Local Planning for Continuous Markov Decision Processes

By Ari Weinstein

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Michael L. Littman

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2014

ABSTRACT OF THE DISSERTATION

Local Planning for Continuous Markov Decision Processes

by Ari Weinstein

Dissertation Director: Michael L. Littman

In this dissertation, algorithms that create plans to maximize a numeric reward over time are discussed. A general formulation of this problem is in terms of reinforcement learning (RL), which has traditionally been restricted to small discrete domains. Here, we are concerned instead with domains that violate this assumption, as we assume domains are both continuous and high dimensional. Problems of swimming, riding a bicycle, and walking are concrete examples of domains satisfying these assumptions, and simulations of these problems are tackled here. To perform planning in continuous domains, it has become common practice to use discrete planners after uniformly discretizing dimensions of the problem, leading to an exponential growth in problem size as dimension increases. Furthermore, traditional methods develop a policy for the entire domain simultaneously, but have at best polynomial planning costs in the size of the problem, which (as mentioned) grows exponentially with

respect to dimension when uniform discretization is performed. To sidestep this problem, I propose a twofold approach of: using algorithms designed to function natively in continuous domains, and performing planning locally. By developing planners that function natively in continuous domains, difficult decisions related to how coarsely to discretize the problem are avoided, which allows for more flexible and efficient algorithms that more efficiently allocate and use samples of transitions and rewards. By focusing on local planning algorithms, it is possible to somewhat sidestep the curse of dimensionality, as planning costs are dependent on planning horizon as opposed to domain size. The properties of some local continuous planners are discussed from a theoretical perspective. Empirically, the superiority of continuous planners is demonstrated with respect to their discrete counterparts. Both theoretically and empirically, it is shown that algorithms designed to operate natively in continuous domains are simpler to use while providing higher quality results, more efficiently.

Preface

Portions of this dissertation are based on work previously published by the author in Weinstein, Mansley, and Littman (2010), Mansley, Weinstein, and Littman (2011), Weinstein and Littman (2012), Goschin, Weinstein, Littman, and Chastain (2012), Weinstein and Littman (2013), and Goschin, Littman, and Weinstein (2013).

Acknowledgements

My life has been filled with unreasonable people. Don't get me wrong, I mean that in the best possible way. Unreasonable people made me who I am and enabled the events that brought me to graduate school and the completion of my dissertation. Without doubt, the majority of the credit for this work goes to those mentioned in this section, and not myself. Some people say that as a disclaimer, but I know for me, it is true. I feel the most pressure writing this because from my own perspective, acknowledging people here is much more important than the remainder of this document (sorry Michael).

My mother Lynda has always been unreasonably patient. As an educator, her primary concern was always my success in school. While she never pressured me to get perfect grades, I'm afraid that I probably caused her a fair amount of grey hair just making sure I wasn't goofing off and getting in to trouble in class as a kid. She was unreasonable in the enormous amount of time spent taking my brother and I to the best camps a kid like me could hope for at the Brooklyn Aquarium, Bronx Zoo, Cold Spring Harbor Laboratory, and Cradle of Aviation (if you are not familiar with New York's geography and traffic, saying these places are not close is an understatement). She was also unreasonable in her insistence that the home be filled with the latest technology, showing real forethought at a time when computers were both a small fortune, and not really useful for anything. Her forethought was responsible

for the kindling of my interest in computers. My father, Leslie, is an unreasonable man. As a physician and science-lover, he eschewed reading me fairy tales (as far as I recall), for wonderful aged tomes on natural science; I have the best volume in my bedroom to this day. Instead of playing baseball he took my brother and I on hikes up the Hudson River, where he would explain the nature along the way. The fact that I love science is due solely to him. My brother, Steve, was unreasonably perfectionist in school and almost all other endeavors (sometimes doing things the hard way just by principle). He taught me by example what it meant to work hard and take pride in labor. In this way, he instilled in me the work ethic that I carry today. There are just some things you can only learn from an older brother, and there are none like him.

Next chronologically, although certainly of equal significance is my new family. My wife, Maayan, was unreasonable when she took a chance on me all those years ago – I'm still not sure what she saw in me then. She was incredibly unreasonable in leaving her amazing career, close friends, and loving family in order to join me as a student at Rutgers. Her unreasonable support has truly been something I have a hard time understanding every day, and I work my hardest to make sure that I give a fraction to her of what she has given me. Certainly, she will be mortified to see all this written about her in public, but I am doing it anyway, because some things just don't deserve to be kept secret. My new parents, Tiki and Moshe, were unreasonable in the way they pampered me for months while I composed the majority of the document; the only finger I had to lift were over a keyboard. Along with them, the other members of my new family in Israel: Savta, Raz, Gal, Michal, and Mayim are similarly unreasonable. From the first day I met them they made it clear that I belong, and have given me a wonderful home half a world away. Although

the distance between us is difficult, they have been as supportive as anyone in my pursuits. I can say with confidence that nobody has in-laws better than mine.

People who aren't familiar science may think it is some dispassionate pursuit of truth. That is false, especially in my case. Behind the statements and equations lie deep personal relationships and (if you are lucky) friendship and respect. In my formal schooling, I was lucky enough to be taught by people who are both brilliant as well as unreasonable, amazing people.

Professor Bill Smart was my undergraduate and master's advisor at Washington University. He is another unreasonably patient person I had the good fortune to work with—only in retrospect do I realize how much time of his I wasted! (If you aren't familiar with the lifestyle of a professor, time is the dearest resource they have.) He is responsible for introducing me to both scientific research and reinforcement learning. It is very clear I would not have reached this point without him.

Its difficult to decide how start discussing Professor Michael Littman. First of all, he is simply an excellent human being; in all things (not just as as my advisor) I know him to be unreasonably dedicated, patient, helpful, and understanding. He has done many things for me that I'm not sure I would have agreed to do if I was in his position. He taught me to be a scientist by letting me choose my own interests and questions, while still making sure I never strayed too far from areas that may be fruitful. I'm sure he deserved a student smarter and more industrious than myself, but even with my deficits he still managed to teach an enormous amount of things. I'm not sure its possible to find a better mentor or friend than Michael.

There are so many other people I feel the need to acknowledge. In my youth, Sol Yousha also taught me how much science truly does rule. Whatever small amount of elegance in the prose in this work belongs to Rabbi Jonathan Spira-Savett and Leslie Bazer, who taught me more about writing than I have learned before or since. There was a John Doe who helped me out in a pinch just before the SAT and is probably responsible for me getting into Washington University for my bachelor's and master's studies. At Washington University, Professor Ron Cytron encouraged me to study computer science.

At Rutgers, I studied a great deal more than just computer science. Professors Eileen Kowler and Jacob Feldman were sources of extraordinary support (in all forms) during my graduate education. They reintroduced me to psychology and presented a way of working in that area that finally made sense to me, and their perceptual science group was the warmest and most welcoming I encountered throughout my studies.

Some of the greatest influences during my doctoral studies have been other members of my laboratory, The Rutgers Laboratory for Real-Life Reinforcement Learning. The amount of information Lihong Li, Tom Walsh, Carlos Diuk, Bethany Leffler, Ali Nouri, John Asmuth, Chris Mansley, Michael Wunder, Sergiu Goschin, and Monica Babes-Vroman have taught me is truly staggering; time spent with them at the whiteboard in the lab was some of the most memorable of my doctoral studies. I would be remiss if I did not mention my other colleagues with whom I have published: Erick Chastain, Peter Pantelis, Kevin Sanik, Steven Cholewiak, Gaurav Kharkwal, Chia-Chien Wu, and Tim Gerstner. Finally, I would also like to thank Professors Kostas Bekris and Alex

Borgida for their wise and kind remarks. This document has been greatly improved by their insights.

Dedication

For Brian.

Table of Contents

List of Figures

Chapter 1
Introduction

> The new ant put down the cadaver vaguely and began dragging the other two in various directions. It did not seem to know where to put them. Or rather, it knew that a certain arrangement had to be made, but it could not figure out how to make it. It was like a man with a tea-cup in one hand and a sandwich in the other, who wants to light a cigarette with a match. But, where the man would invent the idea of putting down the cup and sandwich—before picking up the cigarette and the match—this ant would have put down the sandwich and picked up the match, then it would have been down with the match and up with the cigarette, then down with the cigarette and up with the sandwich, then down with the cup and up with the cigarette, until finally it had put down the sandwich and picked up the match. It was inclined to rely on a series of accidents to achieve its object.
>
> - T.H. White, The Once and Future King

While the use of machines to make *predictions* is now commonplace in many industries and settings throughout the world, the use of machines to make *decisions* is less common. Although at first blush there may not seem a great distinction between the two, but the implications of allowing machines to plan and make decisions for us are far reaching. For example, consider the task of designing a fast walking gait for a legged robot (which will later be discussed

at length). When using machines to make predictions, an engineer would propose a design (how high to lift the leg, how much to bend the knee, and so on) and then request the machine produce a prediction of how well that design functions in a simulation. In turn, the engineer would use this information to propose a revised gait intended to be more effective than the previous design. As such, only using machines to make predictions requires one to labor in indirect ways to find a solution that produces the desired outcome; to design an effective walking gait, the engineer must have some level of expertise in terms of what impact various decisions will have when attempting to further improve the design. On the other hand, allowing machines to make decisions for us allows us to specify exactly what we desire (a fast walking gait) instead of manually providing proposals for, and then checking properties of, designs of a system that may or may not meet that criteria.

One reason the use of machines for predictions is more common than for decision making is that decision making is a more difficult problem. Decision making by its nature requires prediction making, and uses those results to revise the plan, replacing the repetitive task performed by the human just described.

The methods of planning considered here are of a class that perform decision making *locally*, by which it is meant that they only come up for a decision for a particular query configuration of the domain. The other option is to perform planning *globally*, by attempting to find proper choices to make in all possible configurations at the same time. Although this distinction will be discussed at greater length later, local decision making is more robust as risks of divergence, as well as potential inabilities to represent "'good" policies or plans do not exist, and planning costs can be more easily controlled.

In this document, a number of different decision making methods will be explored, making minimal assumptions about characteristics of the environment they operate within. Assumptions made with regard to the setting are:

1. Domains have (potentially high dimensional) continuous valued states and actions.

2. Domains exhibit some form of smoothness (in terms of rewards, dynamics, or value).

3. There may be many local optima with regards to solutions and solution quality, but solutions must be close in value to the globally optimal solution.

4. Expert knowledge of the domain is minimal, and is restricted to a generative model (equivalent to a simulator), that can reset to a given query state.

5. Domains may exhibit stochasticity.

The walking example can be related to these assumptions. One way to define the state would be in terms of the position and velocity of some fixed part of the skeleton, with the remainder of the state being composed of angle and angular velocities of other parts of the body relative to that point. Actions would consist of torques applied at each joint. Even with a highly simplified body structure, the number of dimensions in the state and action spaces are fairly large. Given these state and action spaces, access to a simulator is assumed. This simulator accurately models dynamics of the robot body, which may include stochasticity resulting from noise in the motors, variations in friction of the ground surface, and other possible external forces, such as wind

buffeting. On the other hand, due to the desire for applications with broad applicability (as we will later want to apply the same algorithm to other domains such as swimming and bicycle riding), we eschew planning methods that leverage properties isolated to walking from consideration.

The focus on 1 and 2 differentiates the work here from the vast majority of the RL literature. Continuous domains arise naturally in settings where a physical process is modelled. Among other simulated physical environments, this document will explore planning to swim, ride a bicycle, and walk effectively. By their nature, all of these problems are large both in terms of the number of degrees of freedom in the domain, as well as the number of controls that must be manipulated.

In contrast, the most common assumption made by RL algorithms is that the domain (in terms of the number of discrete states and actions) is of small, finite size. While these algorithms are appropriate for domains that actually do have this property, it has become accepted practice to apply these algorithms to domains that are continuous, by enforcing a coarse discretization of the domain *a priori* and then planning in that discrete space. In high dimensional domains, this form of planning suffers severely from what is called the "curse of dimensionality," meaning that as the number of dimensions in a problem increases, the complexity of the solution grows exponentially in the number of dimensions (Bellman, 1957).

The fact that even canonical domains for testing RL algorithms satisfy assumptions 1 and 2 (violating assumptions of classical RL algorithms) can be taken as evidence that these assumptions, while commonly ignored, are reasonable and important. The most common discrete domains such as *grid world*

(Sutton and Barto, 1998) and *race track* (Barto et al., 1995) are discrete approximations of domains that are truly continuous. Furthermore, domains such as *mountain car*, *acrobot*, and *inverted pendulum* are continuous in terms of states, actions, and dynamics (Sutton and Barto, 1998). Other domains, such as inventory control (Mannor et al., 2003), are of discrete size, but still have meaningful similarity metrics over states and actions that should be leveraged by planning algorithms for optimal efficiency.

An immediate problem is that algorithms using a coarse discretization suffers from poor generalization. That is, it is unable to efficiently apply information that has already been acquired to efficiently plan in the domain in question. The impact of this poor generalization becomes increasingly problematic by increasing costs in two main ways. Firstly, as the smoothness of the domain decreases, discretized resolution must increase linearly throughout the entire domain (whereas more sophisticated algorithms may be able to focus samples only on regions of low smoothness). Secondly, as the size of the domain increases, the number of discrete cells explodes exponentially, resulting in corresponding increases in memory and computational requiements. In the worst case, continuous methods suffer from these factors in the same manner as methods that discretize, but in practice, the impact of increasing problem complexity is less severe.

1.1 Thesis statement

When compared to algorithms that require discretization of continuous state-action Markov Decision Processes, algorithms designed to function natively in continuous spaces have: lower sample complexity, higher quality solutions, and are more robust.

1.2 A Simple Optimization Example

To grant an intuition into the differences between algorithms that function on discretizations of continuous domains, and algorithms that function natively in continuous domains, consider a simple optimization problem. While the exact methods discussed here are not used later in the work, they do reflect fundamental differences of the approaches.

In particular, consider the task of performing optimization over a function $R(a), a \in [0, 5]$. The maximum change of the function within any range (essentially a bound on the derivative), is called the Lipschitz constant, K. In our example, $R(a) = Ka$, and $K = 2$. This simple function illustrates the difference between discrete and continuous methods. The final piece of necessary information is the the tolerated error, ϵ, which will be set to 0.5 in this example.

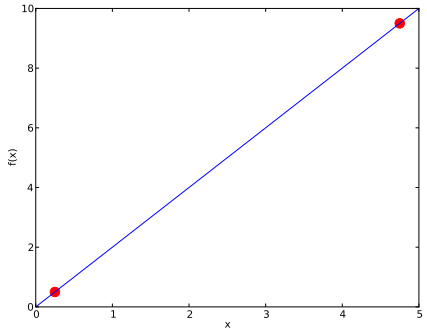
First, let us consider ϵ-optimal optimization by discretization. With this approach, cells must be placed such that there is no point that is further than ϵ/K units away from the center of any cell (assuming sampling is done from cell centers). As a result, the first cell should be centered at ϵ/K, and subsequent centers should be $2\epsilon/K$ units apart. Because information is not generalized outside cells, each cell must be sampled to produce ϵ-accurate predictions of the function. Additionally, the method is not adaptive to the data found during optimization, so, regardless of R, sample placement will always be identical. In our example, 10 cells are needed.

In contrast to optimization through discretization, another approach can follow from the branch-and-bound family of algorithms (Land and Doig, 1960). Samples taken by Lipschitz optimization (sometimes called Shubert's algorithm) vary depending on information obtained from R (Shubert, 1972). To

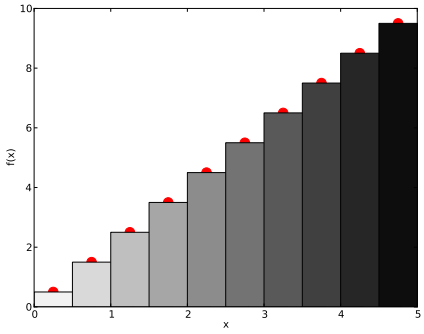
guarantee ϵ-accuracy with Lipschitz optimization, initial samples should be taken ϵ/K units from the minimum and maximum points in the domain. From there, computation is performed adaptively. If $R(4.75) = K(4.75 - 0.25) + R(0.25)$, as is the case in our example, then the maxima of the function must be somewhere between 4.75 and 5.0, and $R(4.75)$ will be ϵ-optimal, so execution terminates after only 2 (instead of 10) samples. Furthermore, as ϵ/K decreases, the number of cells required for naive discretization increases linearly. Lipschitz optimization, however, still completes optimization as long as $R\left(5 - \frac{\epsilon}{K}\right) = K(5 - \epsilon/K - \epsilon/K) + R(\epsilon/K)$ is satisfied. A comparison of the policy used according to discretization and Lipschitz policies are displayed in Figures 1.1(b), and 1.1(a), respectively.

In contrast to the linear growth in sample complexity (the number of samples taken from R) required by discretization with decreasing ϵ/K, the impact of increasing problem size is much more severe. When performing discrete optimization, increasing the dimension of the domain of R causes super-exponential growth in sample complexity. Consider the extension of the above problem to optimization over two dimensions, such that $R(a), a \in ([0,5],[0,5])$. Here, $R = K(a_0 + a_1)$ (K and ϵ remain the same). In this setting, although ϵ/K remains the same, the density of discretized cells must increase, because it it is necessary to consider the case where points lie in the corners of a cell, which are more distant than the sides. The unavoidable explosion in sample complexity makes the method impractical for use in domains with more than a few dimensions. Whereas in 1 dimension, 10 samples were needed when discretizing, in 2 dimensions, the sample complexity explodes to 225. A rendering of this dense sampling is in Figure 1.1(d). This growth continues exponentially as the number of dimensions in the domain increases.

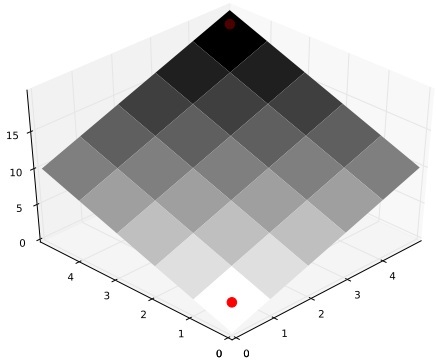
Lipschitz optimization, on the other hand, can still guarantee ϵ-optimal optimization with only 2 samples, even in 2 dimensions. Essentially, samples are taken near the minimal and maximal corners of the domain. As long as the slope between those two points is equal to K, there can be no other place where the function is more than ϵ-larger than the greatest sampled point.



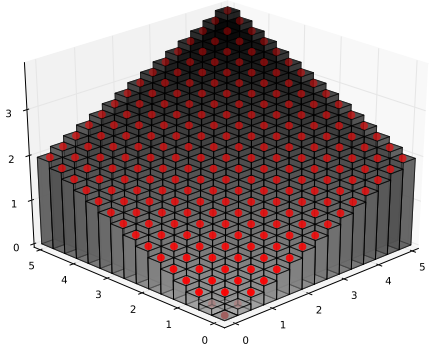
(a) True function, and points sampled by Lipshitz optimization in 1D.



(b) Discretized optimization in 1D.



(c) Open-loop optimization in 2 dimensions.



(d) Discretized optimization in 2D.

Figure 1.1: Comparison of optimization by discretization and Lipschitz bounds in one and two dimensions.

Generally, however, R has a more complex landscape, and Lipschitz optimization must sample more than a pair of points. Even in the worst case, however, Lipschitz optimization has the same sample complexity as uniform discretization, so there is essentially no reason to prefer naive discretization. This can be verified by considering the case where there is only one point $a^* \in A$ such that $R(A^*) > \epsilon$ and $\forall a \in A - a^*, R(a) < \epsilon$. In this case, Lipschitz optimization will be forced to sample as heavily as naive discretization until a^* is found, and then must also verify that no other point is greater than $R(a^*)$.

In practice, K is unknown, and therefore an estimate of it becomes a parameter the the algorithm that must be specified *a priori*. (Optimization when K is unknown is discussed further in Section 6.1.) In the more sophisticated continuous methods discussed here, the algorithm adapts to data, and does not need K to be specified. In discrete methods, selection of K is critical. The system designer must avoid both overdiscretizing (improving the resulting policy, but increasing the computational and data requirements) as well as underdiscretizing (decreasing computational and data requirements at the cost of solution quality). These properties make continuous methods both more sample efficient, as well as more robust. The continuous algorithms proposed here do not require the system designer to attempt this tradeoff, as the approaches either perform an adaptive discretization over the space as the data dictates, while still allowing for near optimal solutions.

1.3 Survey of Planning Approaches

Continuous planning methods have been investigated in a number of fields, each differing based on the assumptions made, with major fields being control

theory, motion planning, and reinforcement learning (although the reinforcement learning algorithms here are actually underpinned by methods from the field of operations research). The most relevant assumptions made are in terms of what domain knowledge exists, characteristics of the dynamics, the existence of noise, and finally the form of rewards or existence of a terminal goal state. As stated in Assumption 4, we consider the case where domain knowledge is minimal, and that in the most relaxed case, smoothness only exists in terms of the value, or quality of actions.

1.3.1 Control Theory

Control theory (Sontag, 1998) makes perhaps the strongest assumptions about knowledge of the domain and the domain itself. Most commonly, dynamics and rewards must be known and in a particular form, violating Assumption 4. One example is the linear quadratic regulator (LQR), which is optimal and can be found in closed form if the domain has linear dynamics with quadratic rewards. These approaches generally tolerate well-behaved noise. The goal is to "stabilize" the domain by performing actions to move the state to the region with the smallest penalty. Model predictive control (MPC) is a field within control theory that is more general than LQRs, but most MPC algorithms still make stronger assumptions than we allow here, and are therefore inapplicable in the setting we consider. In general, significant domain expertise is required so that problems can be formulated in a manner that is solvable by a particular algorithm.

1.3.2 Motion Planning

Motion planning (LaValle, 2006) is concerned with finding a path from a given start state to a goal state. A common assumption is that inverse dynamics are known, so that if a sequence of "nearby" states can found from the start to end states, an appropriate sequence of actions can be discovered to produce that trajectory. Additionally, domains are assumed to be noiseless, and divergence from a plan generally is not tolerated, violating Assumption 5. The subfield of kinodynamic planning introduces constraints on velocities and accelerations in the system, and is most similar to the setting considered here. On the other hand, the assumption of known inverse dynamics and determinism is stronger than the assumptions of this work.

1.3.3 Reinforcement Learning

Reinforcement learning (Kaelbling et al., 1996) problems are the most general of the three discussed in this section. Unlike control theory and motion planning, it has traditionally been concerned with discrete domains, which have arbitrary but non-adversarial noise, rewards, and dynamics. The primary goal of this work is to develop algorithms that satisfy all assumptions by developing RL algorithms that function natively in continuous domains, and outperform algorithms requiring a coarse discretization. While domains with continuous states have been considered in the literature for some time, only a handful of algorithms (Sutton et al., 1999, Kappen, 2005, Lazaric et al., 2007, Van Hasselt and Wiering, 2007, Martín H. and De Lope, 2009, Pazis and Lagoudakis, 2009) function natively with continuous actions, as it introduces a complex optimization step to planning. Most commonly RL algorithms for continuous domains

assume smoothness in transitions and rewards. Here, however, algorithms for new classes of domains are presented that require only weak smoothness in terms of the quality of nearby actions, or even no smoothness at all.

1.4 An Overview of Reinforcement Learning

Because it is the only approach that satisfies assumptions made, this work considers problems from the perspective of RL. In particular, the setting focused on here violates the assumptions made by most algorithms from the fields of control theory and motion planning. Perhaps the earliest significant success of such an approach was in the game of Backgammon, where the algorithm TD-Gammon achieved a level of play competitive with the best human backgammon players of the time, and was so effective that it introduced new techniques in the top-tier of human play (Tesauro, 1995). Another significant success was in the game of Go, which is significantly more complex than chess and was once considered so challenging that computers would never be able to play competitively (Gelly and Silver, 2008). On the other hand, the general reliance on coarse discretization has prevented the use of such approaches on large domains that are naturally continuous. The goal of this work is to introduce algorithms that allow for highly complex problems to be solved, which we argue and demonstrate is not possible with classical RL methods based on coarse discretization.

Chapter 2
Planning in Discrete Domains

In this chapter, reinforcement learning algorithms that assume discrete domains are discussed. The two most commonly discussed discrete settings discussed in the RL literature are the k-armed bandit, and the Markov decision process (MDP). In the k-armed bandit, the agent is repeatedly presented with a choice between a small set of k arms (actions), each of which has an associated reward distribution. The goal of the agent is to maximize expected reward over time through interaction with the domain. Although the problem is quite simple, advances in this area have lended themselves to the development of algorithms for a more complex problem, planning in Markov decision processes (MDPs). Like bandits, MDPs require action selection, but also involve transitions between *state* depending on action selection. This additional factor requires algorithms to plan explicitly for what may occur in the future.

