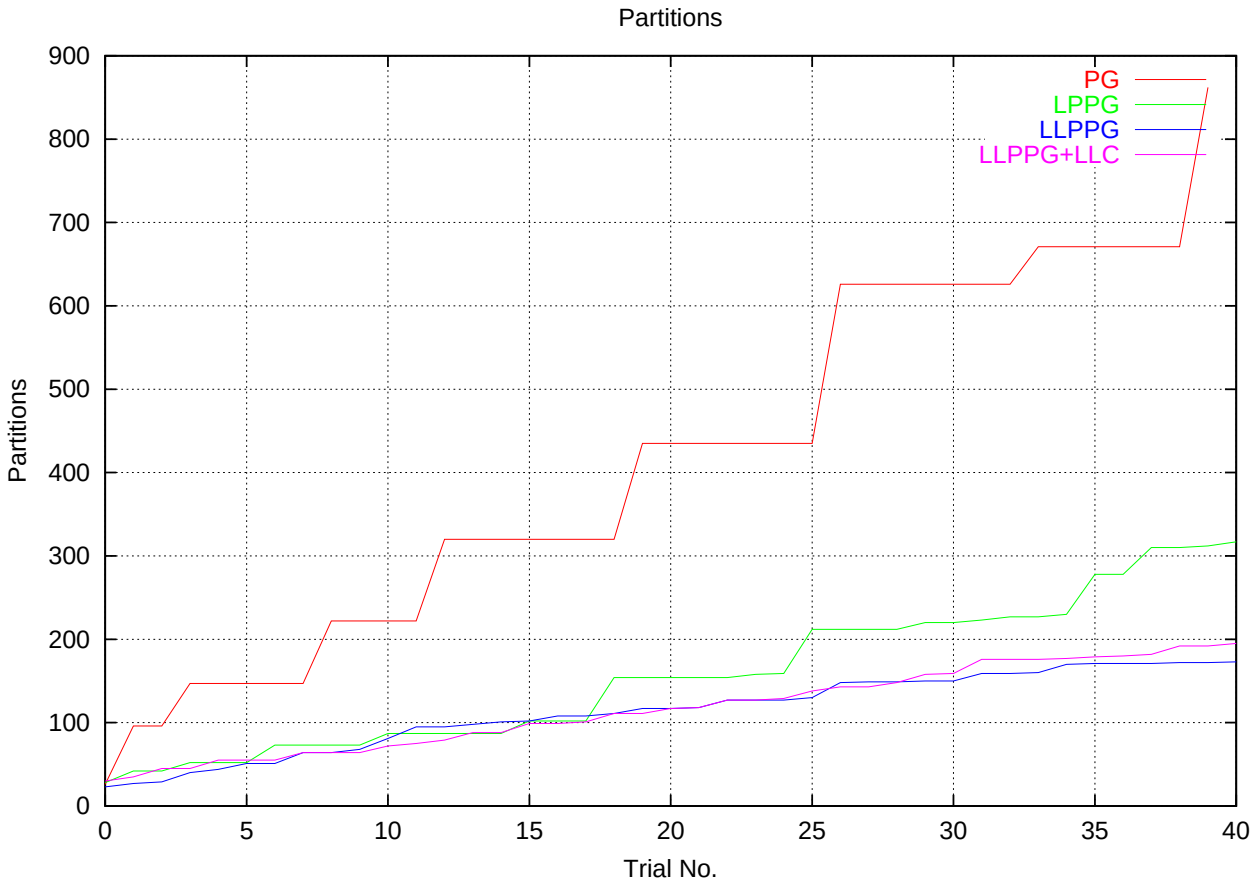


Figure 22: Graph showing the number of partitions needed by PG, LPPG, LLPPG and LLPPG+LLC as a function of trial number when applied to the Acrobot problem.

Chapter 7

Conclusions and Future Directions

7.1 Summary of Contributions

This dissertation introduces a family of variable resolution algorithms that forms a powerful, *a-priori*-model-independent paradigm for finding high quality solutions for deterministic, continuous, goal-type Reinforcement Learning problems. Compared to the parti-game algorithm, the algorithms introduced in this dissertation find better solutions more efficiently, more reliably, with better scalability and less reliance on *a priori* domain knowledge.