2.1 Discrete Bandits

As is true with the origin of the study of probability by Pascal and Fermat, gambling can be a good motivator for important mathematical development. The k-armed bandit (Robbins, 1952) models the setting of a gambler who must choose between k slot machines, and must earn as much money as possible (or, more realistically, to lose it as slowly as possible).

Formally, the agent is presented with a set of arms A, $|A| = k$. At each time step n, the agent is required to select an arm $a_n \in A$. After selecting arm a_n, the agent receives a stochastic reward $r_n \sim R(a_n)$. The goal is to select arms such that r_n is close to the optimal reward $r^* = \arg\max_a \mathbb{E}[R(a)]$, although how this is measured exactly depends on the formalism used.

The two most common formalisms are Probably Approximately Correct (PAC) (Fedorov, 1972, Valiant, 1984), and regret. PAC is concerned with guarantees in the form of: with probability at least $1 - \delta$, the final result will be ϵ close to correct. When applied to bandits, this property means that after some precomputed amount of time N, a PAC algorithm must return an arm $\hat{a}^*$, such that $\mathbb{E}[R(\hat{a}^*)] \geq \mathbb{E}[R(a^*)] - \epsilon$, and the arm selected must satisfy that bound with probability $1 - \delta$. An algorithm is called *efficiently* PAC if it can do so in time, samples, and memory polynomial in the size of the domain, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$. While finding PAC algorithms for many hypothesis classes is interesting, in bandits PAC solutions are fairly trivial. For example, simply pulling each arm $O(K/\epsilon^2 log(1/\delta))$ times and then choosing one with the highest mean is efficiently PAC (Even-dar et al., 2002).

In bandits, a more frequently considered goal is to find an algorithm that minimizes cumulative regret, which at time N is defined as $\sum_{n=0}^{N} (r_n - R(a^*))$. If regret is sublinear in N, it means the algorithm converges to the optimal arm. It has been shown that for most interesting reward distributions R (Gaussian, Bernoulli, Poisson), regret can be no better than $\Omega(log(N))$ (Lai and Robbins, 1985). Shortly, a simple, but optimal, regret algorithm for bandits will be discussed. Another possible performance metric is simple regret, which analyzes the suboptimality of the final decision after planning, as opposed to the suboptimality of all decisions made during planning (Bubeck et al., 2011a).

A central problem in RL is balancing exploration and exploitation. In non-trivial cases, the agent must learn about the domain through exploration, which involves decisions that, in hindsight, are poor. Once exploration has been carried out, the agent can exploit learned behavior to optimize its performance. The fundamental difference between PAC and regret-based algorithms can be framed in terms of the way they conduct exploration and exploitation.

In PAC algorithms, all exploration is done initially in an exact finite amount of time, N, and then an arm that is probably near-optimal is selected and exploited forever. Regret algorithms, on the other hand, interleave exploration and exploitation continuously. As a result, regret algorithms must continuously sample all arms (although the rate changes over time) to ensure that sample estimates are not the result of an unlikely set of samples from R. While regret algorithms sample all arms infinitely often in the limit, the optimal arm must be sampled at an increasing rate as N increases to guarantee sublinear regret. This behavior has the benefit of allowing regret-based algorithms to be "anytime" methods, meaning that execution can be terminated at any time, but the longer the algorithm runs the better its solution becomes. PAC algorithms, on the other hand, cannot be interrupted and only produce a result after N samples have been taken.

2.1.1 Upper Confidence Bounds

The problem of developing an algorithm with optimal regret uniformly over time (as opposed to asymptotically) in the k-armed bandit setting was solved with the introduction of the family of upper confidence bound (UCB) algorithms (Auer et al., 2002). We discuss the simplest UCB method, called UCB1, which has regret $\Theta(log(N))$. Aside from very small factors, UCB1 has regret

equal to theoretically optimal performance. Other UCB algorithms slightly improve regret at the cost of algorithmic complexity. We present UCB1 for a number of reasons: it is simple, theoretically nearly optimal, and has come to play a key roll in the field of Monte-Carlo tree search, which will be discussed later in this chapter.

UCB1 has a policy defined as taking the action

$$\left(\underset{a \in A}{\operatorname{argmax}} \hat{R}(a) + \sqrt{2 \ln(n)/n_a} \right),$$

where $\hat{R}(a)$ is the sample mean of $R(a)$ and n_a is the number of times a has been selected. The two terms in UCB1 explicitly tradeoff between exploration and exploitation, which is interleaved over time. The $\hat{R}(a)$ term causes exploitation by giving a higher value to arms with high sample means. The other term, $\sqrt{2 \ln(n)/n_a}$, is an upper bound based on the Chernoff-Hoeffding bound. It is sometimes called the bias term, as it biases exploration to arms that have been sampled less frequently by inflating the arm's upper bound. In particular, the expectation for the number of samples for each arm is $\mathbb{E}[n_a] \leq \left(8/\Delta_a^2\right) \ln n$, where $\Delta_a = \mathbb{E}[R(a^*) - R(a)]$.

A depiction of the policy produced by UCB1 in a two-armed bandit problem is shown in Figure 2.1. In this particular domain, both arms have rewards that are Bernoulli distributions; the arm represented in green (referred to as g) has a payoff probability of 0.9, while the arm represented in blue (referred to as b) has a payoff probability of 0.3. The figure is separated into three subfigures, with the top figure showing $U(a)$, the middle figure showing $\hat{R}(a)$, and the bottom figure showing the actual sequence of action selection, for the first 100 steps of execution.

A few items are worth noting to intuitively understand the functioning of

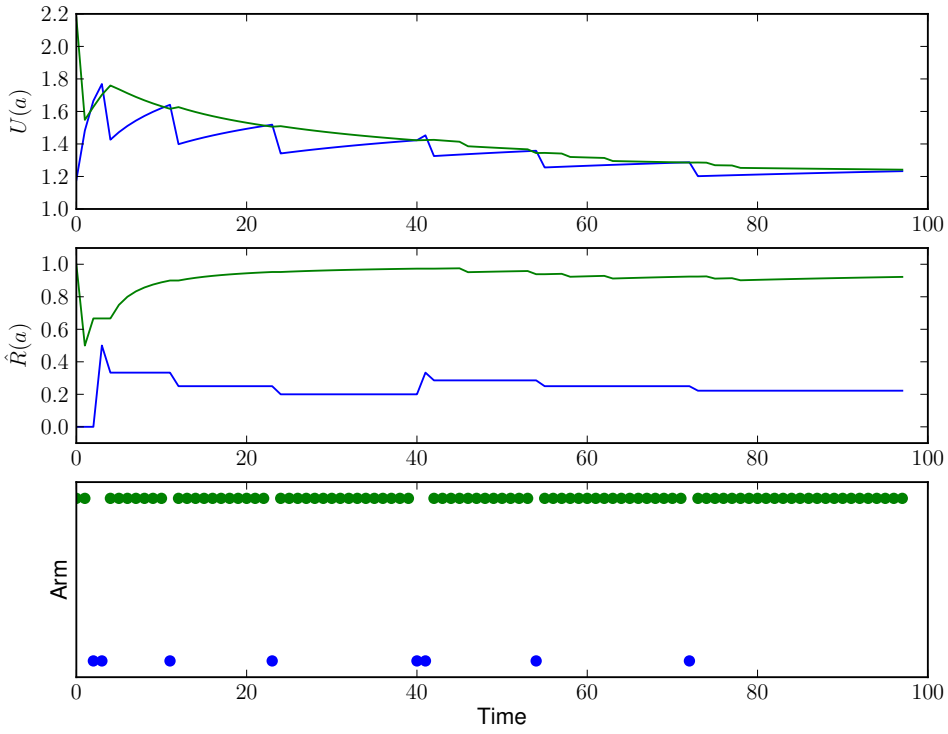


Figure 2.1: Values tracked by UCB in a 2-armed bandit.

the algorithm. In almost every time step, g is selected due to the higher estimate of reward. On the other hand, every time g is selected, $U(g)$ decreases and $U(b)$ increases, due to the way the bounds are computed. Eventually, $U(b) > U(g)$, resulting in b being selected, and a "resetting" of the upper bounds. The rate of change of U in this manner slows with time, however, resulting in less frequent selection of b.

The behavior of UCB1 is discussed at length because the fundamental requirements of trading exploration and expoloitation also applies to other settings, including sequential decision making (discussed later in this chapter) and continuous bandits (discussed in Chapter 3). Although UCB1 is limited

to discrete domains, fundamental aspects of UCB1, such as: anytime behavior, continuous exploration that occurs decreasingly often, and maintenance of mean and bias terms, will be revisited in Chapter 3.

2.2 Discrete Markov Decision Processes

In bandit domains, the agent is required to select from a set of actions, and is only concerned with finding the arm with highest expected reward. In Markov decision processes, there is an additional component involved in decision making, which is called *state*. At any point in time, the agent is situated in some particular state, and must choose between actions that both give a high reward now, and lead to future states that also produces high reward—exactly how this problem is defined will be discussed shortly. In this chapter, we consider discrete MDPs, meaning that the size of the state and action spaces are finite, and it is assumed there is no meaningful similarity metric over states or actions.

As an example, consider the task of selecting a route to take from home to work during rush hour, with the goal of finding the fastest route. In this domain, the state is the current block the car is travelling on, the reward is -1 unit for every minute on a block, and actions determine what choice of direction to take at intersections. The problem involves stochasticity, because the congestion on each road varies naturally from day to day. Additionally, entire routes may be closed due to accidents or road construction, and detours may force the driver onto unexpected paths.

MDPs are a very general formalism for defining a task that requires decision making (a bandit can be described as an MDP with 1 state). The basic

requirement of MDPs is that state transitions and rewards satisfy the Markov property. Informally, this constraint means the state fully describes all features of the domain necessary for optimal decision making. Aside from navigation, some other domains that can be modelled as discrete MDPs are board or card games against a fixed (although potentially stochastic) opponent, or selecting a set of courses to satisfy degree requirements in university (Guerin and Goldsmith, 2011).

2.2.1 Model

An MDP M is a 5-tuple $\langle S, A, R, T, \gamma \rangle$ where:

- S is the set of states.
- A is the set of actions.
- $R(s, a) \rightarrow \mathbb{R}$ is the reward function for taking $a \in A$ from $s \in S$.
- $T(s'|s, a) \rightarrow [0, 1]$ is the distribution over next states $s' \in S$ when taking a from s.
- γ is the discount factor, which controls prioritization of immediate versus future rewards.

Additionally, some algorithms require knowledge of minimum and maximum reward, R_{min} and R_{max}, from which bounds on the value funciton can be derived, with $V_{min} = R_{min}/(1-\gamma)$ and $V_{max} = R_{max}/(1-\gamma)$.

Formally, the Markov property means that for any trajectory through states $\forall s_1, ..., s_t \in S$ achieved by applying actions $a_1, ..., a_t \in A$, the probability of any

transition and reward occurring is conditioned only on the current state and action, or $T(s_t, a_t) = T(s_1, ..., s_t, a_1, ..., a_t)$ and $R(s_t, a_t) = R(s_1, ..., s_t, a_1, ..., a_t)$. Reinforcement learning in MDPs is concerned with finding a good policy $\pi(s) \rightarrow a$ for M. Because of the Markov property, the policy only needs to be conditioned on s, and does not require any information as to what occurred earlier (often called the *history*). Given M and π, the value function is defined recursively as:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) V^{\pi}(s').$$

Equivalently, V^{π} is equal to the expected sum of discounted rewards from s_0 under π: $\mathbb{E}\left[\sum_{h=0}^{\infty} \gamma^h R(s_h, \pi(s_h))\right]$, where h denotes the number of steps in the future where state s_h is encountered during the trajectory through the MDP. Another value of interest is the action-value function

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s'} T(s'|s, a) V^{\pi}(s').$$

$Q^{\pi}(s, a)$ is equivalent to $V^{\pi}(s)$, except a is executed instead of $\pi(s)$ as the first decision. After the initial application of a, π is followed for the remainder of time. An additional relationship is that $V^{\pi}(s) = Q^{\pi}(s, \pi(s))$. Together, V and Q are called the Bellman equations (Bellman, 1957).

For every M, there exists some optimal policy π^* that produces the optimal action-value function, Q^*, such that $\forall \pi, s, a, Q^*(s, a) \geq Q^{\pi}(s, a)$. Alternately, given Q^*, $\pi^*(s) = \text{argmax}_a Q^*(s, a)$.

In this work, most algorithms attempt to optimize the finite-horizon *return*, $\mathbb{E}\left[\sum_{h=0}^{H} \gamma^n R(s_h, \pi(s_h))\right]$, instead of the infinite-horizon value function. Due to discounting, however, the differences in value of optimal finite and infinite horizon policies can be very small, and it is possible to bound the difference between the two. To develop a finite-horizon value function that is less than

ϵ different from the infinite-horizon value, it is sufficient that $H = \log_\gamma \epsilon(1-\gamma)/R_{max}$ (Kearns et al., 1999).

2.2.2 Global Planning

In global planning, the objective is to find a policy that covers the entire state space of the MDP $\pi : S \to A$. The easiest setting for planning is when M is already known. In this case, π^* can be found by a number of methods (Bellman, 1957). In general, finding optimal policies for both finite and infinite-horizon problems have polynomial cost in $S \times A$ and H. Furthermore, the problem is P-complete, meaning that an efficient parallel solution of finding global policies is unlikely (Papadimitriou and Tsitsiklis, 1987, Littman et al., 1995, Boutilier et al., 1999). One example of an algorithm that has such computational costs is value iteration (VI), presented in Algorithm 1, which is simply the Bellman equation turned into an update rule (Bellman, 1957).

Algorithm 1 Value Iteration

```
1: function VALUE ITERATION(M, ε)
2:     ∀_{s∈S,a∈A} Q̂(s,a) ← 0
3:     repeat
4:         e ← −∞
5:         for s, a ∈ S × A do
6:             q ← Q̂(s,a)
7:             Q̂(s,a) ← R(s,a) + γ Σ_{s'} T(s'|s,a) max_a Q̂(s',a)
8:             e ← max{e, |q − Q̂(s,a)|}
9:     until e < ε
```

Although the terminating condition of VI can be defined in a number of ways, a common method is based on the size of the largest change of any $\hat{Q}(s,a)$ between the last and current iteration, occurring in the pseudocode on Line 9. Once this change e drops below a predefined ϵ, it is possible to bound

the difference in value between the Q-function produced by VI, $\hat{Q}^*$ and optimal: $\forall_{s,a}|\hat{Q}^*(s,a) - Q^*(s,a)| < 2\epsilon\gamma/(1-\gamma)$ (Williams and Baird, 1994). When using this stopping method, the number of iterations grows polynomially in $1/(1-\gamma)$ (Littman et al., 1995). Aside from VI, many other algorithms may be used for planning. Linear programming, for example, produces an exact optimal policy, and therefore does not have a cost dependent on ϵ. In practice, however, linear programming is more computationally expensive than finding a near-optimal policy through value iteration, and is therefore rarely used.

If M is completely unknown, reinforcement learning algorithms can be used. These algorithms operate by processing samples of $\langle s, a, r, s' \rangle$ from direct interaction with the environment, and are divided into model-based, model-free, and policy search methods (Sutton and Barto, 1998, Kaelbling et al., 1996). Roughly, model-based algorithms attempt to build an estimate of M, $\hat{M}$, and then produce a policy for $\hat{M}$ (VI can be used for such a purpose) that is then used to behave in M. Model-free algorithms build an approximation of the optimal action-value function $\hat{Q}^*$ directly without estimating $\hat{M}$. Finally, policy search methods directly search over policies, without construction of $\hat{M}$ or $\hat{Q}^*$. Algorithms discussed later in Sections 2.5 and 3.6 perform a particular form of policy search called open-loop planning, where sequences of actions (to some horizon H) are searched over to produce a policy each time an action must be selected.

We consider a case in between 1: requiring full knowledge of M (as is required by VI), and 2: learning from only direct from interaction with M (as is assumed in "on-line" or model-based, value-based and policy search approaches). Here, access to an *episodic generative model* (EGM), G is assumed, which is used for planning. Access to G is distinct from knowing M because G

only allows sampling from R, and T, but does not provide any further description of the functions. This setup allows the agent to plan based on samples from G, occurring effectively in simulation, as opposed to learning from real domain samples from M.

An EGM G begins initialized at some start state s_0, and maintains a current state s throughout use; initially $s \leftarrow s_0$. When interacting with an EGM, trajectories always begin at s_0 (either because of its initialization, or because a reset occurred), and then proceed naturally based on the provided sequence of actions. These types of trajectories are also referred to as *rollouts* (Tesauro and Galperin, 1996), and are performed until the agent terminates interaction for that period. The agent is allowed three options when interacting with an EGM. The most important option is making a query based on action a. In this case, the EGM informs the agent of $r = R(s, a)$, and $s' \sim T(s, a)$, and sets $s \leftarrow s'$ in G, effectively adding another step in the trajectory through the EGM. The second option is to reset $s \leftarrow s_0$, starting a new trajectory. The final option is to terminate querying.

The requirement of such a generative model is weaker than a complete description of the MDP needed by some planning approaches like linear programming, but stronger than the assumption used in on-line RL where information is only obtained by direct interaction with the MDP, as generative models must return samples for any possible reachable $\langle s, a \rangle$ during planning.

In some cases, we will discuss algorithms that assume access to what is called simply a generative model (GM). The distinction is that with such a model, the agent can at any time query for any $\langle s, a \rangle$. That means a reset to s_0 is unnecessary as querying from that state (and all other states) is always permitted. As such, a GM is more powerful than an EGM. Additionally, a

full model (completely describing R and T which we do not assume) is more powerful than a GM and can be used to simulate one.

Assuming access to a GM is stronger than assuming access to an EGM, as there are simulators that do not permit queries from arbitrary states, but do permit queries from a fixed start state. As an example, consider prior research on planning in the video game *Pitfall!* (Goschin, Weinstein, Littman, and Chastain, 2012). In that setting, planning is performed by using an emulator, which functions as G. In this setting, it is impossible to specify arbitrary states from which to plan, because such an operation would require unreasonable knowledge of the internals of the simulator for that particular game. Therefore, the only states that can be planned from are those that are reached directly in a trajectory starting at s_0 during rollouts.

Although the assumption of G may sound strong, there are many cases where it is applicable. Firstly, methods that require generative models can be used whenever a model of the environment is known *a priori*, as is the case in the previous example of *Pitfall!* Indeed, some of the most notable successes of local planning have occurred in the context of board games, where it is assumed both players know the rules before playing. Secondly, it is applicable when G can be built from samples of M (which occurs in model-based RL)(Weinstein and Littman, 2012), which is the approach taken with the application of RL to helicopters (Abbeel et al., 2007). Based on data collected from an actual helicopter, a model of the dynamics is built that allows for planning.

2.3 Local Planning in Discrete Markov Decision Processes

A major distinction exists between two approaches to planning, which we refer to as *global* and *local* planning. In global planning such as carried out by VI, the objective is to find a closed-loop policy that covers the entire state space of the MDP. Global planners with optimality guarantees have planning costs that are polynomial in $S \times A$. When this quantity is large (which we assume is the case), even polynomial costs become prohibitively expensive. In these situations, which is assumed here, local planning methods are preferable. Whereas global planners develop a policy $\pi : S \to A$, local planners develop a policy only for a neighborhood around s_0, called S', so $\pi : S' \subseteq S \to A$.

By doing so, the cost of local planning becomes exponential in the planning horizon H, instead of polynomial in the complete state-action space, somewhat sidestepping the curse of dimensionality in M (Kearns et al., 1999). Although it is generally undesirable to trade polynomial for exponential costs, there are two reasons why local planning is advantageous when working in large domains. Firstly, H can be controlled by the practitioner, setting it to a value that acceptably trades off optimality with a computational and sample budget, which cannot be done with global planning algorithms in terms of the size of the state-action space. Furthermore, in practice, local search algorithms are capable of producing high-quality plans with relatively small amounts of data, so exponential costs in H are only necessary in the worst-case. Secondly, in the setting we consider, M is high dimensional (Assumption 1) meaning that there is already an incurred exponential cost in the dimension of M. These two factors combined mean that by using local planning, it is possible to trade an uncontrollable exponential cost with a controllable one.

To put costs related to M and H in perspective, consider the game of checkers. The total number of reachable positions is approximately 10^{21}, which is exponential in the number of positions on the board, and is roughly the number of cups of water in the Pacific ocean. Completely solving this domain was a task that took 18 years, with an enormous amount of human effort to minimize computation time (Schaeffer et al., 2005, Schaeffer, 2009). Although the number of possible positions in checkers is staggering, in comparison to chess, checkers is very small, as chess is estimated to have approximately 10^{49} reachable states. This number, in turn, is tiny compared to the game of Go, which is estimated to have 10^{170} reachable states (Tromp and Farnebäck, 2006).

Clearly, in complex domains such as these, to do anything at all, it is necessary to aggressively restrict the set of states considered while planning, as even a simple enumeration of all states becomes prohibitively expensive. The issue of enormous state spaces is actually common (although commonly ignored by traditional RL methods), as it arises in any domain that has a factored state representation over a number of features (Walsh et al., 2010). For these reasons, local search methods are state of the art for planning in very complex domains.

2.3.1 History of Local Planning

As this work is focused on local planning, it is worthwhile to discuss the history and major successes of the approach. Due to the ability of local planning to plan in domains with huge state spaces, some of the greatest successes of the approach have been in board games, which have this characteristic. Contributions in this area come from some of the most important computer scientists and mathematicians of the twentieth century.

Two Player Domains

John von Neumann is often credited as the founder of game theory. One of his contributions is perhaps the first local planner, called minimax search. This algorithm exhaustively searches all possible sequences of actions possible by all players from a given state in a zero-sum game. Based on this search, the optimal action and score are computed. Because of this exhaustive search, planning costs are always exponential in the search depth, as the entire search tree is examined without pruning. Although computationally infeasible in games large enough to be of interest, it has formed the core of some of the most important local search algorithms.

Claude Shannon later speculated what aspects would be important in a chess playing program (Shannon, 1950). The work begins "Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance." Indeed, this work introduces and discusses many fundamental aspects of local planning, most of which are motivated by correcting limitations of minimax search. In particular, it discusses the importance of heuristic evaluation functions, which allow an approximate value to be assigned to a particular game state, so that exhaustive search to the end of the game is not required, as each level of search causes costs to grow exponentially. The importance of pruning is stressed, as it is the only way to mitigate the costs exponential in the planning horizon (which is the idea underpinning alpha-beta search, discussed next). Additionally, he discusses the possibility of using learning methods to adjust evaluation functions and learn new policies during play (this component is behind the success of TD-Gammon, discussed later). Finally, he discusses the merit of introducing some

stochasticity in decision making, which is a theme that is revisited in the recent success in computer Go algorithms.

As discussed by Shannon, pruning is a necessity when attempting to perform search even a small number of steps into the future. This basic idea underpins alpha-beta search, which is arguably the most influential algorithm for solving zero-sum games. While the history of the alpha-beta search is unclear, its foundations were laid in the 1950s (Newell and Simon, 1976). Alpha-beta search performs von Neumann's minimax search, but maintains bounds on the value of possible solutions. Because of these bounds, pruning can be conducted, resulting in exponential savings. In best-case settings, this pruning is optimal, and results in costs $O(|A|^{H/2})$, as opposed to $O(|A|^{H})$ (Pearl, 1982). With the use of heuristic functions, this cost can be reduced even further (at the risk of sacrificing optimality). Alpha-beta search with heuristics and other modifications was the key in what is probably the most famous success story of local search: IBM's Deep Blue (Hsu, 1999), which defeated the reigning world chess champion, Gary Kasparov.

At the same time research was being conducted on Chess with Deep Blue, major advances in playing the stochastic game of backgammon occurred, albeit with a very different approach. This algorithm, TD-Gammon (Tesauro, 1995) uses Shannon's proposal of learning an evaluation functions, as opposed to using static programmed rules for estimating the quality of different board positions. TD-Gammon operates by a combination of local search and evaluation functions, examining all possible actions by the agent, and all possible opponent responses, and chooses the action that has the highest expected minimax value according to the evaluation function. The term "rollout" was was popularized in this work, as training and evaluation required rolling dice until the

end of the game was reached in enormous amounts of self-play. This approach created an algorithm that was competitive with the best human backgammon players in the world. Additionally, it developed strategies previously unknown, which subsequently were adopted by the backgammon community (Tesauro, 1995).