Specifically, the algorithms introduced in this dissertation achieve the following goals.

Fast, Consistent, On-Line Discovery of Optimal-Form Solutions. Chapter 3 presents two algorithms, LPPG and LLPPG, which can find "good" solutions to problems from this class relatively quickly, often within few iterations and often outperforming parti-game in the number of iterations and total steps taken (more about the number of partitions below). By "good" solutions we mean solutions with trajectories that are generally direct and do not meander needlessly. However, these solutions can be of sub-optimal form. Chapter 5 introduces a global optimization algorithm that constitutes a general mechanism that can be used with the LPPG and LLPPG (as well as PG) to find solutions of optimal form.

Furthermore, LLPPG is shown to be less sensitive than PG to the configuration of the particular problem being solved and, therefore, exhibits more consistent behavior over ranges of problems and can outperform parti-game by orders of magnitude in certain problem configurations.

Low Space Requirements. LPPG and, to a greater degree, LLPPG, also have lower space requirements than PG because they usually find solutions at lower resolutions (i.e., using fewer partitions). Since PG has been shown to have good scalability, requiring fewer partitions (and thus less space) means that these algorithms are better able to scale to larger problems than PG.

Low Time Requirements. The algorithms for minimax cost computation introduced in Chapter 4 lower the per-step time requirements of these algorithms dramatically, thus making them much better suited for on-line control. This is achieved by an algorithm that performs a fraction of the total number of backups needed to compute cell costs in one pass after every new experience is received. The chapter proves that both the one-pass cost computation and the truncated backups algorithms result in the same agent behavior under a given algorithm (LLPPG, LPPG or PG).

Because the fraction of backups performed by the truncated backups algorithm gets smaller as the number of cells and/or dimensions grows, this strongly contributes to the superior scalability of the family of algorithms we present in this dissertation.

Minimal Domain Knowledge Requirements. Finally, Chapter 6 introduces a general scheme for eliminating the need for the pre-engineered local controller that parti-game requires. This is done by online building of a learned local controller (LLC) from low level experiences. This scheme could be incorporated into LLPPG, LPPG or PG. The chapter introduces one specific implementation of this scheme incorporated into LLPPG, termed LLPPG+LLC, and shows that it can be used without incurring a big hit in performance. In fact, in many cases, LLPPG+LLC

outperforms LPPG or PG (or both) in many of the comparison criteria, with one case of it even outperforming LLPPG in two of the criteria.

Using this method reduces the amount of *a priori* domain knowledge required to tackle the problem at hand. The solution designer only needs to supply a representative discrete set of actions and the number of tilings as well as the number of splits along each dimension of the tile coding approximators used by the LLC. This information is usually much easier to supply than having to design a complete local controller.

Our experiments show that using tile coding approximators that utilize identical tilings with relatively coarse resolutions are sufficient for learning good LLC's and result in competitive performance. This is possible because we use tile coding approximators to represent the dynamics' state change function, which generally has fewer (if any) discontinuities and is smoother than the more complex value (or cost) function, for which current RL researchers use tile coding. The task of representing the more complex cost-go-goal function is delegated to the higher level variable resolution algorithm being used.

7.2 Future Directions

There are many attractive directions of pursuit in which one would like to extend the work presented in this dissertation.

7.2.1 Data Efficiency

The data the RL agent collects is very valuable in the sense that it is the basic source of information about how the world in which it is acting and trying to control responds to its actions. Therefore, it seems that the more knowledge a RL agent can extract from the data representing its raw experiences the better and faster it would perform. Parti-game and the algorithms introduced in this dissertation use the data gathered from the environment in computing cell costs. However, these algorithms intentionally discard experiences related to cells that are split in order to promote exploration of the areas where the splits are made. Although "forgetting" experiences in this fashion achieves an important goal, it is quite wasteful of the precious data resource.