A similar approach was taken with IBM's Watson, the *Jeopardy!* game show agent (Ferrucci et al., 2010). Just as in backgammon, the evaluation function learned the probability of success from state. The difference being that in *Jeopardy!* the game state is much more complex and consists (among other things) of how much each player has won so far, and how far the game has progressed. Based on these estimates, and confidence in correctly answering the current question, a wager is decided that is believed to yield the highest probability of winning. Although only part of an extremely complex system, this betting strategy was an important piece of what allowed Watson to defeat the strongest human *Jeopardy!* players in the world.

Single Player Domains

We will now turn focus to single player stochastic domains. Some of the earliest successes of local search in this setting were achieved by model predictive control (MPC), from the field of control theory, which has been in use in industry since the late 1970s (Richalet, 1978). As is the case with other local planners, MPC has been found to be particularly well suited to high dimensional, complex domains. Differential dynamic programming (DDP), a form of MPC, has recently seen a number of notable successes. When coupled with system identification, DDP was used to successfully perform difficult acrobatic helicopter maneuvers continuously and without loss of control, in real-time

(Abbeel et al., 2007, 2010). Another significant application of DDP has been to the task of humanoid locomotion (Tassa and Todorov, 2010, Erez, 2011, Erez et al., 2011), where DDP (along with other methods), was used to control a simulated humanoid with 22 degrees of freedom, almost at real-time in a number of challenging tasks (Tassa et al., 2012).

Following MPC by roughly two decades, local search began to be explored in the RL literature starting in the 1990s. A significant example of progress made at this point is, real-time dynamic programming (RTDP) (Barto et al., 1995), and is one of the earliest approaches that planned based on simulated trajectories (Bertsekas, 2005). RTDP, however, is specially designed for stochastic shortest-path problems, as opposed to MDPs with general reward functions. The method performs local planning, but maintains the results of local planning for improved performance later in execution. Although not a new method, a variant of RTDP has recently seen a resurgence in use and can be considered state of the art (Kolobov et al., 2012).

In a similar manner to the way minimax search uses brute force search to compute the optimal strategy in two player deterministic domains, sparse sampling (Kearns et al., 1999) is a local planner that produces provably near-optimal policies in single player stochastic settings. Like minimax search, the approach is not practical in most real world settings, as it uses nonadaptive depth-first search (with some additions to account for stochasticity). This depth-first search is performed over a tree that is built based on samples from the generative model consisting of all likely reachable states within the planning horizon (how to construct and search over such a tree is a theme that is revisited in other local planners). Essentially, from the start state s_0, the algorithm samples each action repeatedly, and records the rewards and corresponding

resulting states. This process is repeated from all resulting states until a horizon H is reached. After the tree is built exhaustively in this manner, starting from the leaves, average returns for each action, from each state and depth, are computed, and the best estimated return over all actions is returned up the tree. At the end of planning, these values are passed to the root at s_0, and the action is selected that is estimated to produce the largest return.

The significance of the method is that it was the first to produce finite-time guarantees of performance independent of the size of the state space (although the costs are exponential in the horizon). In most interesting problems, however, this exponential cost is prohibitive for even small values of H, leading to poor myopic behavior. This behavior is due to what is essentially a brute force approach, as there is no pruning of the search tree. Although some mention of how to do so does exist in the original publication, it is not a matter that is considered thoroughly, and was left to future work.

Indeed, in the same manner that the practical limitations of minimax search lead to useful optimized algorithms based on the same concept, the fundamental ideas that make up sparse sampling served as the nucleus of later local planners. The class of local planners discussed at length next are motivated by keeping the strengths of sparse sampling (planning costs independent of $|S|$), while improving performance in real world settings.

2.4 Rollout Algorithms

Due to unpruned breadth-first search, sparse sampling has costs that make the algorithm impractical. The approach taken by rollout methods is likewise analogous to a depth-first search according to a policy dictated by the agent's

history. As compared to sparse sampling, rollout planners tend to conduct a less thorough exploration of the search tree, while planing to longer horizons. Rollout planners therefore outperform sparse sampling when it is not critical to consider all possible outcomes, and important to observe events that may occur far in the future. One advantage of rollout algorithms is their generality; they only assume episodic generative models, as opposed to the full generative models required by other local planners. Rollout algorithms are state of the art in many of the largest and most challenging domains. Their success in the game of Go will be discussed in the next section, and they have also seen success in planning in the extremely complex computer strategy game Civilization (Branavan et al., 2012), which only provides an EGM.

With some exceptions, rollout planners follow the same structure, which is described in Algorithm 7. As outlined, the primary iterative loop occurs in the main function PLAN (Line 1), which calls SEARCH during each iteration. In turn, the majority of the work occurs inside SEARCH (Line 6). The first step in SEARCH is determining whether the search horizon has been reached, and if so, the rollout is terminated. In this case, EVALUATE is called to return an estimate of the value of the current state. Bounds on correctness in local planners generally assume all states evaluate to 0, which is the approach taken in this work; proofs still go through with minor modifications as long as EVALUATE returns any value that is boundably incorrect. The most important component of a rollout planner occurs on Line 9, which is the call to SELECTACTION. Based on the current state and depth in the rollout, in SELECTACTION, the planner chooses the next action to execute, which dictates the policy executed during planning. This process repeats as the rollout is recursively executed, so that a

return is produced (Line 12). After the return of the rollout is acquired, the recursion returns and the relevant data is recorded so the policy can be updated during the next rollout (Line 13). Finally, once PLAN terminates planning, the planner returns its estimated best action (Line 5). Whereas in Sections 2.1 and 3.1, N refers to the number of pulls from individual arms, in the sections on rollout planners, N refers to the number of complete rollouts conducted.

Algorithm 2 Generic Rollout Planning

```
1:  function PLAN(G, s_0, H)
2:      repeat
3:          SEARCH(G, s_0, H)
4:      until Terminating Condition
5:      return GREEDY(s_0)
6:  function SEARCH(G, s, h)
7:      if h = 0 then
8:          return EVALUATE(s)
9:      a ← SELECTACTION(s, h)
10:     s' ← G_T(s, a)
11:     r ← G_R(s, a)
12:     q̂ ← r + G_γ SEARCH(G, s', h − 1)
13:     UPDATE(s, h, a, q̂)
14:     return q̂
```

2.4.1 Upper Confidence Bounds Applied to Trees

The motivation behind upper confidence bounds applied to trees (UCT) (Kocsis and Szepesvári, 2006) is to plan in a manner similar to sparse sampling while pruning the search tree heavily. Arguably the most empirically effective rollout algorithm, it casts rollout planning as a sequential bandit problem, basing its policy off that of UCB1 (Section 2.1.1), with the reward from the bandit setting being replaced with the return of the rollout. Originally designed for single player domains, it was later extended to operate in game trees, and a

variant of the algorithm is currently the state of the art approach in computer Go, having achieved master level play in the smaller, but still enormous, 9×9 variant (Gelly and Silver, 2008). Likewise, in recent general planning competitions, algorithms based on UCT have been dominant. The impact of the approach has been so strong that in the 2011 International Probabilistic Planning Competition, all algorithms aside from one were variants of UCT, including the top performer (Kolobov et al., 2012, Coles et al., 2012). The method is presented concretely in Algorithm 3.

Algorithm 3 Upper Confidence Bounds Applied to Trees

1: **function** GREEDY(s)
2: **return** $\text{argmax}_a \hat{Q}(s, H, a)$
3: **function** SELECTACTION(s, h)
4: **return** $\text{argmax}_a \left(\hat{Q}(s, h, a) + \sqrt{\frac{\ln n(s,h)}{n(s,h,a)}} \right)$
5: **function** UPDATE($s, h, a, \hat{q}$)
6: $n(s, h) \leftarrow n(s, h) + 1$
7: $n(s, h, a) \leftarrow n(s, h, a) + 1$
8: $\hat{Q}(s, a, h) \leftarrow \hat{Q}(s, h, a) + \frac{\hat{q} - \hat{Q}(s,h,a)}{n(s,h,a)}$

Because of its algorithmic underpinnings in UCB1, UCT is an anytime algorithm, and is designed to have performance that improves continuously over time. This property is in contrast with the behavior of sparse sampling and other PAC planning algorithms, which compute the number of samples necessary to satisfy conditions provided, but cannot terminate until all computed requirements have been satisfied, and also may be incapable of improving performance after this point is reached.

Extensive pruning performed by UCT is one of the reasons it been practically successful. While this pruning tends to be effective in practice, theoretical results show that UCT can take a super-exponential number of samples (in H) to find an optimal solution due to premature pruning, as it may only explore

optimal regions of search space after super-exponential time. Case studies on fairly simple MDPs, that have the property of having the optimal solution embedded in a region that is otherwise poor in value illustrate concretely when UCT fails in this manner (Coquelin and Munos, 2007, Walsh et al., 2010). These situations are not simply some pathological worst-case construct, as some natural domains with these characteristics have been identified concretely. The failure of UCT to perform well in chess (where alpha-beta variants are still state of the art) is attributed to the existence of such "search traps" in that game (Ramanujan and Selman, 2011). It is worth mentioning that these super-exponential costs are worse than what would occur in sparse sampling or even naive uniform search (effectively the search performed by sparse sampling).

Another issue from a theoretical perspective is that general analysis of the algorithm is very difficult, as estimates of action quality in UCT are nonstationary. This property arises from the use of a bandit algorithm to perform sequential planning, as bounds for each $\langle s, a, h \rangle$ do not account for policy changes that occur outside that node as rollouts occur. While the bounds used in UCB1 are correct in the pure bandit setting, the way upper bounds and policies are computed means the bounds no longer hold. In terms of general analysis, there are claims made in the orignial publication (Kocsis and Szepesvári, 2006), but in light of the aforementioned case studies, the only real conclusion that can be drawn with confidence is that the algorithm converges to optimal behavior in the limit, as general performance guarantees do not exist and case studies demonstrate doubly-exponential time in H for convergence to optimal results in the worst case.

2.4.2 Forward Search Sparse Sampling

In response to the limitations of sparse sampling and UCT, forward-search sparse sampling algorithm (FSSS) (Walsh et al., 2010) was proposed. Unlike sparse sampling, FSSS performs best-first, as opposed to breadth-first, search. This modification allows for exponential savings in computation when pruning is performed. Planning is executed in this manner until a PAC solution is obtained. Unlike UCT, the bounds maintained by FSSS are ϵ-accurate with probability $1 - \delta$, so it will not prune optimal subtrees. Also, unlike UCT, it is guaranteed to visit each leaf at most once, so it can take at most an exponential number of samples in H to produce optimal policies. A generalization of FSSS has also been produced that extends the algorithm to two zero sum games, and is guaranteed to explore a subtree of the game tree as compared to what alpha-beta expands (Weinstein et al., 2012).

The version of FSSS presented in Algorithm 4 is modified from the original presentation, and this updated algorithm will be referred to as FSSS-EGM, as it has been updated to function with episodic generative models. In a slight modification of the standard rollout structure (Algorithm 7), the update function on Line 4 takes an additional argument $r = R(s, a)$, and $s' \sim T(s, a)$.

A number of variables must be described. L and U hold the lower and upper bounds on $Q(s, h, a)$, respectively. Initially, values in L are V_{min} and U are V_{max}. Rollouts begin from the root and proceed until a leaf is reached. As originally presented, these rollouts are conducted until L and U meet at the root, but in practice rollouts are performed until a budget of samples or time is reached and then the best action according to L is taken.

From a theoretical perspective, C should be computed as a function of ϵ and δ, but in practice this is simply treated as a parameter set to some small

Algorithm 4 Forward-Search Sparse Sampling for Episodic Generative Models

function GREEDY(s)
  return $\text{argmax}_a \, L(s, H, a)$
function SELECTACTION(s, h)
  return $\text{argmax}_a \, U(s, h, a)$
function UPDATE($s, h, a, r, s', \hat{q}$)
  $\hat{T}(s, a) \cup s'$
  $U(s, h, a) \leftarrow r + \gamma$ BOUND(s, h, a, U, V_{max})
  $U(s, h) \leftarrow \text{argmax}_a \, U(s, h, a)$
  $L(s, h, a) \leftarrow r + \gamma$ BOUND(s, h, a, L, V_{min})
  $L(s, h) \leftarrow \text{argmax}_a \, Q(s, h, a)$
function BOUND(s, h, a, B, V)
  $\mu_1 \leftarrow \mathbb{E}_{s' \in \hat{T}(s,a)} \left[B(s', h-1) \right]$
  if $|\hat{T}(s, a)| \geq C$ **then**
    return μ_1
  else
    $\mu_2 \leftarrow V$
    return $\left(|\hat{T}(s, a)| \mu_1 + (C - |\hat{T}(s, a)|) \mu_2 \right) / C$

constant value to speed planning. After each rollout, information from the expanded leaf is propagated up in the following manner: when a leaf is expanded, its upper and lower bounds are set to its reward. From there, $L(s, a, h)$ and $U(s, a, h)$ are updated based on the weighted averages of corresponding estimates over all observed children. Then, $L(s, h)$ and $U(s, h)$ are set to the maximal corresponding values over all actions for that $\langle s, h \rangle$. This process continues up the tree until the bounds at the root have been updated, at which point a new rollout begins.

In the original formulation, FSSS takes C samples from $\forall a \in A, T(s^{\#}, a)$ at any point that a new state $s^{\#}$ is encountered, making it unusable with an EGM. FSSS-EGM computes bounds in a manner such that this resampling is unnecessary, allowing it to be used in an EGM. Another difference is that in FSSS-EGM $\hat{T}$ is a multiset as opposed to the standard set of next states used

in FSSS. The distinction is important, because a multiset allows for value estimations to be preformed based on weighted averages (Line 12), which cannot be done in FSSS. Additionally, the version presented here is potentially much more sample efficient due to the fact that estimates of T (unlike that of U, and L) are independent of search depth, and can therefore be done globally (Line 6), and may not have to be taken C times due to the modifications to make the algorithm compatible with an EGM.

2.4.3 Limitations of Closed-Loop Planning

Although closed-loop local planning methods produce state of the art results in a number of challenging domains, there are a number of limitations of the approach that are worth discussing. Closed-loop methods build statistics (and commonly data structures) based around samples of $\langle s, h, a, \hat{q} \rangle$, which can incur large costs, especially if new states are encountered frequently. Although in some cases (especially small deterministic domains), $\langle s, h \rangle$ are revisited often enough such that such effort can be put to good use, in stochastic domains with large state spaces, states may be revisited infrequently given a limited number of trajectories, meaning that statistics maintained do not help decision making.

Essentially, almost all closed-loop planners must revisit an $\langle s, h \rangle > |A|$ times in order for such effort to actually be useful, as prior to that point algorithms perform action selection by chance (with even more revisiting necessary in the case of stochastic domains). As an illustration, Figure 2.3(a) demonstrates how increasing problem size leads to increasing rates of chance action selection as a function of problem size and search depth. In the figure, the x-axis represents the search depth in a rollout, and the y-axis represents the observed probability of action selection by chance, because a visit to a particular

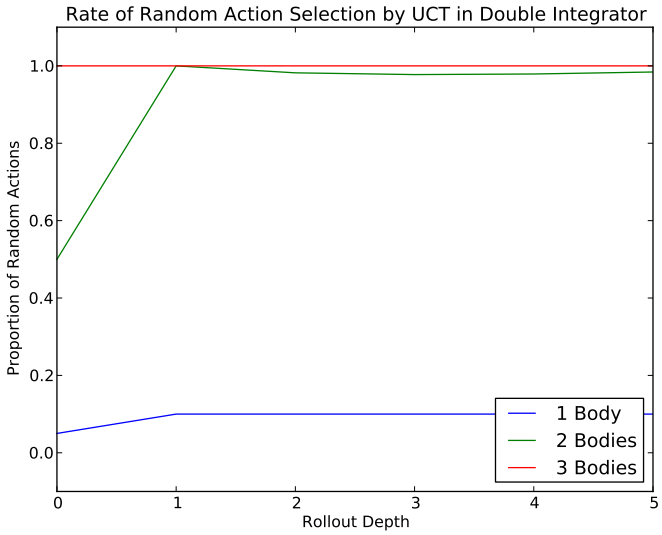
$\langle s, h \rangle$ encountered in the rollout occurred less than $|A|$ times. Curves are rendered in blue, green, and red, corresponding to increasing problem complexity by controlling 1, 2, or 3 instances of a domain simultaneously (for a complete description see Section 4.3.2). Particulars of the domains and algorithm for the purposes of illustration are not important, as the property being displayed occurs with essentially all discrete closed-loop planners as problem size increases while the number of available trajectories are held constant.

A consequence of this phenomenon is that producing a good policy becomes very difficult. Firstly, estimating $\hat{Q}(s, h, a)$ comes to require many samples, as rollouts devolve into random walks in the domain, producing returns of high variance. In such a situation, it becomes very difficult to select a good action, as a difficult signal-to-noise problem arises. In the experimental setting used to create the illustration, random action selections are responsible for approximately 95% of the return meaning that the initial action (which is what the planner ultimately cares about) only has a weak influence on returns. Secondly, $\hat{Q}(s, h, a)$ only comes to estimate the action value according to a random policy, which can be very different from $Q^{\star}(s, h, a)$, leading to suboptimal policies being developed.

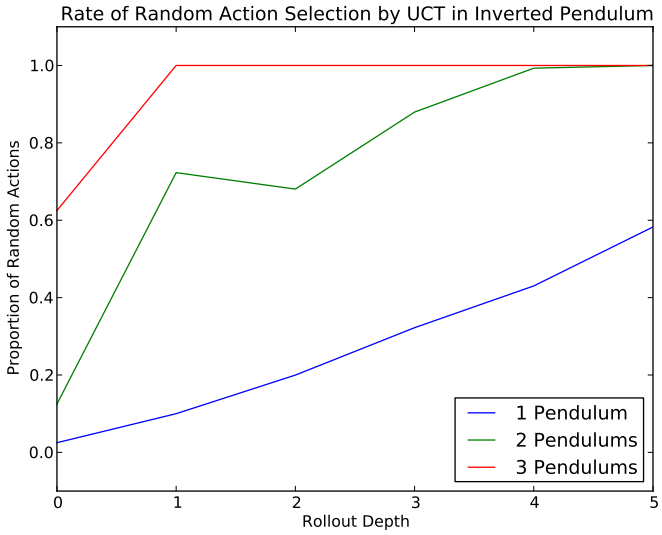
2.5 Open-Loop Planning

While closed-loop planners perform action selection conditioned on $\langle s, h \rangle$, open-loop planners only do so based on h, planning over sequences of actions irrespective of state. Therefore, instead of mapping states to actions with $\pi : S \rightarrow A$, open-loop planners map a step in a rollout to an action, with $\pi : h \in \mathbb{Z}^{+} < H \rightarrow A$. Although this method is a form of policy search, it is

Figure 2.2: Rate of chance action selection by closed-loop planners in increasingly complex domains.



(a) Double integrator.



(b) Inverted pendulum.

different from the most common approach of searching for a global representation of the policy that is incrementally improved at the end of each episode

(Williams, 1992), as here we discuss local planning that is conducted entirely anew at each time step. An advantage of open-loop methods is that ignoring state reduces the size of the hypothesis space, naturally making planning costs independent of S. This change helps resolve the problem of chance action selection (just described) that occurs with closed-loop planners when operating in large domains. Additionally, because open-loop planners form simpler plans, they are applicable in more settings. As long as a reset to s_0 is possible, open-loop planners operate identically in discrete, hybrid, and continuous domains (discussed in Chapter 3), as well as in partially observable Markov decision processes (Littman, 2009); for a concrete example see Section 5.4.

As a more powerful decision making paradigm, however, closed-loop planners are capable of planning effectively in some stochastic domains where open-loop methods are incapable of producing the optimal policy. As an example, consider the MDP in Figure 2.3. In this MDP, there are four different open-loop plans. The solid-solid and solid-dashed sequences have an expected reward of 1, whereas both sequences beginning with the dashed transition produce 0 on average. Thus, the best open-loop plan is solid-solid. A better closed-loop policy exists, however. By first selecting dashed, the agent can observe whether it is in state s_2 or s_3 and choose its next action accordingly to get a reward of 2, regardless.

In spite of this performance gap, open-loop planners considered in this work have two properties that mitigate this issue. First, these planners attempt to maximize the *expected* reward of a given action sequence, so this estimate reflects the fact that a particular sequence of actions can lead to a distribution over returns due to differences in trajectories that arise from the same action sequence. Therefore stochasticity is accounted for. The second property is that

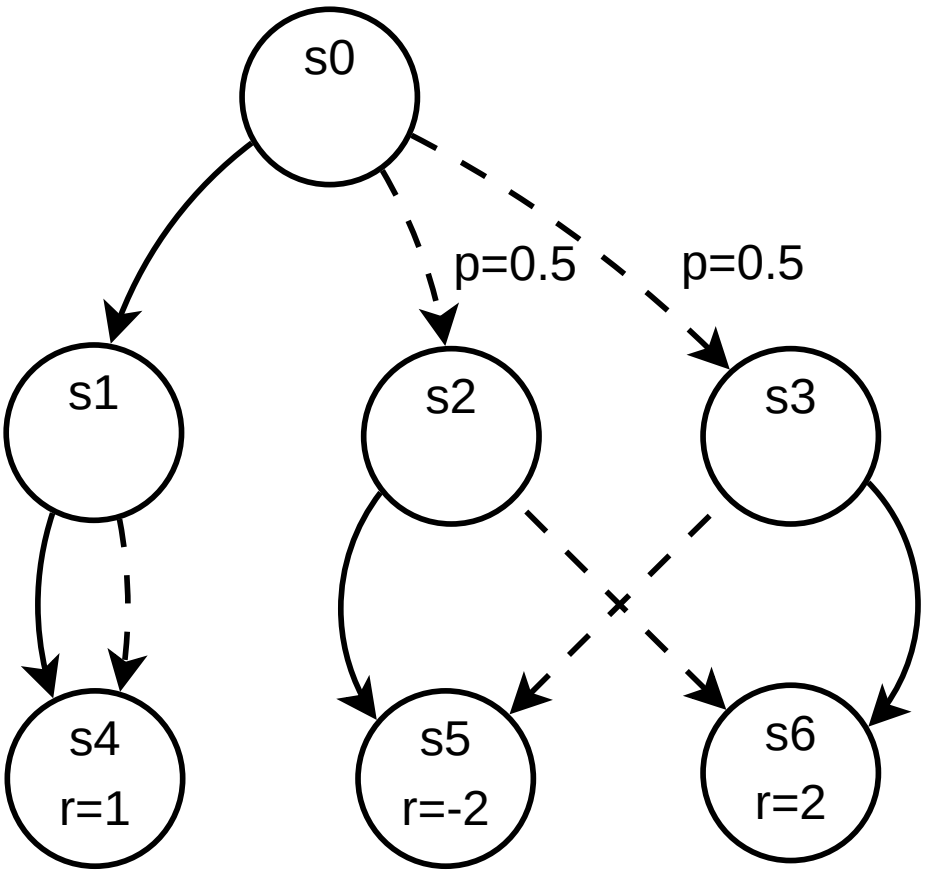


Figure 2.3: An MDP with structure that causes suboptimal behavior when open-loop planning is used.

although planning open-loop, the policy these planners execute is closed-loop; replanning occurs at every step in time from the current state. Therefore, the expected return obtained by these planners as a result of execution in the true domain is guaranteed to be no worse (and can be considerably higher) than the return predicted originally during planning at each step.

These properties of open-loop planners put them in a middle ground between algorithms like global closed-loop planners such as linear programming and FF-Replan (Yoon et al., 2007), which take opposite positions in the way

stochasticity is handled during planning. Linear programming finds an optimal solution for an MDP by computing a policy for each state while fully considering stochasticity, but does so in time polynomial in the size of the MDP. When the MDP is large, however, this method becomes prohibitively expensive, making other planning methods necessary. FF-Replan is a planning algorithm for finite MDPs that removes all stochasticity from an MDP by planning in a modified MDP where all transitions are deterministically set to be the most likely next state. The policy computed by FF-Replan in this modified MDP is followed until the agent encounters a transition that was unexpected, and then planning is started again. Although this method will fail in MDPs with particular structures, it has shown a considerable amount of empirical success, winning a number of planning competitions due to its reduced computation costs (Younes et al., 2005). These results support the claim that only partially reasoning about stochasticity in the manner done by open-loop planners is still capable of producing high-quality results in practice.

2.5.1 Open-Loop Optimistic Planning

The open-loop optimistic planning algorithm (OLOP) (Bubeck and Munos, 2010) constructs a policy by considering sequences of rewards that emerge from action sequences, without considering states encountered during the trajectory. While it is a regret based algorithm, it is different from standard regret based algorithms (such as UCB1) because it optimizes for what is called *simple* regret, which are bounds on error of a recommended action after training, as opposed to *standard* regret, which bounds the suboptimality of all actions

selected during training. As a result, while standard regret algorithms are anytime approaches, simple regret algorithms are not. As such, simple regret algorithms have much in common with PAC methods. The distinction between simple regret and PAC is that whereas PAC algorithms compute a required number of samples N as a function of δ and ϵ, simple regret algorithms are given a budget of samples N and give an expectation on the quality of the reward, which is essentially the opposite operation.

The main idea behind the functioning of OLOP is that the differences in returns of two action sequences can be bounded by the first point along those two action sequences where they diverge, which is possible because of γ and $R_{max} - R_{min}$. For example, if two action sequences of length H only differ in the final action, the difference in their expected returns can be at most $\gamma^H(R_{max} - R_{min})$. On the other hand, if the action sequences have different initial actions, the differences in those sequences can be close to $(R_{max} - R_{min})/(1-\gamma)$, or $V_{max} - V_{min}$.

Somewhat similar to UCB1, OLOP produces policies based on upper bounds, except in OLOP the bound is on the return of an action sequence $\mathbf{a} \in \mathbf{A}$, as opposed to the reward of a bandit arm $a \in A$. For any action sequence $\mathbf{a}$ of length $1 \leq h \leq H$, the algorithm computes the number of times $\mathbf{a}$ has been executed $n_{\mathbf{a}}$, the average observed return of that sequence of actions $\hat{\mu}_{\mathbf{a}}$, as well as an upper bound on that sequence $U_{\mathbf{a}}$. Formally,

$$
\begin{aligned}
n_{\mathbf{a}} &\leftarrow \sum_{n=1}^{N} \mathbb{1}\{\mathbf{a}_{1:h}^n = \mathbf{a}\} \\
\hat{Q}_{\mathbf{a}} &\leftarrow \frac{1}{n_{\mathbf{a}}} \sum_{n=1}^{N} \mathbb{1}\{\mathbf{a}_{1:h}^n = \mathbf{a}\} R_h^n \\
U_{\mathbf{a}} &\leftarrow \sum_{h'=1}^{h} \left(\gamma^{h'} \hat{Q}_{\mathbf{a}_{1:h'}} \gamma^{h'} \sqrt{\frac{2 \log N}{n_{\mathbf{a}_{1:h'}}}} \right) + \frac{\gamma^{h'+1}}{1-\gamma}
\end{aligned}
$$

where R_h^n is the reward received on rollout n at depth h, and $\mathbf{a}_{1:h}^n$ refers to the first h actions on the n^{th} trajectory sampled. Finally, based on these values, the B value of each action sequence $\mathbf{a}$ of length H is defined as the smallest upper bound of all subsequences of $\mathbf{a}$, starting at its first element

$$B_{\mathbf{a}} = \min_{1 \leq h \leq H} U_{\mathbf{a}_{1:h}}$$

At each time step, OLOP selects an action sequence $\mathbf{a} \in \mathbf{A}^H$ that has the highest $B_{\mathbf{a}}$ value, with ties broken arbitrarily. As the minimal value of varying upper bounds, the $B_{\mathbf{a}}$ value encodes a tighter upper bound on the value of action sequences than U. At the end of execution, the algorithm returns the most used first action: $\text{argmax}_{a \in A} n_{\mathbf{a}}$.

We present the algorithm in this form, as opposed to a manner conforming to Algorithm 7, as this algorithm in particular is much simpler to understand when presented in this way. As presented, the algorithm simply specifies what the behavior must satisfy, as opposed to how to construct an algorithm that satisfies this behavior, which can be done with a tree structure that encodes different seqeuences of actions, while recording relevant sample counts, mean estimates, upper bounds, and B values.

Finding a concrete open-loop planning algorithm with optimal simple regret in all cases is still an open problem, although it is known the best achievable simple regret is (Bubeck and Munos, 2010):

$$\begin{cases} \Omega\left(\left(\frac{\log n}{n}\right)^{\frac{log 1/\gamma}{log|\mathbf{A}|}}\right) & \text{if } \gamma\sqrt{|\mathbf{A}|} > 1 \\ \Omega\left(\sqrt{\frac{\log n}{n}}\right) & \text{if } \gamma\sqrt{|\mathbf{A}|} \leq 1 \end{cases}$$

OLOP, on the other hand, achieves a simple regret of

$$\begin{cases} \tilde{O}\left(n^{-\frac{\log 1/\gamma}{\log \kappa'}}\right) & \text{if } \gamma\sqrt{\kappa'} > 1 \\ \tilde{O}\left(n^{-1/2}\right) & \text{if } \gamma\sqrt{\kappa'} \leq 1 \end{cases}$$

Where κ' is related to the proportion of near optimal paths. This regret, while quite good, is not tight with the lower bound, and depending on properties of the domain may be better or worse than a related algorithm UCB-Air (Wang et al., 2008). A distinction between the two algorithms, however, is that UCB-Air is less general as it requires knowledge of κ', while OLOP does not.

2.6 Discussion

This chapter has dealt with planning in discrete domains. The traditional method of developing global policies for discrete domains has costs polynomial in the size of the state-action space. In some domains, however, even polynomial costs (which are generally considered to be efficient) may be prohibitively expensive. Take for example, the complete solution of the game of checkers, which took almost two decades to complete. (Furthermore, as a deterministic domain checkers only requires a linear as opposed to polynomial-time solution.) In some domains considered in this work, even a very coarse discretization can result in domains which have size comparable to checkers, but may contain stochasticity. Because solutions on the order of seconds or minutes as opposed to years is desired, local planning methods must be used, which have planning costs that depend on the H and $|A|$, but not S. While having much smaller planning time than global planners, closed-loop local planners in large domains may still require many trajectories to get sufficient coverage of the local area of the MDP. When the budget of trajectories is severely

limited relative to the size of the domain, open-loop methods can make more effective use of available data. In the following chapter, analogous planning methods for continuous spaces will be discussed, followed by a comparison of discrete and continuous planning algorithms.

Chapter 3
Planning in Continuous Domains

In this chapter, we consider planning algorithms that function natively in continuous domains. Continuous-valued states and actions arise naturally in many domains, especially in those that involve interactions with a physical system. Although there is work in the RL literature that considers continuous spaces, the focus has been on domains with continuous state but *discrete* action spaces (Lagoudakis and Parr, 2003, Ernst et al., 2005, Rexakis and Lagoudakis, 2008). Algorithms that function in continuous action spaces have been examined less thoroughly, primarily because working in working in continuous action spaces is significantly more difficult. In both cases, algorithms must generalize information from one point in the space elsewhere, but planning in continuous action spaces also requires optimization over the action space to plan effectively. As a result, the planning methods described here are based heavily on algorithms from the field of nonconvex continuous optimization.

While the main focus of this chapter is on domains with real-valued states and actions, most algorithms presented here are not strictly limited to that setting, and may also be used in discrete domains where a meaningful distance metric exists. One example is the inventory control problem (Mannor et al., 2003), which has integer-valued states (corresponding to the number of an item in stock). In such a domain, for example, there is essentially no distinction in

between having 99 units of an item as opposed to 100 units. While classical discrete planners would treat both states as completely distinct, continuous state planners are able to generalize intelligently, saving both samples and computation.

The few algorithms designed for use in continuous action spaces can be divided between those that attempt to build a value function based on a function approximator (Lazaric et al., 2007, Van Hasselt and Wiering, 2007, Martín H. and De Lope, 2009), and those that search the policy space directly (Sutton et al., 1999, Kappen, 2005). Unfortunately, the literature devoted to value-function approximation, discussed at length in Section 3.3.1, has many negative results showing divergence, documented from both empirical and theoretical standpoints. Classical policy search methods, discussed in Section 3.3.2, likewise have their own set of limitations. While "safer" than value-function approximation, such methods generally require significant domain expertise to produce high-quality results. The methods espoused here perform policy search, but do so in a manner different from classical methods, and are unique in that they safely yield high quality results without the need for domain expertise or the risk of divergence. When dealing with continuous spaces, we will abuse the notation $|S|$ and $|A|$ to refer to the dimensionality of the state and action spaces, respectively.

3.1 Continuous Bandits

The continuous bandit problem is an adaptation of the K-armed bandit to the setting where arms exist in a continuous space (Agrawal, 1995, Moore and Schneider, 1995). Although algorithms designed to operate in this setting make

differing assumptions, almost all assume some form of smoothness with respect to the reward. A rare exception to this rule is discussed in Section 5.4. Most commonly, the constraint is related to Lipschitz continuity and is something of the form $K|R(a_1) - R(a_2)| < D(a_1, a_2)$, for some constant K and distance metric D.

As discussed in Chapter 1, a common method of planning in continuous spaces is to discretize the space and then use an algorithm intended for discrete spaces on the resulting problem. To demonstrate why this paradigm is misguided, consider the regret this approach produces when applying a discrete bandit algorithm to a continuous bandit problem. In this case, there is some truly optimal arm $a^* \in A$, and then there is some optimal arm among the discretization A', with $\mathbb{E}[R(a^*)] - \max_{a' \in A'} \mathbb{E}[R(a')] = \epsilon_1 > 0$. In this setting, the best possible regret the discrete bandit algorithm could produce over N trials by always pulling $\text{argmax}_{A'}$ would result in regret of $N\epsilon_1$, which is $O(N)$. In contrast, consider the regret that would be produced by the poorest arm, $\mathbb{E}[R(a^*)] - \min_{a' \in A'} \mathbb{E}[R(a')] = \epsilon_2 \geq \epsilon_1$. In this case, the regret is $N\epsilon_2$ which is also $O(N)$. Therefore, from the perspective of regret, acting optimally according to a discretization is indistinguishable asymptotically from the worst behavior possible. As is the case in discrete bandit problems, the goal is to develop algorithms that have regret sublinear in N, which is impossible when interacting according to a discretization.

3.1.1 Hierarchical Optimistic Optimization

The Hierarchical Optimistic Optimization or HOO strategy is a bandit algorithm that assumes the set of arms forms a general topological space with an expected reward that is locally Holder, meaning $|R(a_1) - R(a_2)| < KD(a_1, a_2)^\alpha$

(Bubeck et al., 2008). An important property of HOO is that it is one of the few available algorithms designed to perform global optimization in noisy settings, a property we build on to perform sequential planning in stochastic MDPs. HOO operates by developing a piecewise decomposition of the action space, which is represented as a tree (Figure 3.1). The decomposition is essentially equivalent to a k-d tree (Bentley, 1975) although the purposes of the decomposition are different. When queried for an action to take, the algorithm starts at the root and continues to a leaf by taking a path according to the maximal score between the two children at each step, called the B-value (to be discussed shortly). At a leaf node, an action is sampled from any part of action space that the node represents. The node is then bisected at any location, creating two children. The process is repeated each time HOO is queried for an action selection. A depiction of the tree constructed by HOO in response to a simple continuous bandit problem is rendered in Figure 3.1.

A description of HOO is shown in Algorithm 5, with some functions defined below. A node ν is defined as having a number of related pieces of data, with the root of the tree decomposing the action space denoted by ν_0. Unless ν is a leaf, it has two children $C(\nu) = \{C_1(\nu), C_2(\nu)\}$. All nodes cover a region of the arm space $A(\nu)$, with $A(\nu_0) = A$. For any non-leaf node ν, $A(\nu) = A(C_1(\nu)) + A(C_2(\nu))$, and $A(C_1(\nu)) \cap A(C_2(\nu)) = \emptyset$. The total number of times a path from root to leaf passes through ν during action selection is $n(\nu)$, and the average reward obtained as a result of those paths is $\hat{R}(\nu)$. The upper bound on the reward is $U(\nu) = \hat{R}(\nu) + \sqrt{\frac{2 \ln n}{n(\nu)}} + v_1 \rho^h$ for $v_1 > 0$ and $0 < \rho < 1$, where v_1 and ρ are parameters to the algorithm. If the dissimilarity metric between arms a_1 and a_2 of dimension $|A|$ is defined as $||a_1 - a_2||^\alpha$, setting $v_1 = (\sqrt{|A|}/2)^\alpha$, $\rho = 2^{-\alpha/|A|}$ will yield minimum possible regret. Finally,

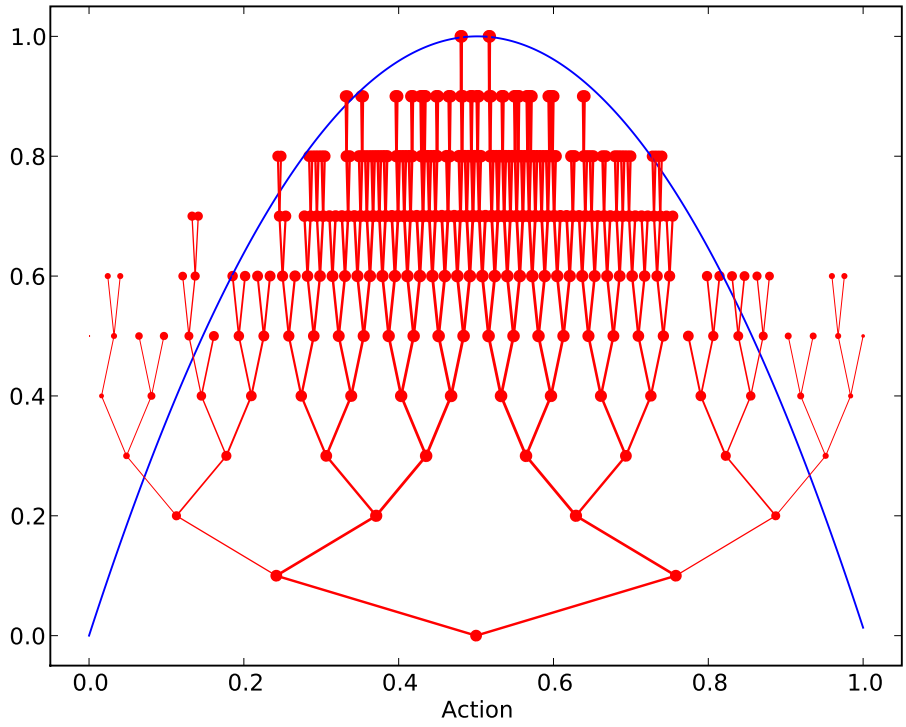


Figure 3.1: Illustration of the the tree built by HOO (red) in response to a particular continuous bandit (mean in blue). Thickness of edges indicates the estimated mean reward for the region each node covers. Note that samples are most dense (indicated by a deeper tree) near the maximum.

n, $\hat{R}$, and U are combined to compute the B-value, defined as

$$B(v) = \min\{U(v), \max\{B(C_1(v)), B(C_2(v))\}\},$$

which is a tighter estimate of the upper bound than U because it is the minimal value of the upper bound on the node itself, and the maximal B-values of its children. Taking the minimum of these two upper bounds produces a tighter bound that is still correct but less overoptimistic. Nodes with $n(v) = 0$ must be leaves and have $U(v) = B(v) = \infty$.

Given the assumption that the domain is locally Holder around the maximum, HOO has regret $O(\sqrt{N})$, which is independent of the dimension of the

Algorithm 5 Hierarchical Optimistic Optimization

function PULL()
  loop
    UPDATE(v_0)
    $a \leftarrow$NEXTACTION()
    $r \sim R(a)$
    INSERT(a, r)
function UPDATE(v)
  $U(v) \leftarrow \hat{R}(v) + \sqrt{\frac{2 \ln n(v_0)}{n(v)}} + v_1 \rho^h$
  $B_1 \leftarrow$ UPDATE($C_1(v)$)
  $B_2 \leftarrow$ UPDATE($C_2(v)$)
  $B(v) \leftarrow \min\{U(v), \max\{B_1, B_2\}\}$
  return $B(v)$
function NEXTACTION()
  $v \leftarrow v_0$
  while v is not a leaf **do**
    $v \leftarrow \text{argmax}_{c \in C(v)} B(c)$
  return $a \in A(v)$
function INSERT(a, r)
  $v \leftarrow v_0$
  while v is not a leaf **do**
    Update $\hat{R}(v), n(v)$
    $v \leftarrow c \in C(v)$ such that $a \in A(c)$
  Update $R(v), n(v)$
  Create children $C(v) = \{C_1(v), C_2(v)\}$

arms and is tight with the lower bound of regret possible. Therefore, based on that performance metric there is no reason to consider any other optimization algorithm as long as assumptions are maintained.

One of the major limitations of the original presentation of the algorithm is that it has planning costs that are $O(N^2)$, as the entire tree must be reevaluated at each point in time to recompute the B-values. Later work presents a number of extensions to the algorithm, one of which reduces the amortized computational complexity to $O(N \log N)$ with only minor changes (Bubeck et al., 2010).

3.1.2 Alternate Continuous Bandit Algorithms

Aside from HOO, there are numerous other continuous bandit algorithms. Perhaps most similar to HOO is the work (Kleinberg, 2004) that performs computation based on what is called the zooming dimension. This method, however, is more complex than HOO and also has poorer theoretical regret.

The Gaussian Process-UCB algorithm (Srinivas et al., 2010) leverages the variance in the Gaussian process function approximator to decide what arm to sample. The goal is to sample from the point where the upper bound on reward is greatest as estimated by the Gaussian process as that point is where the highest reward may lie. The main drawback of this approach is that the algorithm is unable to calculate where these points of interest may be, and another optimization algorithm must be used to find the potentially optimal points, trading one optimization problem for another.

While given different names, regret minimization on continuous bandits and continuous optimization (of stochastic functions) are the same problem cast differently, and perhaps with different metrics. Although we will not discuss such algorithms in detail in this chapter, the field of non-convex optimization is concerned with the essentially the same problem. The main difference, however, is that many algorithms such as cross-entropy optimization, genetic algorithms, and others have very poor or no theoretical guarantees, whereas the bandit literature is primarily interested with providing guarantees on regret bounds. In this chapter, we will build new algorithms from HOO because of its simplicity and near-optimal regret, which can also be used to produce regret bounds for planning algorithms.

3.1.3 Continuous Associative Bandits

In the associative bandit setting (Kaelbling, 1994), state is added to the bandit problem. That is, instead of $R(a)$ defining the reward distribution, $R(s, a)$ does. As with continuous bandits, the common assumption made when working with continuous associative bandits is smoothness of the reward function R, although here smoothness is assumed not only over A, but also S. At each point in time, the algorithm is informed of a state s (the selection of which is outside of the control of the bandit algorithm).

Weighted Upper Confidence Bounds

The simplest associative bandit algorithm we will discuss is designed for discrete action, continuous state domains. Because UCB1 (not designed for associative bandits) only uses sample counts and averages to operate, it is possible to adapt the algorithm by using *weighted* counts and averages, based on a provided distance metric $D(s_1, s_2), s_1 s_2 \in S$. The extension requires samples to record the state of each sample, s_n. Whereas the original rule for UCB1 is $\text{argmax}_{a \in A} \left(\hat{R}(a) + \sqrt{2 \ln(n)/n_a} \right)$, the weighted UCB algorithm (WUCB) is:

$$\begin{aligned} \hat{R}(s, a) &= \frac{\sum_n \mathbb{1}_{a_n = a} r_n D(s, s_n)}{\sum_n \mathbb{1}_{a_n = a} D(s, s_n)} \\ U(s, a) &= \sqrt{\frac{\log \left(\sum_n \mathbb{1}_{a_n = a} D(s, s_n) \right)}{\sum_n \mathbb{1}_{a_n = a} D(s, s_n)}} \\ \pi(s) &= \underset{a \in A}{\text{argmax}} \left(\hat{R}(s, a) + U(s, a) \right). \end{aligned}$$

At a high level, WUCT extends UCT to the associative bandits setting by using an instance-based approach (Atkeson et al., 1997) for generalizing across state. A consequence of this approach is that whereas UCB1 only needs to

maintain 2 numbers for each action during operation (the sample mean and sample count), WUCB needs to record every piece of data encountered during operation, and must perform linear-time calculations based on all samples at each time step. Therefore, the computational complexity of WUCB is $O(N^2)$, which can in some cases be prohibitively expensive, especially in comparison to the $O(N)$ cost of UCB1.

Weighted Hierarchical Optimistic Optimization

While WUCB extends UCB1 to the continuous state, discrete action associative bandit setting by the use of weighted averages and sample counts, in this dissertation, the interest is ultimately in algorithms that function natively in fully continuous state and action spaces. Similar to UCB1, HOO also uses sample averages and sample counts, but is designed for use in continuous action spaces. Therefore, it is possible to apply the same transformation to UCB1 that creates WUCB to HOO, creating the Weighted Hierarchical Optimistic Optimization algorithm (WHOO). Because HOO functions natively in continuous action spaces and operates on a distance metric D that functions in continuous state spaces, WHOO is an associative bandit algorithm designed for fully continuous state and action spaces (Mansley, Weinstein, and Littman, 2011). It is detailed fully in Algorithm 6.

While most of the properties of WHOO extend from the most similar algorithms discussed, HOO and WUCB, the algorithm itself is significantly more complex in terms of algorithmic details. As is the case in WUCB, all samples must be analyzed at each step, leading to computational costs of $O(N^2)$, as opposed to the $O(N \log N)$ cost of standard HOO with optimizations.

Algorithm 6 Weighted Hierarchical Optimistic Optimization

function PULL(s)
  loop
    UPDATE(s, ν_0)
    $a \leftarrow$NEXTACTION(s)
    $r \sim R(s, a)$
    INSERT(s, a, r)
function UPDATE(s, ν)
  $N_\nu(s) \leftarrow \sum_{n=1}^{N} D(s, s_n)$
  $\hat{R}_\nu(n, s) \leftarrow \frac{1}{N_{\nu_0}(s)} \sum_{n=1}^{N} \mathbb{1}_{a_n \in A(\nu)} D(s, s_n) r_n$
  $U_\nu(n, s) \leftarrow \hat{R}_\nu(n, s) + \sqrt{\frac{2 \ln N_{\nu_0}(n,s)}{N_\nu(n,s)}} + v_1 \rho^h$
  $B_1 \leftarrow$ UPDATE($C_1(\nu)$)
  $B_2 \leftarrow$ UPDATE($C_2(\nu)$)
  $B(\nu) \leftarrow \min\{U_\nu(n, s), \max\{B_1, B_2\}\}$
  return $B(\nu)$
function NEXTACTION(s)
  $\nu \leftarrow \nu_0$
  while ν is not a leaf **do**
    $\nu \leftarrow \text{argmax}_{c \in C(\nu)} B(c)$
  return $a \in A(\nu)$
function INSERT(s, a, r)
  $\nu \leftarrow \nu_0$
  while ν is not a leaf **do**
    Store s, a, r as s_N, a_N, r_N
    $\nu \leftarrow c \in C(\nu)$ such that $a \in A(c)$
  Create children $C(\nu) = \{C_1(\nu), C_2(\nu)\}$

3.2 Continuous Markov Decision Processes

We will now discuss planning in MDPs that have continuous state and action spaces. Planning in this class of domains raises distinct problems as compared to classes of domains previously discussed. In contrast to planning in continuous associative bandits, planning in continuous MDPs requires additional temporal consideration of value as opposed to simply immediate reward. As compared to planning in discrete MDPs, planning in continuous MDPs requires

both generalization and optimization to construct a policy.

The addition of all these factors introduces significant challenges to the construction of effective planning algorithms for continuous MDPs. Whereas in discrete MDPs, the equation defining the value of a policy is called the Bellman equation, in continuous MDPs there is the Hamilton Jacobi Bellman (HJB) equation. Although there are some cases where the HJB equation is simple to solve such as domains with piecewise quadratic dynamics (Zamani et al., 2012) in the common case, solving the HJB equation is not feasible. As a result, unlike the simple transformation that allows value iteration (or other algorithms) to be derived from the Bellman equation, it is generally not possible to move from the HJB equation to an algorithm that produces a near-optimal policy.

Because of these difficulties, it has become common practice to simply discretize continuous dimensions, which allows algorithms designed for discrete MDPs to be used. As mentioned, this method is not practical in the setting we consider as domains may be high dimensional (Assumption 1), and the number of cells resulting from this form of discretization is super-exponential in the dimension of the problem. Chapter 4 includes further arguments against this approach.

3.3 Global Planning in Discrete Markov Decision Processes

In this section, we will discuss the two main forms of global planning, which are based on value-function approximation and policy search. The two methods differ in how they develop policies. Value-based methods attempt to find the optimal value function for a domain and then derive a policy from this value function. Policy search methods, on the other hand, forego estimating

the value function and instead search the policy space directly.