One way we experimented with for reusing data is to "recycle" the raw experiences the algorithm at hand had collected into newly created cells by considering them as experiences collected under the new partitioning. This was done by attempting to map the source, target and actual destination states of each triple

representing a raw experience onto the newly formed cell structure and considering an actual experience any such triple for which this mapping could be done onto cells that are neighbors under the new partitioning. Although this sounds like a very good way of reusing data, it suffers from the fact that the data triples do not accurately reflect experiences that can actually be collected in the cells they are mapped to. In our experiments, recycling data in this way caused more dense partitioning and generally more iterations and total steps required before a solution is found.

This suggests that data should rather be retained in a form that can more closely represent actual transitions when it is reused after partitioning. One way this could be done is by storing experiences at the lowest level seen by the agent in some economical form and utilizing it after partitioning is done. Since newly formed cells start with the default assumption that all neighbors are reachable, all the agent could gain from its experiences is to remove default assumptions that prove to be erroneous when they are actually tested. If collected data can be used properly such that erroneous assumptions are discovered without having to try them, we could affect a considerable gain in the performance of these algorithms.

7.2.2 Using Other Variable Resolution Representations

The algorithms introduced and studied in this dissertation can be viewed as algorithms that try to approximately isolate regions in which the system being controlled behaves more or less uniformly with respect to the local policy being followed (i.e., the local controller being used). Although the algorithms succeed in quickly finding good solutions, the hyper-rectangular-shaped regions they use are almost always very rough approximations of the real control regions of the underlying problem. The result of this rather rough approximation was clearly seen in the Acrobot problem, where, due to its chaotic nature, the algorithms we studied could not stabilize at a solution because they continued to add more partitions in an attempt to find better approximations of the control regions that fit the minimax view employed by these algorithms.

For all these reasons, one is driven to believe that a different, less rigid and uniform partitioning of the state space might have a better chance at approximating the control regions of the problem. For example, using radial-basis functions as bases for control regions and using the distance from the centers of these regions for computing probabilities of being in a given control region sounds like a promising direction to pursue. This would have to be done in conjunction with a more relaxed method for representing cell costs than the minimax, best-worst case view used by

the algorithms discussed in this dissertation.

7.2.3 Expanding the Class of Problems

The algorithms introduced in this dissertation arrive at good solutions very quickly and efficiently with minimal *a priori* domain knowledge and with the promise of very good scalability. However, they are only applicable to deterministic, goal-type problems. Finding ways to extend the applicability of these algorithms to a wider class of problems where such superior performance could be realized is a very attractive proposition. In what follows, we touch on what the major obstacles are in the face of expanding the applicability of these algorithms in each of these dimensions.

Stochastic Problems

The minimax approach for computing cell costs considers a cell losing—and thus results in partitioning if the agent ever finds itself in one—if the agent knows of no completely reliable way of getting from that cell to the goal. If any of the algorithms introduced in this dissertation is applied to a stochastic problem unchanged, it would at least result in very dense partitioning and in the worst case would cause indefinite partitioning because it will try (in vain) to find a completely reliable way

to reach the goal from each cell through which the agent passes.

What is needed to effectively handle stochastic problems is a more "relaxed" definition of how a cell's cost is computed with which we can guarantee that indefinite partitioning does not occur. Clearly, this would be closely tied with the variable resolution method we choose to use to address the issues discussed in Section 7.2.2 above.

Non-Goal Problems

The algorithms introduced in this dissertation only apply to goal-type problems. One interesting area of further pursuit is introducing modifications to these algorithms that allow them to be applied to problems with more general reward structures, while retaining their attractive features as much as possible.

One possible extension involves defining artificial goal regions in non-goal problems if we believe the problem can be solved in this fashion. For example, the classical pole balancing problem (Sutton and Barto 1998) can be defined to contain a long goal region, which spans the whole width of the track, but is limited in the other dimensions to angles and horizontal and angular velocities with small absolute values. The algorithm being used can be modified to try to stay in the goal region once it enters it, instead of terminating. We have obtained preliminary

results of such a setup that have shown that an LLPPG-based algorithm manages to improve the time it takes before the pole crashes. However, we have found that the rate of improvement declines quickly, and the algorithm settles on significantly worse solutions than what has been achieved using other methods. Clearly, further research into this area is warranted.