3.3.1 Value-Function Approximation

When performing global planning in high dimensional domains, more sophisticated forms of function approximation may be used in place of coarse discretization (which itself is simply a primitive form of function approximator) to estimate the value function. Although the literature on algorithms that perform VFA and function in continuous state *and* action spaces is very limited, there have recently been a number of algorithms proposed for this setting. Some examples are Ex$\langle a \rangle$ (Martín H. and De Lope, 2009), the continuous actor-critic learning algorithm (Van Hasselt and Wiering, 2007), and fitted-Q iteration (Weinstein and Littman, 2012). Unfortunately, all forms of function approximation introduce the risk of failing to produce a near-optimal value function, and therefore, policy. Fundamental risks stemming from the use of function approximators (FAs) can be separated into two categories.

The first category includes issues that arise whenever supervised learning is performed; these problems are not unique to RL. Issues of the bias-variance tradeoff (underfitting and overfitting), overtraining, lack of convergence, and the need to tune parameters depending on the particular problems are fundamental supervised learning issues that naturally also apply when used in RL (Tesauro, 1992).

The other category of risk stems from using an FA as a value function approximator (VFA), and is unique to RL applications. In particular, problems stem from the way errors are compounded during bootstrapping when estimating the value function. To obtain reliable results and worst-case guarantees, one class of algorithms that can safely be used as VFAs is the class of

algorithms called averagers (Gordon, 1995). Examples of averagers are the k-nearest neighbor or decision tree algorithms. Non-averagers, such as artificial neural networks and linear regression, offer no such guarantees, and commonly diverge in practice when used as VFAs (Boyan and Moore, 1995). The difficulty in using averagers as VFAs is that they generally underfit (oversmooth) the value function, leading to very poor policies, and are particularly problematic in domains with many local optima in the value function, as we assume is the case here (Assumption 3).

Even averagers, however, are not entirely safe to use, as VFAs may fail based on many other factors. For example, noise can lead to a systematic overestimation of the value function, causing a degenerate policy to be computed (Thrun and Schwartz, 1993, Ormoneit and Sen, 1999). Even in the case when the resulting policy produced by the VFA is effective, the actual value estimates may be unrelated to the true value function (Boyan and Moore, 1995). Additionally, the representation used in the VFA is another source of difficulty; the wrong set of features can cause failure due to either inexpressiveness (with too few features), or overfitting (too many features) (Kolter and Ng, 2009), which again introduces the requirement for domain expertise to find an appropriate representation of the value function. Yet another complication is that the function approximator must be able to fit many different value functions on the way to fitting Q^*, which requires a great deal of flexibility in the FA.

3.3.2 Policy Search

Policy search algorithms do not build a policy derived from a value function but instead search the policy space directly. These algorithms are safer from those that require VFAs as they do not estimate a value function and do not

risk divergence, but have their own set of limitations.

Policy search methods function by searching for parameters Φ to a function approximator π such that the policy $\pi(s, \Phi) \rightarrow a$ maximizes the return, starting from state s_0. An example of this definition may be used in practice is to have Φ encode weights in an artificial neural network, with s being the values at the input layer.

Because policy search algorithms maximize for return from s_0, their use is restricted to domains that are episodic, meaning trajectories always start from s_0, and only proceed for a finite number of steps. The limitation to episodic domains is one reason why policy search methods are not suitable to the setting considered in this work, although it is a problem that can be addressed, given an episodic generative model.

More significantly, a near-optimal policy must be representable by π—if it is not the case it is impossible for the algorithm to produce effective policies. Finding a good hypothesis class is is generally a difficult task as domains may have sharp boundaries in the state space where policies change and policy representations must be able to fit these boundaries closely (Rexakis and Lagoudakis, 2008). Practically, another requirement is that the complexity of π must be low to allow the search over Φ to be completed in a reasonable amount of time. These two requirements are fundamentally in opposition, because (all other factors held the same) increasing representational richness of a FA requires a more complex function with more parameters. As such, the only way both can be accomplished simultaneously is by levering domain expertise and constructing π carefully (Erez, 2011). Aside from limiting generality, this requirement violates Assumption 4, that domain knowledge is restricted only to access to a black-box generative model.

Another issue that arises when performing policy search is the difficulty in determining how the modification of Φ ultimately alters π. Relatively small changes in Φ may cause large changes in the policy, and as a result action selection may change drastically and even fall outside of the range allowed in the domain. As a result, most approaches only make slight changes to Φ during each iteration of the algorithm. Most commonly, these small changes are made according to an estimate of the gradient of the return with respect to Φ. Such algorithms are appropriately named *policy gradient algorithms*.

Aside from the fact that gradient estimations are unreliable in the presence of noise (Heidrich-Meisner and Igel, 2008, Sehnke et al., 2008), which is present because of the transition distribution, policy gradient methods have more significant limitations. Because they perform gradient ascent, policy gradient algorithms only converge to local optima (Williams, 1992, Sutton et al., 1999). Additionally, when there are large plateaus in Φ-space with regard to return, gradient methods perform a random walk in policy space, leading to a failure to improve the policy (Heidrich-Meisner and Igel, 2008). Both of these limitations mean that when using policy gradient algorithms (or gradient algorithms in general) initializing search in the basin of attraction of the global optimum is critical (Deisenroth and Rasmussen, 2011, Kalakrishnan et al., 2011, Kober and Peters, 2011). The requirement of search initialization near the global optimum is another example of necessary domain expertise that we do not assume is available (Assumption 3), making such algorithms unusable in the setting considered.

3.4 Local Planning in Continuous Markov Decision Processes

Fundamentally, the differences between local and global planning that exist in discrete MDPs (discussed in Section 2.3) also hold in continuous MDPs (with the addition of difficulties involved in performing VFA or policy search). In both discrete and continuous MDPs, global planners must consider all of S when producing a policy. Therefore, in high dimensional continuous domains, the cost of global planning becomes prohibitive, just as it does in high dimensional discrete domains. Instead, producing local policies for regions in the MDP allows for planning problems of tractable size. Additionally, while global planners (ultimately based on VFA or policy search) for continuous MDPs are unusable in the setting we consider due to risks of failure (Assumption 3) or required domain expertise (Assumption 4), the local planners as presented here do not suffer from such issues. The price paid for this flexibility is a replanning cost at each time step. Just as is the case with discrete planners, local planning algorithms can be divided between closed-loop and open-loop policies.

3.5 Closed-Loop Local Planners

In this section, a number of closed-loop planners are presented for a number of different classes of MDPs. These planners are constructed for domains with continuous state, discrete action; discrete state, continuous action; and finally continuous state, continuous action spaces (planners for fully discrete MDPs were described in Chapter 2). These algorithms are all rollout planners, and are based on the structure described in Algorithm 7, which is reprinted here for ease of reading.

Algorithm 7 Generic Rollout Planning

```
1: function PLAN(G, s_0, H)
2:     repeat
3:         SEARCH(G, s_0, H)
4:     until Terminating Condition
5:     return GREEDY(s_0)
6: function SEARCH(G, s, h)
7:     if h = 0 then
8:         return EVALUATE(s)
9:     a ← SELECTACTION(s, h)
10:    s' ← G_T(s, a)
11:    r ← G_R(s, a)
12:    q̂ ← r + G_γ SEARCH(G, s', h − 1)
13:    UPDATE(s, h, a, q̂)
14:    return q̂
```

3.5.1 Hierarchical Optimistic Optimization Applied to Trees

Building on UCT (Section 2.4.1), which takes actions during rollouts according to UCB1, the same approach can be used to create new rollout planners by using other bandit algorithms in place of UCB1 to define policy. In particular, a continuous bandit algorithm such as HOO can be used in place of UCB1, resulting in a planner that operates natively in discrete state, continuous action MDPs. We call this algorithm Hierarchical Optimistic Optimization applied to Trees (HOOT) (Weinstein, Mansley, and Littman, 2010, Mansley, Weinstein, and Littman, 2011). Aside from the modification of replacing one bandit algorithm for another, all other aspects of UCT and HOOT are the same, with the exception of computational costs. Just as the computational cost of HOO is greater than UCB1, at $O(N \log N)$ as opposed to $O(N)$, the computational cost of HOOT is greater than that of UCT (Bubeck et al., 2010). HOOT is described concretely in Algorithm 8; function calls in Algorithm 8 that are not defined in generic rollout planning (Algorithm 7) refer instead to HOO, as defined in

Algorithm 5.

Algorithm 8 Hierarchical Optimistic Optimization applied to Trees

```
1: function GREEDY(s)
2:     ν ← ν_0 of HOO_{s,h}
3:     while ν is not a leaf do
4:         ν ← argmax_{c∈C(ν)} R̂(c)
5:     return a ∈ A(ν)
6: function SELECTACTION(s, h)
7:     HOO_{s,h}.UPDATE(HOO_{s,h}.ν_0)
8:     return HOO_{s,h}.NEXTACTION()
9: function UPDATE(s, h, a, q̂)
10:    HOO_{s,h}.INSERT(a, q̂)
```

3.5.2 Weighted Upper Confidence Bounds Applied to Trees

Weighted Upper Confidence Bounds Applied to Trees (WUCT) is a planner for continuous state, discrete action MDPs. Just as UCT uses UCB1 to perform action selection during rollouts, WUCT uses weighted upper confidence bounds (WUCB) in a similar structure for the same purpose. The primary difference between the structure created by UCT and WUCT is the structure built by the two algorithms to perform planning. UCT attempts to maintain statistics based on each unique state encountered during trajectories through the domain, but when working in continuous state spaces it is necessary to generalize across states, as exact states may never be revisited due to the presence of stochasticity. As a result, WUCT, like WUCB, performs generalization according to a memory-based approach (Moore, 1990). During each step of each rollout, a tuple $d = \langle s, h, a, \hat{q} \rangle$ is recorded in the data set D, which contains the return, $\hat{q}$ associated with a taken from s at depth h in the rollout, which is later used to compare to states reached in the future during planning. WUCT is described

in Algorithm 9.

Algorithm 9 Weighted Upper Confidence Bounds Applied to Trees

function GREEDY(s)
  $\Delta' \leftarrow \langle s', h', a', \hat{q}' \rangle \in \Delta$ such that $h' = 0$
  for $a \in A$ **do**
    $\hat{Q}(s, a, h) \leftarrow \sum_{\Delta'} D(s, s')\hat{q}'$
  $a \leftarrow \text{argmax}_{a \in A}\, \hat{Q}(s, 0, a)$
  return a
function SELECTACTION(s, h)
  $\Delta' \leftarrow \langle s', h', a', \hat{v}' \rangle \in \Delta$ such that $h' = h$
  for $a \in A$ **do**
    $\hat{Q}(s, a, h) \leftarrow \sum_{\Delta'} \mathbb{1}_{a_n = a} D(s, s')\hat{v}'$
    $U(s, h, a) \leftarrow \sqrt{\frac{\log\left(\sum_{\Delta'} \mathbb{1}_{a' = a} D(s, s')\right)}{\sum_{D'} \mathbb{1}_{a' = a} D(s, s')}}$
  $a \leftarrow \text{argmax}_{a \in A}\, \left(\hat{Q}(s, h, a) + U(s, h, a)\right)$
return a
function UPDATE($s, h, a, \hat{q}$)
  $\Delta \leftarrow \Delta \cup \langle s, h, a, \hat{q} \rangle$

A limitation of this approach is that due to the fact that each sample is examined during each rollout, the computational cost of planning is $O(N^2)$, which in practice makes the algorithm computationally too expensive to be of use when large amounts of data are needed in high dimensional domains. Another limitation is that a distance metric D must be provided. Both properties follow directly from the use of WUCB to perform action selection. Finally, as a planning algorithm for discrete action, continuous state MDPs, WUCT is not a planner that functions natively in fully continuous MDPs.

3.5.3 Weighted Hierarchical Optimistic Optimization Applied to Trees

We have discussed two planning algorithms in this section, that are designed for different combinations of discrete and continuous state and action spaces.

HOOT builds a DAG in the same manner as UCT, but replaces UCB1 with HOO, producing a rollout planner that functions in continuous state, discrete action MDPs. Incorporating a memory-based approach and distance metric allows for the associative bandit algorithm WUCB to be used in a similar planning structure, creating WUCT which allows for rollout planning in continuous state, discrete action domains. Combining features of both results in a closed-loop rollout planner that functions natively in continuous state and action spaces (Mansley, Weinstein, and Littman, 2011).

Algorithm 10 Weighted Hierarchical Optimistic Optimization applied to Trees

```
1: function GREEDY(s)
2:     ν ← ν_0 of WHOO_h
3:     while ν is not a leaf do
4:         ν ← argmax_{c∈C(ν)} R̂(c)
5:     return a ∈ A(ν)
6: function SELECTACTION(s, h)
7:     return WHOO_h.NEXTACTION(s)
8: function UPDATE(s, h, a, q̂)
9:     WHOO_h.INSERT(s, a, q̂)
```

This algorithm, which places an associative bandit algorithm WHOO at each depth in the rollout sequence, while maintaining a record of all $\langle s, h, a, \hat{q} \rangle$ tuples observed, results in a closed-loop rollout planner that functions natively in both continuous state and action spaces. While this algorithm has many of the properties that we desire, like WUCT, the memory-based approach ultimately leads to prohibitive computational costs, making the algorithm too computationally expensive to be of practical use. The algorithm, called weighted hierarchical optimistic optimization applied to trees (WHOOT), is outlined in Algorithm 10.

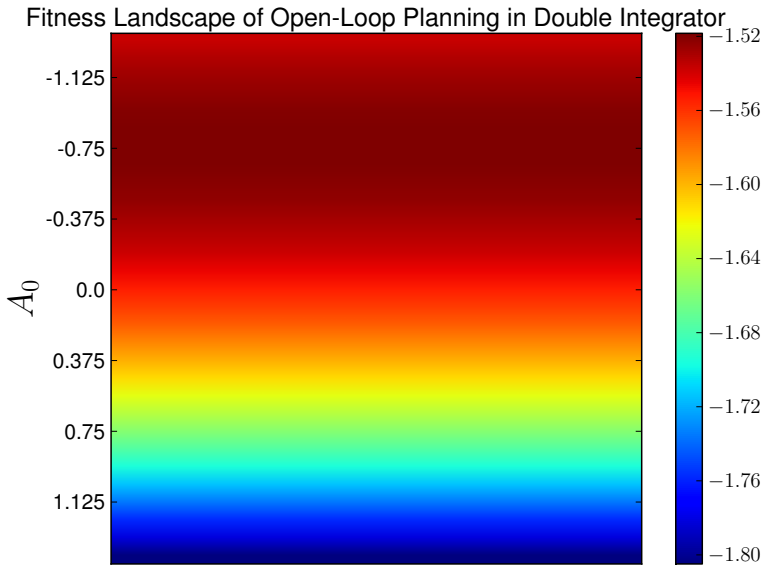
3.6 Open-Loop Planners

While we have presented a number of closed-loop planning algorithms for use in various settings including fully continuous MDPs, the weighted planners for use in continuous state MDPs are computationally too intensive to be of use in practice. This overhead comes from the fact that estimates of action quality depend on state, but since continuous states may never be revisited, comparisons must be made according to a distance metric applied to *all* samples previously observed.

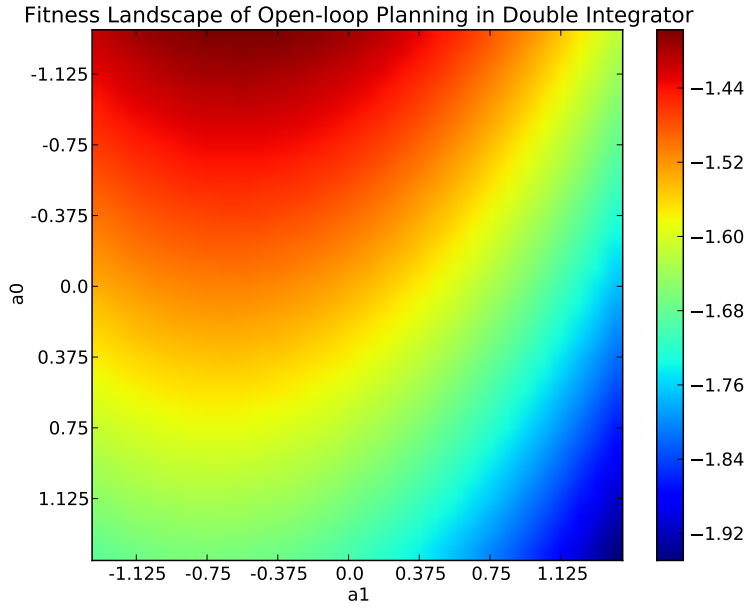
Being that the issue of generalizing across state introduces a significant computational burden, one option is to simply disregard state while planning. As discussed, in Section 2.4.3 this approach can be particularly effective when planning in domains with high-dimensional state spaces and a relatively limited budget of trajectories, because in that setting trajectories are likely to spread out in the state space quickly. As a result, data is spread too thinly over the state space to greatly improve decision making.

An illustration of they type of optimization performed by open-loop planning algorithms is presented in n Figures 3.2(a) and 3.2(b), which graphically show the return of one or two steps of open-loop planning (followed by a near-optimal solution, for the sake of illustration) in the double integrator domain (Santamaría et al., 1996). Although figures only render the fitness landscape for $H = 1$ and 2, that optimization can naturally be extended to an arbitrary number of steps in the future, with the related fitness landscape becoming more complex accordingly as the dimension of the problem grows.

Although closed-loop planners may exhibit provably poor results in stochastic domains with particular structure, there are also proofs of performance in



(a) Open-loop optimization in 1 dimension.



(b) Open-loop optimization in 2 dimensions.

Figure 3.2: Fitness landscape of open-loop planning in the double integrator, based on $H = 1$ or 2.

deterministic domains. Along with the results presented next, guarantees exist such that, if the domain is Lipschitz smooth, open-loop planning methods are complete (meaning it will find a desired goal state), even in the presence of noise (Yershov and LaValle, 2010).

3.6.1 Hierarchical Open-Loop Optimistic Planning

The central concept of this section is that optimization algorithms can be applied to planning simply by an appropriate casting of the problem. This general approach has previously been examined in Bayesian and PAC-MDP settings (Duff and Barto, 1997, Kaelbling, 1993, Strehl and Littman, 2004, Even-Dar et al., 2006). When performing planning in this manner, the optimization space is all action sequences of length H, $\mathbf{A}$, and vectors representing a solution to that optimization problem, $\mathbf{a}$, encode a sequence of actions. Correspondingly, the evaluation function executes $\mathbf{a}$ in the domain and produces the return of the resulting trajectory.

In this section, we will discuss the application of HOO to open-loop planning, which we name Hierarchical Open-Loop Optimistic Planning, or HOLOP (Bubeck and Munos, 2010, Weinstein and Littman, 2012, Schepers, 2012). A rare property of HOO that allows it to be used in such a manner is its ability to tolerate noise while performing optimization, which arises due to stochasticity in MDPs where policies are evaluated. Presented in Algorithm 11, HOLOP is a rollout planner that is a simple wrapper around HOO (Algorithm 5). In HOLOP, there are exceptions to the standard rollout model (Algorithm 7), because, as an open-loop planner, HOLOP does not perform action selection conditioned on state in the rollout. Therefore, SELECTACTION produces the entire action sequence $\mathbf{a}$ that is executed in the rollout, and UPDATE is called only

after the entire sequence is executed with the resulting return for all of **a**.

Algorithm 11 Hierarchical Open-Loop Optimistic Planning

```
1: function GREEDY(s_0)
2:     ν ← ν_0 of HOO
3:     while ν is not a leaf do
4:         ν ← argmax_{c∈C(ν)} R̂(c)
5:     return a ∈ A(ν)
6: function SELECTACTION(s, h)
7:     return HOO.NEXTACTION
8: function UPDATE(a, q̂)
9:     HOO.INSERT(a, q̂)
```

As a planner, the properties of HOLOP are derived jointly from the open-loop manner in which it plans, as well as the particular algorithmic underpinnings of HOO. Because of the strong theoretical guarantees of HOO, HOLOP has a guaranteed fast rate of convergence to optimal open-loop behavior, and has regret of $\tilde{O}(\sqrt{N})$. In particular, guarantees of the regret of HOLOP are independent of $|\mathbf{A}|$, but this bound is only true as $N >> |\mathbf{A}|$, so in practice when the number of trajectories are fairly limited the size of the domain has an impact on performance. This property, however, is an unavoidable aspect of local planning, and the fact that theoretical bounds are independent of S, A and H (as the number of trajectories grows large) is to our knowledge unique among planning algorithms.

While the regret of HOLOP is optimal, its simple regret is not (Bubeck and Munos, 2010). In particular, HOO has an expected simple regret of

$$\tilde{O}\left(N\frac{\log 1/\gamma}{\log\kappa + 2\log 1/\gamma}\right),$$

where κ describes the number of near-optimal sequences of actions. This bound on simple regret is actually similar to the bound for naive uniform planning, so depending on the measure of performance that is relevant to a particular use

scenario, the performance of HOLOP may be near optimal (in terms of regret), or fairly poor (in terms of simple regret).

Because HOLOP is an open-loop planner, it functions identically in domains with discrete, continuous, hybrid, and partially observable state spaces. In addition to these properties, HOLOP plans in continuous domains without the risk of divergence that occurs from the use of value-function approximation. Likewise, because the policy is represented by a sequence of actions, as opposed to parameterization of an FA (as is the case in traditional policy search), the algorithm will always be able to represent the sequence action sequence $\mathbf{a}^*$ that produces optimal returns.

In terms of empirical results, HOLOP has been shown to outperform a number of continuous planning algorithms that use other forms of tree decomposition to conduct planning in continuous MDPs (Schepers, 2012). In the full RL setting where a generative model is not provided, HOLOP combined with multi-resolution exploration (Nouri and Littman, 2008) to perform exploration, and k-d trees to conduct model building, was found to outperform a number of continuous RL algorithms (Weinstein and Littman, 2012), including Ex$\langle a \rangle$ (Martín H. and De Lope, 2009), which won the 2010 reinforcement learning competition in the high dimensional helicopter control task (Whiteson et al., 2010).

3.7 Discussion

This chapter has dealt with planning in continuous domains. Due to the fact that existent global continuous planners are not applicable in the setting considered (because of costs, risk of divergence, convergence to local optima, or

need for significant domain expertise), the focus of the chapter is on local continuous planners which do not suffer from these issues. A number of novel closed and open-loop planning algorithms are introduced for differing combinations of discrete and continuous state and action spaces, as well as fully continuous domains.

Focus is placed on HOLOP, which has many desirable characteristics. Due to the strong theoretical underpinnings of HOO, the quality of actions selected provably improves rapidly during planning. The planner itself is state agnostic and behaves identically regardless of the size of the state space, and whether the domain is discrete, continuous, hybrid, or partially observable. Indeed, this algorithm is used in Chapter 4 to demonstrate the superiority of planning algorithms that run natively in continuous MDPs over those that require coarse discretization of continuous dimensions to plan in continuous domains. Additionally, the fundamental idea of optimization as planning that underpins HOLOP is revisited in Chapter 5 to produce state of the art results in extremely high-dimensional, complex domains.

Chapter 4

Empirical and Analytic Comparison: Discrete Versus Continuous Planners

In this chapter, we compare a number of planning algorithms discussed or introduced in this work, specifically UCT, FSSS-EGM, OLOP, HOOT, and HOLOP. These algorithms cover the state of the art in planning from both empirical and theoretical perspectives, and are designed for differing combinations of discrete and continuous state and action spaces. All domains tested have fully continuous state and action spaces, and vary in size from small domains with 2 state and 1 action dimension up to very large domains with 16 state and 5 action dimensions. In all cases, planning algorithms are presented only with an EGM of the domain, with bounds on allowed rewards and action ranges. Planning is always restarted entirely anew at each planning step to test the effectiveness of the planning algorithms in the absence of evaluation functions, shaping, warm-starting, and any other enhancements.

4.1 Planning Algorithms Revisited

In practice, UCT is currently the state of the art, dominating recent general planning competitions (Kolobov et al., 2012), as well as computer Go tournaments (Gelly and Silver, 2008). On the other hand, the theoretical guarantees of the algorithm are extremely poor, as it may be outperformed by naive uniform

planning in domains with particular structures. Most domains presented here, however, have fairly smooth value functions, so the properties that are known to be problematic for UCT are not present. Based on all these factors, UCT should be regarded as the most competent discrete planner we could compare against, especially in the domains considered.

Aside from the fact that both are discrete rollout planners, FSSS-EGM is in many ways quite different from UCT. Unlike UCT, the original FSSS has strong theoretical properties based on accurate upper and lower bounds of the return of each $\langle s, h, a \rangle$, and will therefore never take a super-exponential number of samples to find the optimal policy. On the other hand, the algorithm has not seen thorough empirical testing, and unlike UCT, FSSS has not been selected for use in prominent planning competitions. The only known published results of the algorithm are from its original presentation, and have it in some cases outperforming, and in some cases being outperformed by UCT (Walsh et al., 2010).

OLOP goes even futher in this direction as it has extremely strong theoretical backing, but has had limited empirical examination. The only results we are aware of indicate that the performance of OLOP in practice is fairly poor, and roughly equivalent to uniform planning (Busoniu et al., 2011). OLOP has the distinguishing property of being a discrete action open-loop planner, so it selects sequences of action while ignoring state. One of the reasons for its selection is that it is the closest discrete analogue to HOLOP.

In contrast to OLOP, which requires discretization only of the action space, HOOT requires discretization only of the state space, as it adaptively decomposes the action space according to data acquired during planning. Since HOOT performs planning in a manner highly similar to UCT, it likewise suffers from

a lack of formal guarantees, due to the difficulty of the analysis of nonstationary return estimates that evolve with policy changes over time. Related weighted algorithms of WUCT and WHOOT, discussed in Chapter 3, are not tested here due to the heavy computational requirements that are a product of the memory-based operation of those algorithms.

HOLOP, finally, does not require any discretization as it is state agnostic and functions naturally in domains with continuous action spaces. Like OLOP and FSSS, HOLOP has very strong theoretical guarantees, as it is based on HOO, which has optimal regret, and also has formally analyzed (although suboptimal) simple regret. As an open-loop planner, its optimal open-loop policies may still be poorer than those of closed-loop planners. The implementation of HOLOP used here employs simple root parallelization (Weinstein and Littman, 2012, Chaslot, Winands, and Van den Herik, 2008). Empirical results in this section show that HOLOP has the desirable properties of having both strong theoretical backing as well as excellent results in practice.

While there is a fairly small number of other fully continuous planners we could select from for empirical testing, the ultimate goal in this chapter is not to document which continuous planner is best, but rather to show empirical superiority of a continuous planning algorithm over state of the art discrete planners such as UCT, FSSS-EGM, and OLOP, when applied to canonical continuous domains.

4.2 Domains

All domains are based on actual physical systems in some form, but have differing properties relating to linearity, smoothness, dimensionality, and the

existence of terminal states, among many other factors. For example, while the double integrator can be controlled optimally according to a simple policy (Sontag, 1998), the other domains tested do not have such properties, making optimal solutions extremely difficult to find.

4.2.1 Double Integrator

The double integrator domain (Santamaría et al., 1996) models the motion of a point mass along a surface. The object starts at some position and must be moved to the origin (corresponding to a position p and velocity v of 0) by accelerating the object by a selected amount at each time step, balancing immediate and future penalties. The dynamics of the system can be represented as a discrete time linear system as follows: $s' = T_1 s + T_2 a$ where $T_1 = \begin{bmatrix} 1 & 0 \\ \Delta t & 1 \end{bmatrix}$, $T_2 = \begin{bmatrix} \Delta t \\ 0 \end{bmatrix}$.

The reward is quadratic and is defined as $-\frac{1}{D}\left(s^T Q s + a^T R a\right)$ where $Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$, $R = [1]$.

Due to the characteristics of the domain, the value function and optimal policy are a quadratic function of state, making the domain relatively simple to plan in, and it therefore allows for a simple baseline against which performance can be presented. In the experiments here, the initial state is set to $(p, v) = (0.95, 0)$. Stochasticity is introduced by perturbing all actions taken by ± 0.1 units uniformly distributed.

In later experiments, the agent will be required to control multiple double

integrators simultaneously. By extending the A, B, Q and R matrices to create an appropriate number of position velocity dimensions. In all cases, agents are allowed to plan from 200 trajectories per step, and episodes are 200 steps long. Optimal performance is -1.312 ± 0.001, and random performance is -23.957 ± 2.342.

4.2.2 Inverted Pendulum

The second domain tested is the inverted pendulum, which models the physics of a pendulum balancing on a cart (Wang et al., 1996, Pazis and Lagoudakis, 2009), where actions are in the form of force applied to the cart on which the pendulum is balanced. Like the double integrator, the domain has 2 state dimensions and 1 action, but is more complex than the double integrator because it, like the other domains to follow has nonlinear dynamics and does not have a trivially computable optimal policy. Additionally, poor policies can lead to terminal failure states (or states from which a terminal state is ultimately unavoidable), which do not exist in the double integrator, introducing discontinuities in the value function.

The state $s = \langle \theta, \dot{\theta} \rangle$ consists of the angle and angular velocity of the pendulum, and the action is the force in Newtons applied to the cart. The dynamics of the domain are computed in terms of the angular jerk of the pendulum:

$$\ddot{\theta} = \frac{gsin(\theta) - \alpha ml\dot{\theta}^2 \sin(2\theta)/2 - \alpha \cos(\theta)a}{4l/3 - \alpha ml \cos^2(\theta)},$$

where $g = 9.8m/s^2$ is the gravity constant, $m = 2$ is the mass of the pendulum, $M = 8$ is the mass of the cart, $l = 0.5$ is the length of the pendulum, and $\alpha = 1/(m + M)$. The control interval is set to 100msec.

The reward function in this formulation favors keeping the pendulum as

close to upright as possible using low magnitude actions and maintaining low angular velocities of the pendulum:

$$R(\langle\theta,\dot{\theta}\rangle,a) = -\left((2\theta/\pi)^2 + \dot{\theta}^2 + (a/50)^2\right),$$

with $|\theta| > \pi/2$ leading to the end of the episode with a reward of -1000.

Noise is introduced by perturbing the actions by ± 10 Newtons uniformly distributed. The full action range is $(-50, 50)$ Newtons. Like the double integrator, some experiments will test the ability of planning algorithms to scale by controlling a number of independent pendulums simultaneously. In all cases, agents are allowed to plan from 200 trajectories per step, and episodes are 200 steps long. Random performance is -1273.202 ± 80.746.

4.2.3 Bicycle Balancing

Bicycle balancing is a popular medium-sized domain (Randlov and Alstrom, 1998, Li et al., 2009). This domain is highly nonlinear with regards to dynamics and values, and is considered to be one of the most difficult canonical reinforcement learning domains, with most algorithms requiring pre-supplied basis functions and shaping to plan effectively (Lagoudakis and Parr, 2003). Although it is a fully continuous domain, earlier publications use algorithms that only plan over a discrete set of actions and rely on a particular hand-engineered discrete set of actions (not strictly a coarse discretization) to make successful planning possible due to the difficulty of maintaining balance (Lagoudakis and Parr, 2003).

The domain has a 4-dimensional state space and 2-dimensional action space. The state consists of the angle and angular velocity of the handlebars with respect to the body, and the angle and angular velocity of the body with respect

to the ground. The actions are torque applied to the handlebars and displacement of weight from the bicycle. The full dynamics are fairly complex and can be found with other details in Randlov and Alstrom (1998). The one distinction of the domain tested here is with regards to the reward function:

$$R(\langle \omega, \theta \rangle, a) = -\left(\left(\frac{\omega}{\pi/15} \right)^2 + \left(\frac{\theta}{\pi/2} \right)^2 + \left(\frac{a_1}{2} \right)^2 + \left(\frac{a_1}{0.02} \right)^2 \right),$$

with ω being the angle of the bicycle relative to the ground, θ being the angle of the handlebars relative to the body of the bicycle, and $a = \langle a_1, a_2 \rangle$ being the torque applied to the handlebars and weigh displacement, respectively.

In all cases, agents are allowed to plan from 400 trajectories per step, and episodes are 200 steps long. Unlike the double integrator and inverted pendulum, no results controlling multiple bicycles simultaneously are presented as no planning algorithm was able to balance more than one bicycle with the number of planning trajectories provided.

4.2.4 D-Link Swimmer

In the D-link swimmer domain (Coulom, 2002, Tassa et al., 2007b), a simulated snakelike swimmer is made up of a chain of D links, where $D - 1$ joint torques must be applied between links to propel the swimmer to the goal point. The total size of the state space is $2D + 4$ dimensional, consisting of the absolute location and velocity of the head of the swimmer and angle and angular velocities of the joints. The swimmer's body exists in two dimensions, with all the body at the same depth in the liquid in which it is swimming. In the experiments here, planners must control swimmers with 3 to 6 links, which means the smallest domain has 10 state dimensions and 2 action dimensions, while the largest domain has 16 state dimensions and 5 action dimensions, making

it the largest domain that is used as a comparison. As is the case with the bicycle domain, the dynamics are highly complex. Full details can be found in Tassa et al. (2007a,c). In all cases, agents are allowed to plan from 300 trajectories per step, and episodes are 300 steps long. The appearance of the 3-link swimmer (the smallest tested) is presented in Figure 4.1, which also shows a stroboscopic rendering of the policy constructed by HOLOP in the experimental setting used here, with the goal location being the origin on the plane.

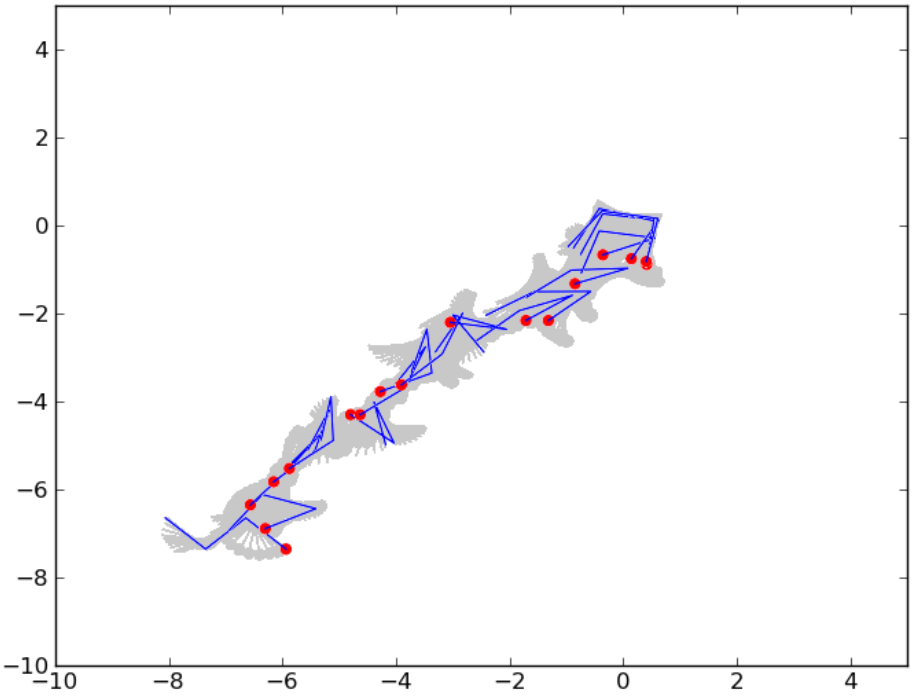


Figure 4.1: A depiction of the policy produced by HOLOP in the 2-link swimmer domain.

4.3 Results

We refer to an episode as the result of an algorithm interacting with the actual environment, and rollouts as being the result calculations based on the provided episodic generative model. In all empirical comparisons, the performance metric used is mean cumulative reward per episode. In all domain domains, the discount factor $\gamma = 0.95$. Rollouts are performed with $H = 50$, with the exception of OLOP, which computes the depth and number of rollouts based on the total budget of samples allowed (in practice, however, the number of rollouts N and H computed by OLOP were extremely close to the values selected *a priori* for the other planning algorithms).

4.3.1 Optimizing the Planners

Being that all domains considered are fully continuous, but only HOLOP operates natively in continuous domains, all other planning algorithms must plan in a discretization of the state and/or action dimensions. Because we do not assume that expert knowledge is available, a "good" parameterization is unknown and therefore coarse discretizations of state and action spaces are additional parameters that must be searched over to optimize performance of the planner. Here, it will be demonstrated that even when searching over a large number of possible discretizations for state of the art discrete planning algorithms, HOLOP, a native continuous planner, is able to produce lower sample complexity and higher quality solutions, while being more robust.

The performance of different planning algorithms according to discretizations are presented as "heat maps", where each cell in the map indicates the average performance of that particular parameterization. In these graphs, changes

along the vertical axis indicate discretizations in state space (if applicable), and changes along the horizontal axis indicate discretizations in the action space (again, if applicable). In this experimental setting, discretizations produce between 5 and 35 (in multiples of 5) different cells per dimension. Discretizations are considered separately between state and action dimensions, but within the state or action dimensions, the number of cells produced are not considered separately. We will refer to the number of cells per state dimension as σ and the number of cells per action dimension as α, and the resulting discretized state and action spaces as S' and A', respectively. Therefore, $|S'| = \sigma^{|S|}$, $|A'| = \alpha^{|A|}$.

Because of the significant computational costs of running experiments in the D-link swimmer, experiments testing the quality of varying parameterizations in that domain have not been produced. In particular, the 49 parameterizations that would need to be tested for UCT and FSSS-EGM discretizing state and action spaces are prohibitively expensive. Results from the double integrator, inverted pendulum, and bicycle balancing domains are presented below.

Double Integrator

The first set of heat maps is displayed in Figure 4.2. The top row, from left to right shows the performance of UCT, HOOT, and HOLOP, while the bottom row displays the performance of FSSS-EGM and OLOP. Because UCT and FSSS-EGM require discretizations of both state and action spaces, heat maps for those algorithms are checkered, with a total of 49 different parameterizations tested. Because HOOT only requires discretizations of the state space, its corresponding heat map has 7 horizontal stripes. Likewise, OLOP only

requires discretization of the action space, and has a heat map with 7 vertical stripes. Because HOLOP naturally functions in continuous MDPs, no discretization is required so the entire heat map is a constant color.

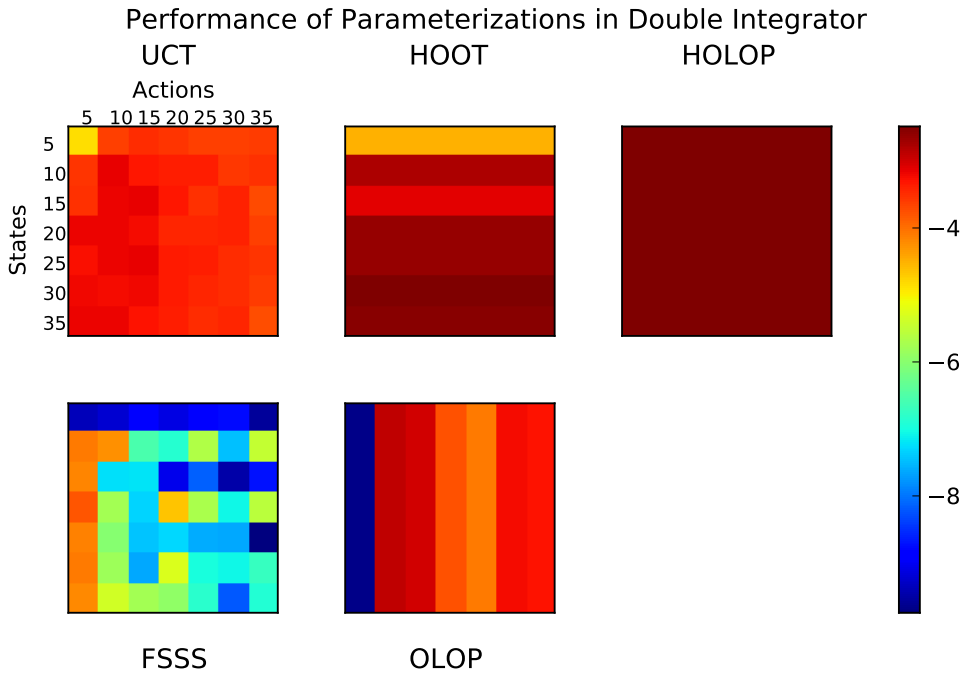


Figure 4.2: Heat map representation of performance UCT, HOOT, HOLOP, FSSS-EGM, and OLOP in the double integrator.

Both OLOP and FSSS-EGM have some parameterizations that lead to very poor performance in the domain, with cumulative rewards ranging to approximately -10, while the worst cumulative reward among UCT, HOOT, and HOLOP was achieved by UCT at -4.9. Because the heat map scales colors according to the entire range of values, another graph with OLOP and FSSS-EGM omitted is presented in Figure 4.3, to more clearly differentiate the better performing algorithms. In this domain, all parameterizations of UCT have cumulative rewards statistically significantly worse than HOLOP, and 3 of 7

parameterizations of HOOT are statistically significantly worse than HOLOP, while none are statistically significantly better.

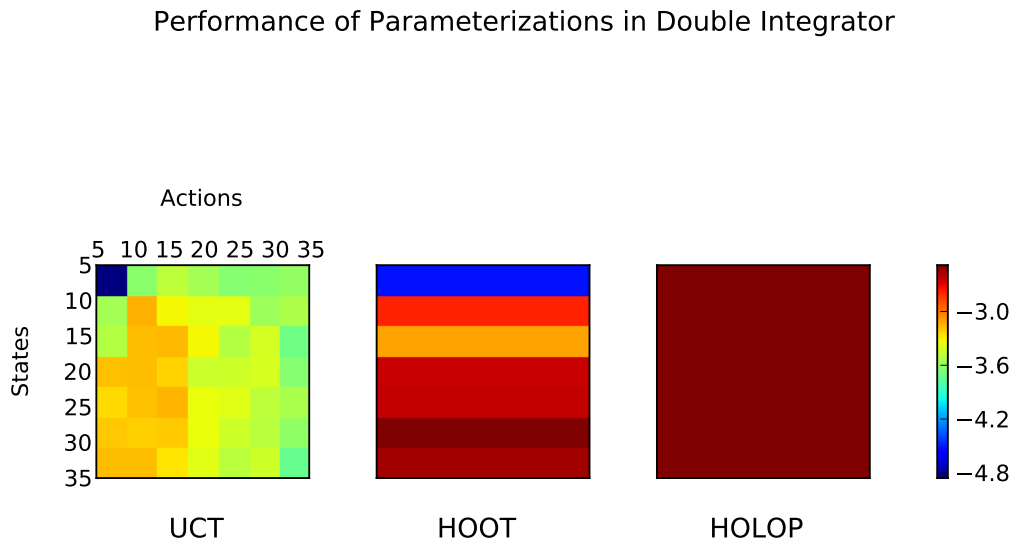


Figure 4.3: Heat map representation of performance UCT, HOOT, and HOLOP in the double integrator.

Inverted Pendulum

Results in the inverted pendulum domain are mostly consistent with results from the double integrator, with performance of OLOP and FSSS-EGM not competitive with UCT, HOOT, or HOLOP, as they were unable to consistently maintain balance of the pendulum. Both algorithms displayed policies consistent with myopic behavior, opting for very small magnitude actions because of the immediate penalty for higher-magnitude actions. This policy leads to some episodes with very good cumulative rewards (when stochasticity does

not push the pendulum off-balance), but leads to failure when noise begins to move the pendulum off-balance, as algorithms do not recover with necessary high-magnitude actions. This pattern of failure of OLOP and FSSS-EGM due to myopic decision making is consistent in all of the experimental domains, so their performance will be omitted for the remainder of the chapter to simplify presentation.

The performance of the best algorithms, UCT, HOLOP, and HOOT are displayed in Figure 4.4. The heat maps of UCT and HOOT both have a strange but noteworthy characteristic, which is alternating bands of quality with respect to discretizations of the state space. Specifically, discretizations of the state space into 5, 15, 25, and 35 cells performed poorly, while discretizations into 10, 20, or 30 cells performed relatively well. It is unclear what the cause of this artifact is, but the fact that it arises in both UCT and HOOT indicate that the phenomenon has more to do with properties of the domain than peculiarities of a particular planner.

It is worthwhile to dwell on this point for a moment to consider its implications. In contrast to what may be the common view on parameter search of discretization in experimental settings, these results show that it is not the case that there is some optimal discretization, with values close to that optimal parameter being better and others being worse. Instead, these results show that the impact of discretization on policy quality can be very unsmooth, and that great care must be taken to ensure that parameterizations considered create discrete state and action sets that allow for reasonable policies, as there can be potentially strange interactions between planners, domains, and discretizations.

As was the case in the inverted pendulum domain, every parameterization

of discretizations among the 49 tested for UCT was statistically significantly worse than HOLOP. Out of the parameterizations tested for HOOT, 4 were statistically significantly worse than HOLOP while none were statistically significantly better.

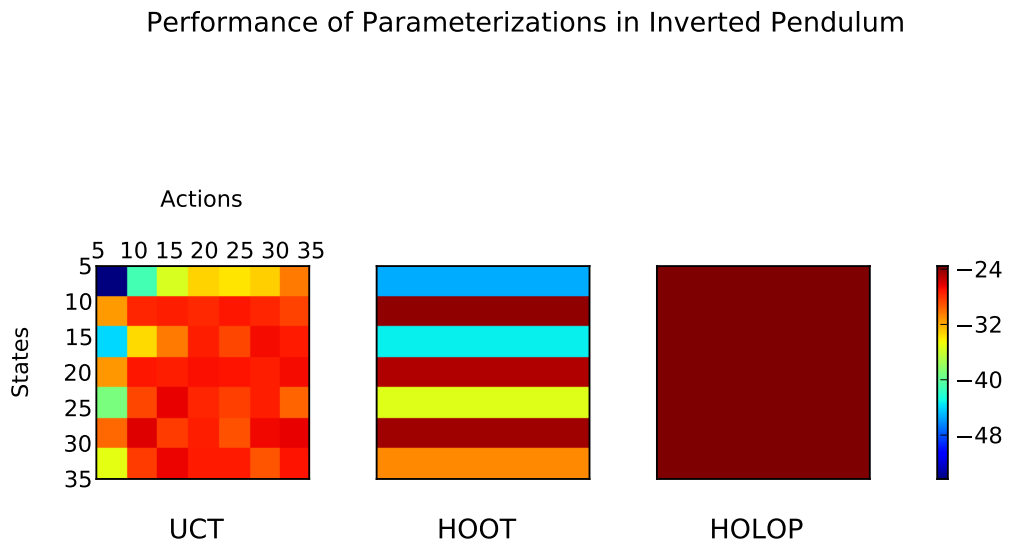


Figure 4.4: Heat map representation of performance UCT, HOOT, and HOLOP in the inverted pendulum.

Bicycle Balancing

Bicycle balancing is considered the most difficult baseline domain. Although it is smaller than the 3-link swimmer, it has terminal states that can be quickly reached by an ineffective policy. At 4 state dimensions and 2 action dimensions it is also twice as large as the double integrator and inverted pendulum. Empirical results in Figure 4.5 show that bicycle balancing causes the poorest

performance for UCT presented in this chapter. Although some parameterizations of UCT are not statistically significantly different from that of HOLOP, over half of the discretizations resulted in very poor policies similar to that of OLOP and FSSS-EGM, which were not able to consistently maintain balance. All parameterizations of HOOT, on the other hand, were not statistically significantly different from that of HOLOP.

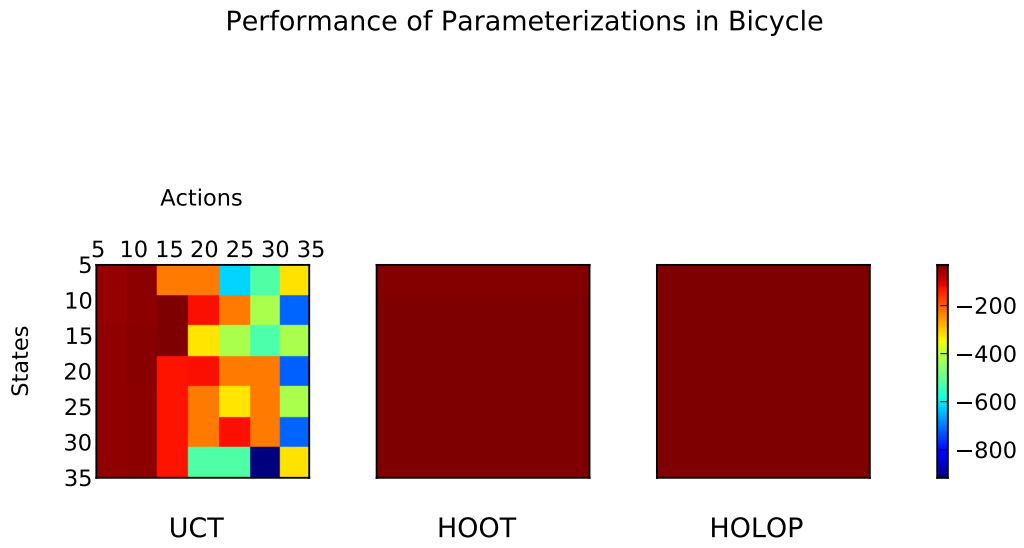


Figure 4.5: Heat map representation of performance UCT, HOOT, and HOLOP in the bicycle domain.