Bibliography

Al-Ansari, M. A. and R. J. Williams (1998a). Modifying the parti-game algorithm for increased robustness, higher efficiency and better policies. In *Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems*, New Haven, CT, pp. 204–209.

Al-Ansari, M. A. and R. J. Williams (1998b). Modifying the parti-game algorithm for increased robustness, higher efficiency and better policies. Technical Report NU-CCS-98-13, College of Computer Science, Northeastern University, Boston, MA.

Al-Ansari, M. A. and R. J. Williams (1999). Robust, efficient, globally-optimized reinforcement learning with the parti-game algorithm. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11. MIT Press, Cambridge, MA.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis and S. Russell (Eds.), *Machine Learning: Proceedings of the Twelfth International Conference*, San Francisco, CA. Morgan Kaufman.

Baird, L. and A. H. Klopf (1993). Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-1147, Wright Laboratory. Wright-Patterson Air Force Base.

Baird, L. and A. Moore (1999). Gradient descent for general reinforcement learning. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11, Cambridge, MA. MIT Press.

Barto, A. G., R. S. Sutton, and C. J. C. H. Watkins (1990). Learning and sequential decision making. In M. Gabriel and J. Moore (Eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, Chapter 13, pp. 539–602. Cambridge, MA: The MIT Press.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice-Hall.

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientific.

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

Boyan, J. A. and A. W. Moore (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen (Eds.), *Advances in Neural Information Processing Systems*, Volume 7. MIT Press.

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems*, Volume 5. Morgan Kaufmann.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning 8*(3/4), 341–362.

Friedman, J. H., J. L. Bentley, and R. A. Finkel (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*(3), 209–226.

Littman, M., T. L. Dean, and L. P. Kaelbling (1995). On the complexity of solving markov decision problems. In *Proceedings of the Eleventh Annual Conference*

on Uncertainty in Artificial Intelligence (UAI–95).

Luus, R. (1989). Optimal control by dynamic programming using accessible grid points and region contraction. *Hungarian Journal of Industrial Chemistry 17*, 523–543.

Luus, R. (1992). On the application of iterative dynamic programming to singular optimal control. *IEEE transactions on automatic control 37*(11), 1802–1806.

Maclin, R. and J. W. Shavlik (1994). Incorporating advice into agents that learn from reinforcements. Twelfth National Conference on Artificial Intelligence (AAAI-94).

Michie, D. and R. A. Chambers (1968). BOXES: An experiment in adaptive control. In E. Dale and D. Michie (Eds.), *Machine Intelligence 2*. Oliver and Boyd.

Miller, W. T., F. H. Glanz, and L. G. Kraft (1990). CMAC: an associative neural network alternative to backpropagation. *Proceedings of the IEEE 78*, 1561–1567.

Moore, A. W. (1990). *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge.

Moore, A. W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state spaces. In L. Birnbaum and G. Collins (Eds.), *Machine Learning: Proceedings of the Eighth International*

Workshop. Morgan Kaufman.

Moore, A. W. (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Proceedings of Neural Information Processing Systems Conference 6*. Morgan Kaufman.

Moore, A. W. and C. G. Atkeson (1992). An investigation of memory-based function approximators for learning control. Technical report, MIT AI Lab.

Moore, A. W. and C. G. Atkeson (1994). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning 13*(1), 103–130.

Moore, A. W. and C. G. Atkeson (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning 21*.

Peng, J. and R. J. Williams (1992). Efficient learning and planning within the dyna framework. In *Proceedings of the Second International Conference of Simulation of Adaptive Behavior*.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning 3*, 9–44.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing*

Systems, Volume 8, Cambridge, MA, pp. 1038–1044. MIT Press.

Sutton, R. S. and A. G. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Sutton, R. S., D. McAllester, S. Singh, and Y. Mansour (1999). Policy gradient methods for reinforcement learning with function approximation. Submitted to NIPS 12.

Thrun, S. and A. Schwartz (1993). Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ. Lawrence Erlbaum.

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

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, University of Cambridge.

Watkins, C. J. C. H. and P. Dayan (1992). Technical note: Q-Learning. *Machine Learning 8*(3/4), 279–292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 8*, 229–256.