4.3.2 Scaling The Domains

In this group of experiments, the ability of planning algorithms to scale to large domains is tested, with the motivation that the combinatorial explosion of $|S'|$

and $|A'|$ with respect to $|S|$ and $|A|$ will have a negative impact on the ability of discrete planning algorithms (and in particular, UCT) to function in domains of higher dimension. In the first two domains presented, increasing domain size is achieved by creating new domains that are composed of a number of independent subproblems, the number of which we will refer to as D. In particular, agents must simultaneously control an increasing number of independent instances of the double integrator or inverted pendulum domains, while the number of samples available for planning remains fixed. The planning algorithms are not presented with the fact that the state and action spaces are composed of multiple, independent problems, which would greatly simplify the planning problem (Diuk et al., 2009). In all experiments, the discretizations used for UCT and HOOT are those with the best average value in the heat map experiments. Rewards are averaged over all instances, and a terminal state in one instance is treated as a terminal state for the entire domain. In the D-link swimmer, the complexity of the domain is increased by adding additional links to the swimmer.

As in Section 4.3.1, because the empirical performance of FSS-EGM and OLOP is not competitive with the other algorithms, their results are omitted from this section, as their presentation would only clutter presentation of the results of the more effective planning algorithms.

Double Integrator

Because of the smoothness of T and R, along with the lack of terminal states, the double integrator is the simplest domain tested, and it is therefore expected

that algorithms will scale more effectively in this domain than others. The cumulative reward of UCT, HOOT, and HOLOP, when faced by various numbers of instances of the double integrator, are presented in Figure 4.6. The first point on the x-axis corresponds to the original domain, while the point where x=5 corresponds to controlling 5 examples simultaneously, with $|S| = 10$ and $|A| = 5$. As can be seen, the performance of HOLOP and HOOT are not statistically significantly different, while the performance of UCT is statistically significantly worse than HOOT and HOLOP, regardless of the size of the domain. Furthermore, the gap in performance between UCT and HOLOP grows as complexity increases, with a gap of 0.61 growing to 1.84 by the end of the experiment.

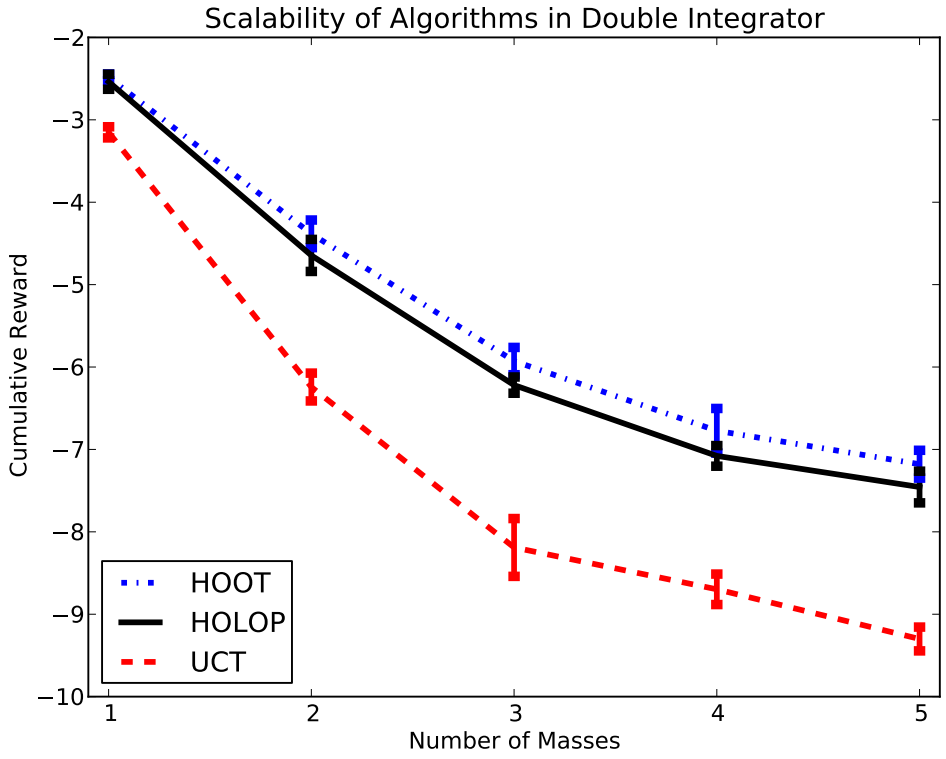


Figure 4.6: Performance of UCT, HOOT, and HOLOP while controlling multiple instances of the double integrator problem.

Inverted Pendulum

For the most part, the patterns that arose from increasing problem complexity in double integrator also hold in the inverted pendulum, although they are more exaggerated here, with the results presented in Figure 4.7. In particular, once again the performance of HOLOP and HOOT are not statistically significantly different, and UCT is statistically significantly worse. The poorer performance of UCT is very clear because, in the largest instances of the domain, it loses the ability to consistently balance the pendulum, leading to many episodes that end with a large penalty.

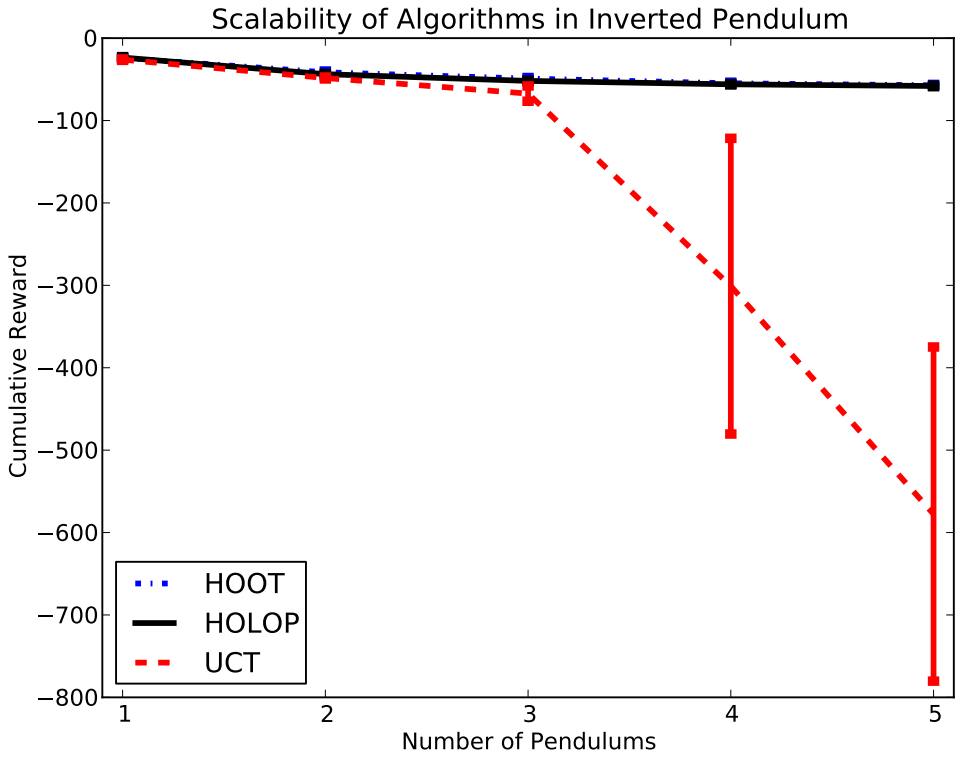


Figure 4.7: Performance of UCT, HOOT, and HOLOP while controlling multiple instances of the inverted pendulum problem.

D-Link Swimmer

Because of the significant costs of running simulations (due to more sophisticated methods of integration to estimate dynamics) in this domain, testing different parameterizations of α and σ for UCT and HOOT would be prohibitively time consuming. For this reason, heat map results are not presented for the D-link swimmer. Being that parameters could not be selected experimentally, α and σ were set to 5, with the motivation being that small values for these parameters would lead to the slowest (although still exponential) growth in $|S'|$ and $|A'|$, easing planning. Even with this parameterization, UCT still must reason over an enormous space $|S'| \times |A'| > 10^{14}$ for the largest domain of $D = 6$. The results of HOLOP, HOOT, and UCT in this domain are depicted in Figure 4.8. As is the case in previous domains, FSSS-EGM and OLOP are not competitive as their myopic decision making selects high-reward but low-value actions where no (or minimal) torque is applied, resulting in a swimmer that remains stationary for the duration of the experiment.

There are a number of items to note with regards to the results in this domain. While HOOT was able to perform essentially as well as HOLOP in domains where σ was selected carefully, when tuning is not, or cannot be performed, the performance of HOOT suffers significantly. In the case of this domain, performance of HOOT without tuning is always statistically significantly worse than HOLOP, and occasionally equivalent to the very poor UCT. Additionally, due the huge size of the discretized state and action spaces, the performance of UCT and HOOT with $D = 6$ is closer to that of chance than of HOLOP.

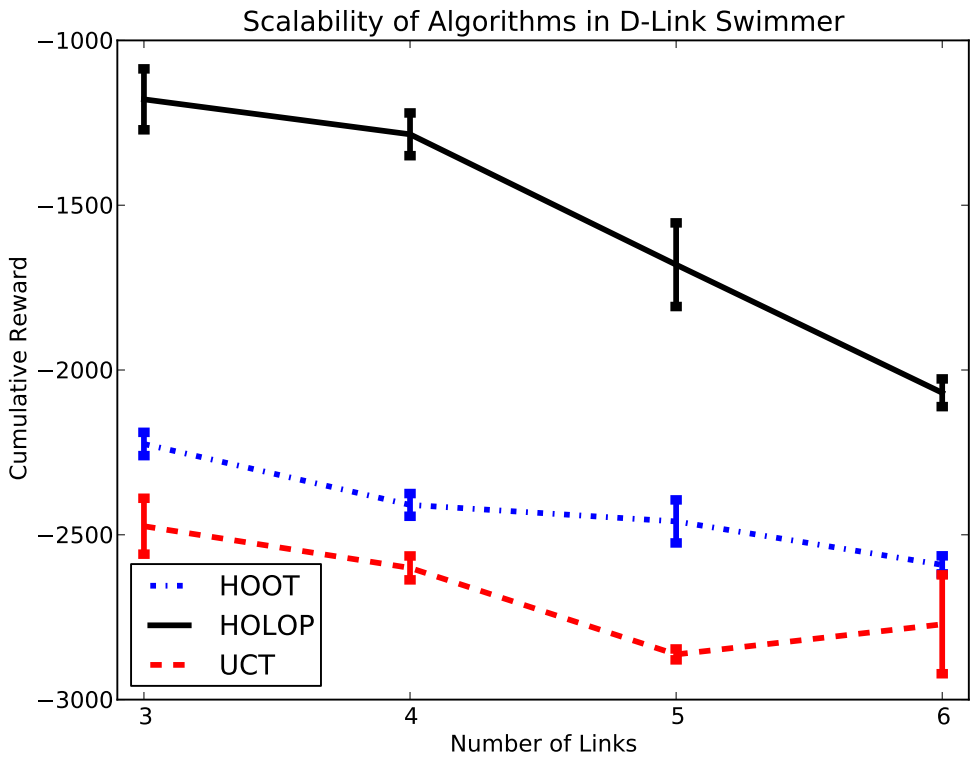


Figure 4.8: Performance of planning algorithms in 2- to 6-link swimmer.

4.3.3 Sample Complexity

Previous scaling experiments demonstrate the performance of planning algorithms with a fixed budget of samples, as domains increase size. Another evaluation approach is to consider how many samples planning algorithms need to match the performance of HOLOP. In this experiment, we use the performance of HOLOP as a baseline, and examine the factor of samples UCT needs over HOLOP to reach the same level of performance. (HOOT is ignored as it almost always has performance indistinguishable from HOLOP when discretization is optimized.)

In particular, we consider the scaling problems for the double integrator and inverted pendulum domains. Using the performance of HOLOP with 200

trajectories as a baseline for each problem, we repeatedly double the number of trajectories available for UCT to use (starting at 400 trajectories, as-in all cases-it has worse performance with 200 trajectories per step), and stop doing so once the difference in performance between HOLOP and UCT cease to be statistically significantly different. Essentially the goal is to measure the factor of samples needed to shift the error bars of UCT up to overlap with those of HOLOP in Figures 4.6 and 4.7. We call this measure (the relative number of trajectories used) the *sample complexity*, and present the results of UCT as compared to HOLOP in terms of sample complexity in Figure 4.9.

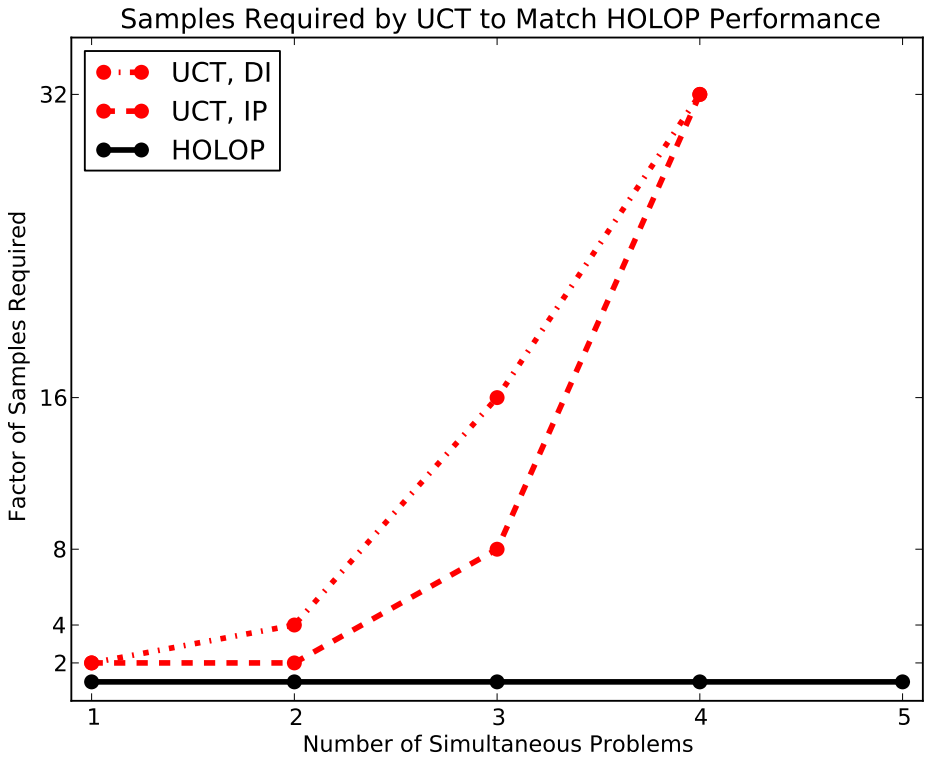


Figure 4.9: Sample complexity of UCT as compared to HOLOP in scaling problems of the double integrator and inverted pendulum

As can be seen quite clearly, the number of samples needed by UCT grows

superlinearly in the number of problems being controlled, and the curve appears to have exponential growth. When controlling 4 masses or pendulums, UCT needs 32 times the number of trajectories given to match the performance of HOLOP for the same problem. Results for what would be the final point on the x-axis, where 5 problems are controlled, are not shown because even at 32 times the number of trajectories, the performance of UCT is still statistically significantly worse than HOLOP in both problems, and running the domains with $N = 12800$ (or more) trajectories per planning step is unreasonable given the amount of time needed to run such experiments.

There is an excellent explanation as to why the numbers of trajectories needed by UCT would grow exponentially to match the performance of HOLOP. As parameterized, the number of actions available by UCT in both domains increases by a factor of 10 every time problem complexity increases (the size of the state space increases even more quickly). Because coarse discretization has no means of generalizing, the number of samples needed must grow exponentially to get a sufficient number of samples of each policy to maintain quality (which devolves into a selected initial action followed by a random action sequence).

Until the number of samples increases exponentially to match the exponential growth in actions, each initial action simply does not have enough data to perform an accurate estimate of quality, due to the signal and noise issue discussed in Section 2.4.3. HOLOP, on the other hand, performs adaptive discretization and is able to generalize effectively. As such, even based on a limited number of samples, HOLOP is always able to give some reasonable estimate of action quality for any point in the entire action space, and therefore suffers much less from increasing problem size.

4.4 Running Time and Memory Usage

One reason that native continuous planners have traditionally been avoided is the common belief that costs (both in terms of computation and memory) of continuous methods are prohibitively large when compared to those of discrete algorithms. In this section, we debunk this misconception. An interesting aspect of working in high dimensional domains is that, as domains grow in size, steps that are taken as trivial when operating in small discrete domains can dominate costs, and even make planning prohibitively expensive. As such, continuous methods can be significantly less expensive than discrete planning methods. In particular, we show that HOLOP has almost constant planning times and memory requirements, while UCT has an exponential growth in running time and memory usage as dimension increases. In terms of the analysis, we will consider S, A, H, N as variables that may influence costs. We will make the reasonable assumption that $|A| << N << |A'|$, as $|A'|$ grows exponentially in $|A|$, and likewise for state.

4.4.1 Illustration of Signal-to-Noise Problem

Local search presents planning in stochastic domains as a signal-to-noise problem. That is, fixed policies (both closed- and open-loop) will produce a distribution of returns when executed in the domain. The goal of a planner, then, is to create reliable estimates of return based on a finite amount of noisy data. In low-dimensional domains, the rollout budget is generally large enough to construct reasonable estimates regardless of the planning technique. Once planning moves to high-dimensional domains, reasoning more carefully about data becomes critical. Coarse discretization, however, is very poor at doing such

computation. This is because data is not generalized outside of cells, and as the number of cells explodes in higher dimensional domains, the amount of samples per cell vanishes to either 0 or 1 samples. Methods constructed to plan natively in continuous domains, however, are capable of reasoning about data in a more sophisticated manner that allows for better decision making, especially in larger domains.

As an illustration, an experiment is conducted in the 2-double integrator. In particular, the return estimates of UCT (with a 10x10 discretization) are compared with that of HOLOP. Just as in the previous experimental setting, $N = 200$. Because HOLOP performs an adaptive decomposition of the action space, it performs a hierarchy of estimates, with average returns recorded for tree depths of 3, 5, and 6 presented in Figures 4.10, 4.10 and 4.11, respectively. In each figure, the region of the action space that is selected by the adaptive decomposition at the end of planning is represented in pink. The importance of this adaptive discretization is highlighted in Figure 4.11, which has an additional region shaded in green, which has a better average reward than the pink region (selected by HOLOP). This distinction is important because the green region corresponds to poorer actions, but appears better based on the available noisy data. A more naive approach that does not reason as carefully about available data, therefore, would end up selecting an action from the green region, producing a suboptimal policy.

Indeed, this exact behavior is what occurs with UCT, shown in Figure 4.11. Because $N = 200$, even from the root, UCT is able to select each action at most 2 times, leading to a very limited amount of data to attempt to retrieve the desired signal (the optimal action) from the present noise (stochasticity in T, as well as in the randomized action selection in UCT lower in the tree).

Longer rollouts produce higher variance results (Gabillon et al., 2011). UCT has been shown to produce particularly high variance rollouts, and bagging has even been proposed to help mitigate the issue (Fern and Lewis, 2011). Modifications in UCT that allowed a variant to achieve master level play in 9x9 Go was designed specifically to help reduce variance of estimates (Gelly and Silver, 2008). Additionally, because this has been identified as a problem when performing local search, work has explored how to compute estimates of bias and variance of Monte-Carlo algorithms (Fonteneau et al., 2010).

4.4.2 Analytical and Empirical Memory Costs

With regards to memory, data structures built by HOLOP will be smaller than those built by UCT. For every $\langle s, h \rangle$ pair that is visited by UCT, a node must be created that $\forall a \in A'$ maintains a constant amount of information. In the worst case, $\langle s, h \rangle$ are never revisited, leading to memory costs of $O(HN|A'|)$. In the 5-double integrator, as considered in this section, the worst-case memory cost is quite large at $50 \cdot 200 \cdot 10^5 = 10^9$, and is incurred during *each* planning step.

HOLOP, on the other hand, builds a data structure that always grows by a single node every trajectory, as opposed to every step of every trajectory. Additionally, each node contains a constant amount of information, as opposed to UCT which requires $|A'|$ memory for each node. Constant costs can be achieved in HOLOP by representing $A(v)$ only by the index in $H|A|$ where the child differs from the parent, and what the value of the difference is (this incurs an additional $\log N$ computational cost but such a cost is already incurred during normal execution). Because one such node is built only after each trajectory, the memory requirement of HOLOP is $O(N)$. Because we assume $N << |A'|$ (Section 4.4), UCT has significantly higher memory requirements as compared

Figure 4.10: Estimated initial action quality as estimated by HOLOP at tree depth 2 and 4.

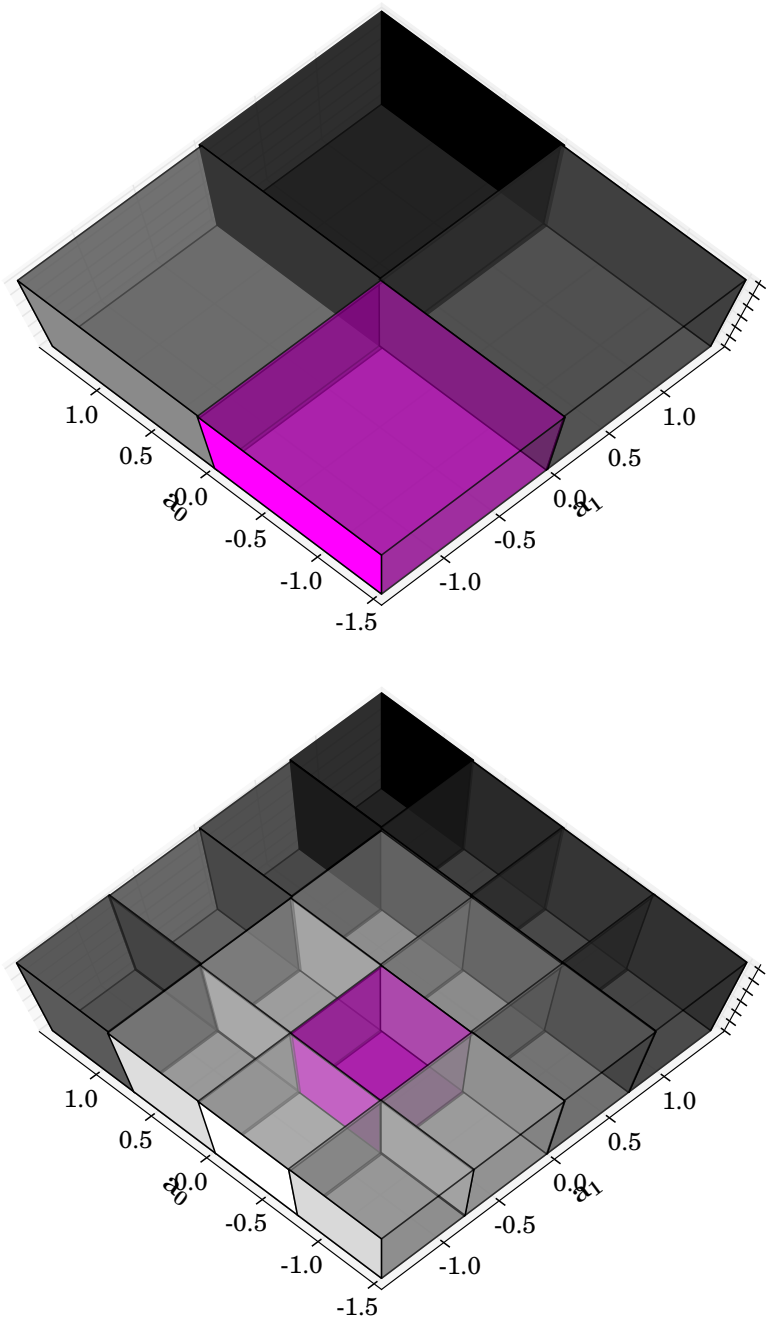
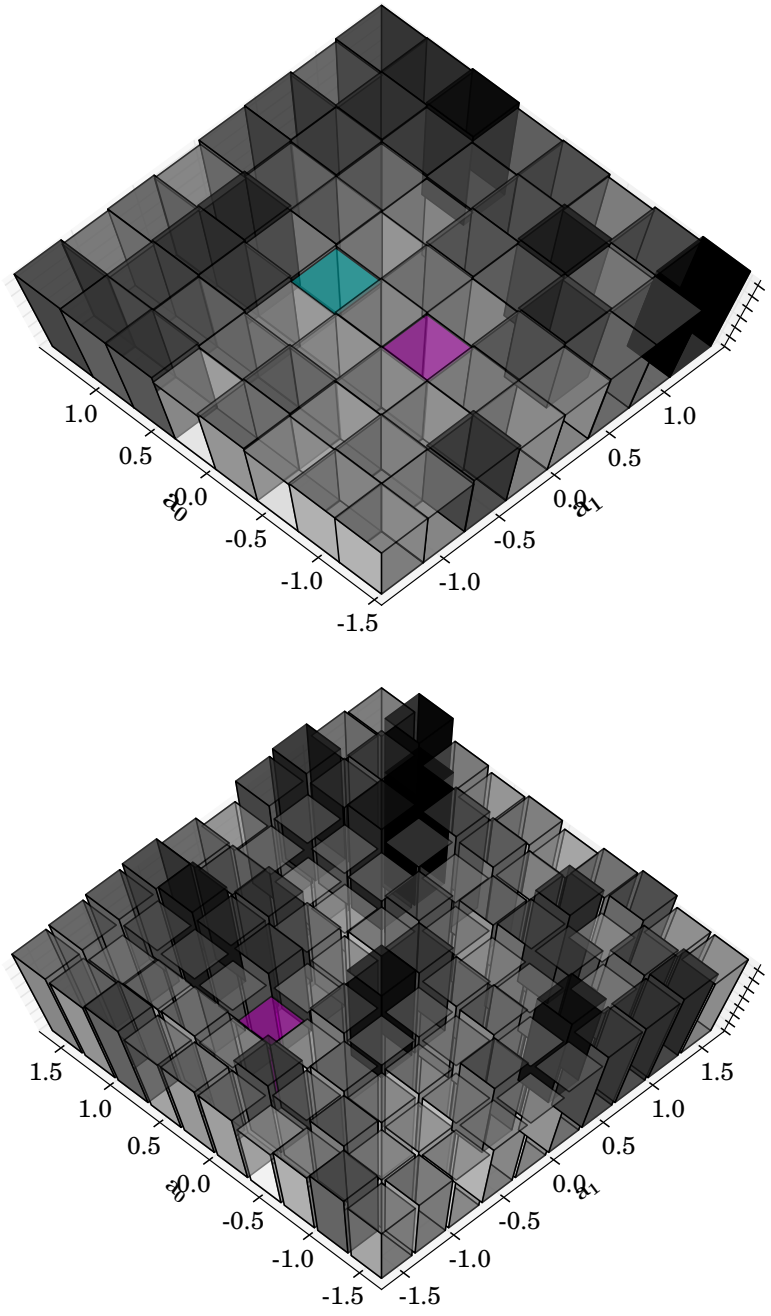


Figure 4.11: Estimated initial action quality as estimated by HOLOP at tree depth 6, and by UCT with $\alpha = 10$.



to HOLOP.

As the overall framework used by HOOT closely resembles UCT, with an algorithmic underpinning of HOO, it has memory costs somewhere between UCT and HOLOP. Unlike UCT, HOOT does not have different memory costs based on whether $\langle s, h \rangle$ are revisited. Because whereas UCT must maintain statistics on each $a \in |A'|$ for each node, HOOT only adds one constant cost HOO node regardless of whether that $\langle s, h \rangle$ was previously visited or not. The distinction is that whereas HOLOP adds one node during each trajectory, HOOT adds H per rollout. Therefore, the memory requirements of HOOT are greater than HOLOP, with a cost of $O(NH)$.

The actual memory usage of the 3 algorithms in the D-double integrator are presented in Figure 4.12. A few background items are worth discussing. Firstly, the implementations are in Python 2.6 64 bit (van Rossum and de Boer, 1991, Dubois et al., 1996), and as such exist in an environment devoid of direct memory control, so numbers reported here should be taken as only approximate values, keeping various items such as garbage collection and imported libraries into consideration. The entire experimental environment aside from the planning algorithms was subtracted from the memory use displayed in Figure 4.12. Secondly, the implementations used here emphasize correctness and simplicity over optimization of computational or memory costs.

The actual memory results have a number of noteworthy elements, and are in line with analytical results. Firstly, HOOT is shown to have very high memory costs, which is between roughly 10 to 25 as much memory as used by HOLOP depending on the particular problem instance. This value is not terribly far from the factor of $H = 50$ expected by the analytical results when considering the actual environment the experiments are run in. Also, memory

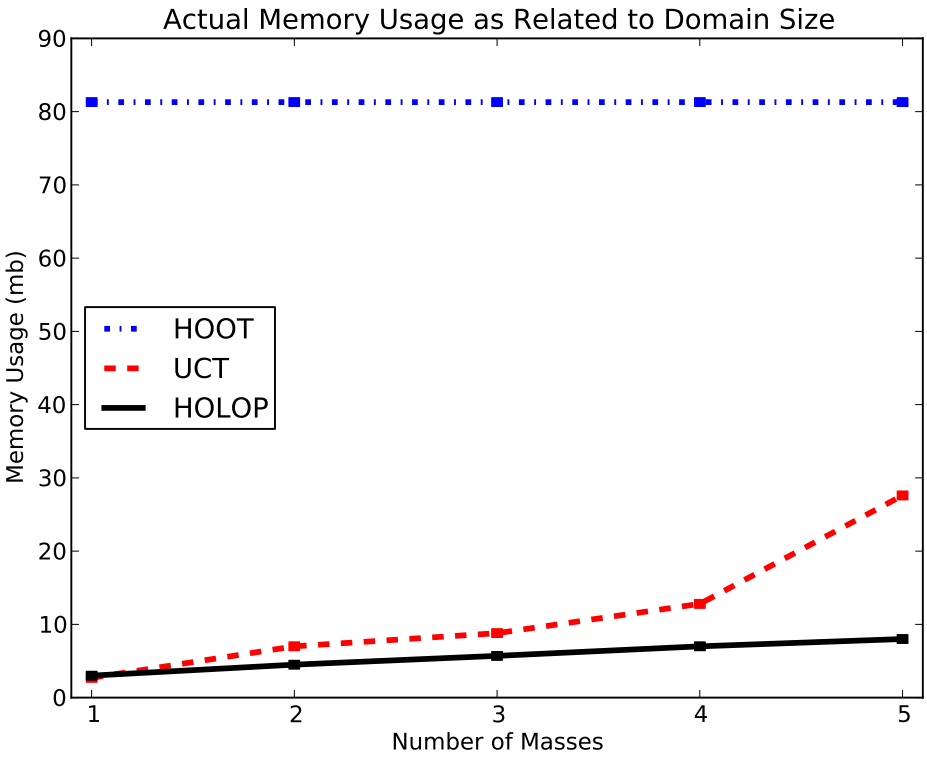


Figure 4.12: Memory usage of UCT, HOOT, and HOLOP in megabytes in the scaling domain of the double integrator.

usage of both HOOT and HOLOP are shown to be nearly constant with respect to domain size.

The only algorithm that is significantly impacted by increasing domain size, also in agreement with the analytical memory requirements, is UCT. Although somewhat difficult to discern due to the scaling imposed by HOOT, it is clear that the growth in memory usage in UCT is exponential. In particular, the entire change in memory from $D = 1$ to 4 is 10 mb of memory, whereas the change in memory usage from $D = 4$ to 5 is more than double that value at 24 mb. Based on this rate of growth in memory use, and the almost flat memory use of HOOT and HOLOP, it is clear that in domains of only slightly larger size UCT would have the largest memory costs. Indeed with $D = 6$ (not shown) the

costs of HOOT and HOLOP are essentially unchanged, while UCT becomes the most expensive planner by a large margin, requiring approximately 200 megabytes of memory.

4.4.3 Analytical and Empirical Computational Costs

For the most part, proof that computational costs of UCT are heavier than HOLOP follow from the analysis of the memory requirements. Once UCT stores data on $\forall a \in A'$, it has already incurred an equivalent computational cost of $|A'|$, which we assume dominates other variables (aside from $|S'|$). The main difference between the two costs are that memory costs are worst case, but computational costs are the same in best and worst case analysis. During each step of each rollout, even if an $\langle s, h \rangle$ is reencountered, UCT must compute $\max_{a \in A} U(s, h, a)$. There is no simple way to work around this cost, as all $U(s, h, a)$ change continuously and must be recomputed at each time step (some other planners can mitigate this cost by maintaining priority queues based on value estimates of all $a \in A'$). It is worth mentioning that simply due to interaction with the generative model, executing rollouts has computational costs of $O(HN(|S| + |A|))$, so this cost will appear in all analyses. During each step of each rollout, UCT simply computes $\max_{a \in A'} U(s, h, a)$, followed by some constant-cost operations, leading to a computational complexity of $O(HN(|A'| + |S|))$.

The cost of traversing the tree built by HOLOP to conduct action selection is $O(\log N)$, and incorporating the results of the rollout are constant-time operations, so the computational cost of the entire algorithm is $O(N(\log N) + H(|A| + |S|))$. HOOT has an equivalent $O(\log N)$ cost, but this cost is incurred

during each step in the rollout, as opposed to once for each rollout. Combining the computational cost of tracking state results in a computational cost of $O(HN(\log N + |A| + |S|))$. From a theoretical standpoint, UCT has the highest computational requirements, HOOT is in the middle, and HOLOP is the most computationally efficient.

The $O(N \log N)$ times are based on an optimization of the original presentation of HOO, which does not impact the formal regret bounds. The original presentation of the algorithm requires recomputing U and B for the entire tree at every time step, and therefore has an $O(N^2)$ running time just to interact with the HOO tree. Based on our testing, however, the version of the algorithm that has the $O(N^2)$ running time has statistically significantly better performance than the $O(N \log N)$ version, so the empirical results in this chapter are presented with respect to the $O(N^2)$ version of the algorithm. The difference in observed performance is because exploration occurs more uniformly in the $O(N \log N)$ version, which impacts performance in practice.

To determine whether real-world computational costs match the analytic values, computation times of 1 round of planning in the D-double integrator are presented in Figure 4.13 (note that the y-axis is log-scaled). Based on the results, it is clear that UCT has exponentially larger computational costs than HOOT or HOLOP. In fact, the curve for UCT is slightly super-linear as scaled in the graph, so in practice the running costs of UCT actually look super-exponential. Clearly, UCT has the heaviest computational requirements, going from the fastest planner at $D = 1$ to taking over 350 times as long as HOLOP to conduct planning when $D = 5$.

Aside from the extremely high computational costs of UCT, a few other points are worth mentioning. HOOT is actually significantly more expensive

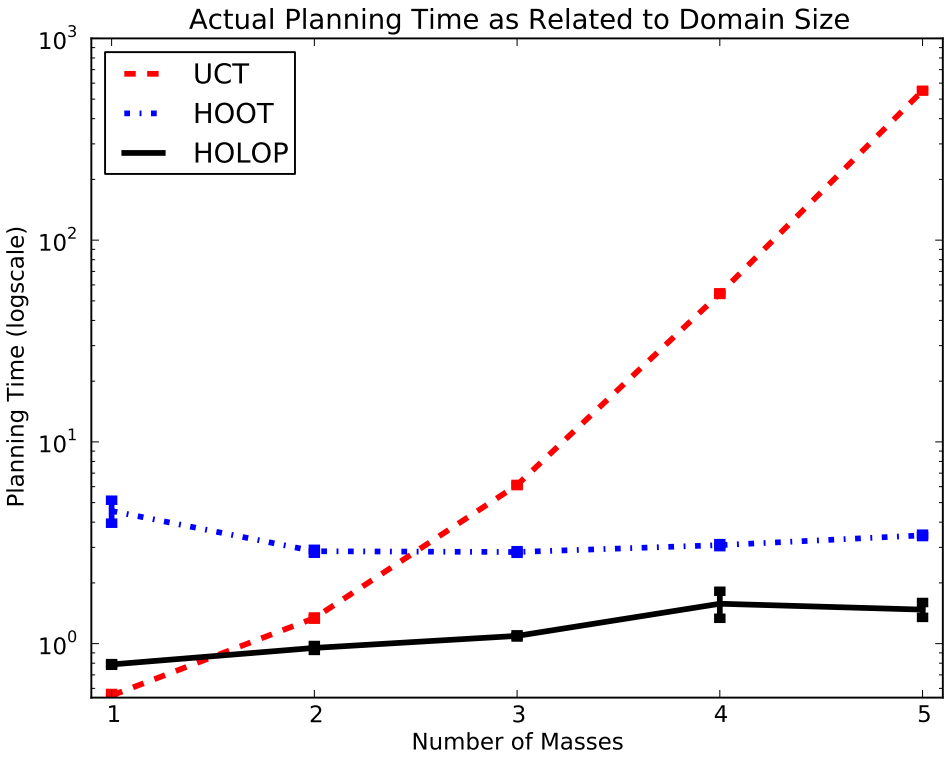


Figure 4.13: Logscale running time of UCT, HOOT, and HOLOP, in seconds in the scaling version of the double integrator domain.

for $D = 1$ than for even $D = 5$, which at first seems unusual. As discussed in Section 2.4.3, the reason this pattern appears is because in the 1-double integrator, the state space is small and trajectories are therefore significantly more likely to reencounter $\langle s, h \rangle$ pairs. When pairs are revisited, action selection must traverse deep into each tree for each $h \in H$ during action selection. As D increases, the effective state-space S' used by HOOT explodes, meaning that $\langle s, h \rangle$ pairs are revisited much less frequently, and the internal HOO trees are therefore more shallow and also take less time to traverse.

With respect to HOLOP, aside for 1-double integrator, the actual running time of HOLOP is significantly less expensive than the alternative algorithms,

which is a result of the fact that a HOO tree is only traversed once at the beginning of the rollout, and the rest of execution is performed open-loop. The slight increase in running time over increasing D is due to the extra computational costs of running D simulations instead of one for each planning step.

It is worth focusing on the fact that consistent with the motivations of local planning, state is almost a non-factor in the analyses conducted. Aside from the unavoidable $HN|S|$ costs of simply performing rollouts, they are not a factor in memory or computational costs. Even though UCT becomes untenable in high dimensional domains due to costs based on $|A'|$, there is no such direct contribution of $|S'|$ in the running time of the algorithm.

4.5 Discussion

In this chapter, HOLOP, a fully continuous state-action open-loop planner is compared to a number of state of the art discrete planning algorithms. These competitor algorithms span the spectrum from the closed-loop empirically dominant UCT, to the theoretically motivated open-loop OLOP. The unifying property of the competitor algorithms is that they all require some form of discretization to plan in continuous domains. The experiments that test the relative performance of these algorithms are set up in two main groups, examining different aspects of the performance of planners in practice. The heat map results show that, even in small domains, the performance of HOLOP is not exceeded by, and and is generally better than, the various discrete alternatives. The second set of scaling experiments shows that as domains grow in size, the impact of discretization causes the gap in performance to grow even wider, especially in the case of comparison to the fully discrete planner

UCT. In an extension to the scaling experiments, important metrics of memory and computational requirements are considered both analytically as well as empirically. Results show that HOLOP is superior to its rivals, with UCT having exponentially higher costs in terms of required samples, memory, and computation time as domain size increases. In this discussion, we will further consider the implications of each comparison.

Discussion of Heat Map Results

As the empirically most effective discrete planner, the most significant result of the heat map experiments is empirically demonstrating HOLOP's ability to outperform UCT in all domains tested. There was not a single parameterization of UCT or HOOT tested that outperforms HOLOP with statistical significance, although there are many bad parameterizations that lead to performance that is statistically significantly worse. Additionally, we have shown that two discrete planning algorithms that have excellent theoretical properties fail to be competitive in practice. Specifically, OLOP was selected both because of its analytical qualities and because it is the most directly comparable discrete planner, as both OLOP and HOLOP are regret-driven open-loop planners with the main distinction being that one requires a discretization *a priori* while the other selects a discretization adaptively. The heat map results show the importance of selecting a "good" parameterization, and indeed, the worst parameterizations for UCT resulted in performance not statistically significantly different from the empirically ineffective OLOP and FSSS-EGM. Conversely, HOLOP had absolutely no parameter changes throughout all of the experiments presented, and always produces excellent results without parameter tuning.

It is also worth focusing again on the phenomenon of the nonsmooth optimization landscape over optimiziation discussed in Section 4.3.1 and displayed in Figure 4.14, which shows how the optimization space over coarse discretizations with respect to policy quality can be highly unsmooth. This is due to the fact that, domains, discretizations, and algorithms can interact in unexpected ways, leading to behavior that is both poor, and may require domain expertise to avoid. As another concrete example of this phenomenon occurring, but this time over discretization of the action space, consider the results of sparse sampling (discussed in Section 2.3.1) planning in the double integrator, presented in Figure 4.14. A local planner that requires discrete action space, sparse sampling is well known for being extremely myopic (even more so than OLOP or FSSS-EGM). Because of this, if $0 \in A'$, $a = 0$ will always be selected (as it produces the least immediate penalty), and the object is not moved for the duration of the experiment (ultimately creating poor cumulative reward). If, on the other hand, $0 \notin A'$, the algorithm will the smallest magnitude acceleration in A' in the appropriate direction. Therefore, discretizations that have an odd number of cells (that present the 0 action) produce poor results while those that have an even number of cells produce more reasonable results (at the point where there are 30 cells, there is almost no distinction between the near-0 and 0 actions, so results from that parameterization are also poor).

In general, *any* coarse discretization is going to be suboptimal. As discussed in Section 3.1, from the perspective of regret, acting optimally according to a coarse discretization is indistinguishable from a degenerate policy. From a purely practical standpoint, in real-world domains exhibit smoothness, there are large regions that can quickly be determined to be unimportant, along with contrasting high-value areas that need to be examined with care to sample as

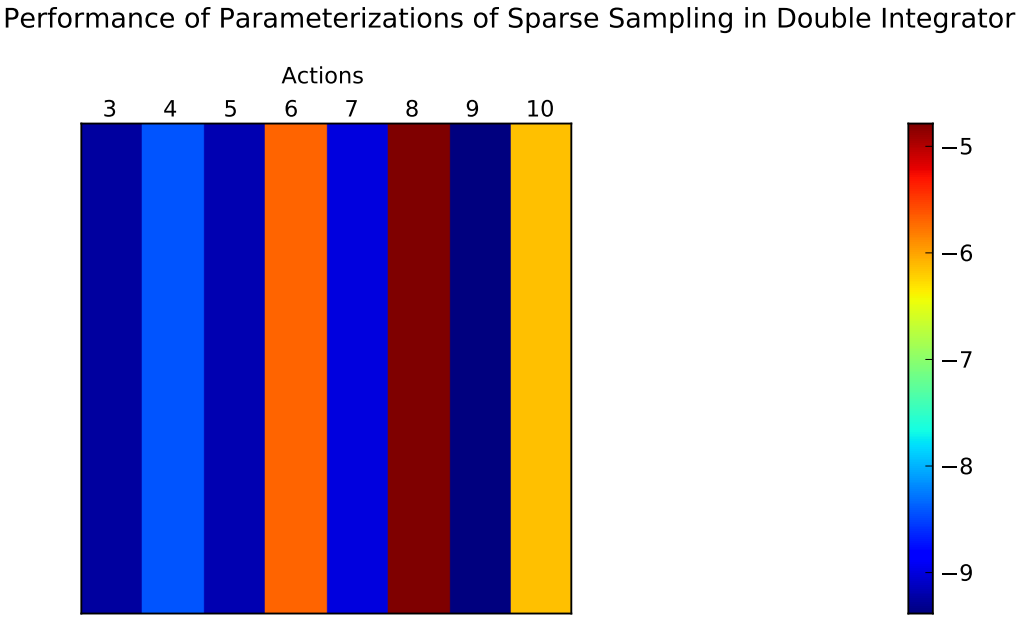


Figure 4.14: Unintuitive influence of action discretization on cumulative reward of sparse sampling in the double integrator.

closely as possible to the optimum. Methods that rely on coarse discretization fail on both counts. By allocating resources evenly, too much is expended on suboptimal regions, leaving few samples to focus on the most promising areas of the domain. HOLOP and HOOT, on the other hand, succeed in focusing samples on appropriate regions of the action space, and as a result are the most effective planning algorithms tested.

Another critical point is that coarse discretizations apply the same discretizations at all times, but there is no fixed discretization (coarse or not) that will work for all states in a domain; if one considers planning from two different states, discretizations should look different to maximize performance. For example, consider the case in the inverted pendulum where the pendulum is

about to fall either to the left (which we will call s_1), or to the right (s_2). From s_1, it is clear that the whole range of actions that moves the pendulum further to the left are suboptimal, and therefore a planning algorithm should quickly cut off about half the action space from consideration entirely. From s_2, however, the good and bad regions are exactly opposite as those from s_1, and using the same discretization found to be effective for s_1 in s_2 would lead to very poor behavior. As opposed to algorithms that require a discretization provided *a priori*, HOLOP and HOOT will perform different, appropriate discretizations over A for any provided start state.

Discussion of Scaling Results

The primary purpose of the scaling experiments is to show how coarse discretization causes discrete algorithms to suffer from the curse of dimensionality. One could argue that the results are unfair to the discrete planning algorithms due to the fact that the best discretization from the $D = 1$ domains is applied to the larger instances of the domain, where coarser discretizations may allow for increased performance, but in the largest instances where $D = 5$, even discretizing action dimensions into 5 cells results in 3125 actions in the double integrator and inverted pendulum, far greater than the allowed budget. Allowing even lower resolution cells would also not serve to improve results much as agents would be allowed only to take actions from the extremes of the action ranges, which is heavily penalized. No matter how one tries to set up a more advantageous situation for the discrete planners, it is simply not possible for such methods to be competitive in larger domains.

As has been mentioned many times, algorithms that rely on coarse discretization fail in high dimensional domains due to the explosion in the size

of $|S'|$ and $|A'|$ (for local planners, growth in $|A'|$ is particularly problematic). Given this exponential growth, almost all discrete local planners (including UCT) quickly devolve into what is commonly called vanilla Monte Carlo planning, where rollouts are performed by chance. This situation occurs because coarse discretization in high dimensional spaces leads to violation of one of the most common assumptions among discrete planners, which is that $N > |A'|$. When this assumption is violated, each action is selected from the start state at most once, and policies are otherwise selected uniformly at random. After the budget of N rollouts is exhausted, the planner simply selects the action that produced the highest return (no averaging is needed as each action initiates at most one rollout). Therefore, the results of UCT in the scaling problems would be consistent with the performance a large class of discrete local planners in that setting.

Discussion of Cost Results

Analytical results show that UCT must maintain an exponentially increasing amount of data in D, $|A'|$. Adding insult to injury, results presented in Section 2.4.3 demonstrate that as domains grow in size, the rate at which $\langle s, h \rangle$ are revisited decreases rapidly. In such a situation, the statistics maintained by UCT serve to increase computational requirements substantially while also not helping with policy construction, as action selection is not meaningful until each the number of visits to any $\langle s, h \rangle > |A'|$. Although UCT must maintain $O(HN|A'|)$ items in memory, this data does not become of use until N greatly exceeds some combination of $|S'|$ and $|A'|$. Therefore, the algorithm is left with enormous memory costs that do not even serve to influence decision making.

With regards to HOOT, after parameter optimization, HOOT generally produces policies of quality equivalent to HOLOP. On the other hand, while the quality of these results are not statistically significantly better than HOLOP, the computational and memory costs of HOOT are larger than HOLOP by a factor of H. That is, while HOLOP only inserts and queries for an action sequence at the beginning of each rollout, HOOT must insert and query for an action at each individual step in a rollout, with each query being potentially as expensive as that made by HOLOP. In summary, HOOT is more expensive by a factor of H while also requiring a parameter search over σ to perform well.

In Closing of Comparison

In this chapter, algorithms from a number of different areas and backgrounds, intended for use in different settings, were tested. Some were selected because of strong analytical bounds, while others due to strong empirical support in the literature. Others yet were selected to cover different aspects of planning, such as closed- versus open-loop, or different aspects of discrete versus continuous planning.

FSSS-EGM and OLOP in particular, have strong guarantees of performance in worst-case settings. On the other hand, it should be noted that guarantees do not prove optimal performance. In the case of OLOP, its bounds are not equivalent to the best bounds possible. (No currently existent algorithm achieves those theoretically optimal bounds in all settings.) FSSS, on the other hand, simply guarantees that unlike UCT, it will not take super-exponential time to find the optimal policy, and that its estimates of action quality are PAC. Regardless, from the perspective of regret,S attempting to behave optimally according to a discretization is meaningless (Section 3.1).

FSSS-EGM and OLOP are not effective in the domains considered, and likewise do not have support in the literature of being effective planners in practice. UCT, on the other hand, has extremely poor worst-case performance, but is known to be effective in many real-world problems. Considering the results here, a conclusion that can be drawn is that based on prior state of the art for discrete planning, it is possible to select either an algorithm that has strong theoretical guarantees *or* optimize for real-world performance, but not both. Because this chapter primarily focuses on empirical results, the focus has been on the empirically dominant UCT.

Regardless, UCT was still shown to be ineffective in comparison to HOLOP. Although it is known to be suboptimal with respect to simple regret, HOLOP still has formal guarantees both in terms of standard and simple regret, which is not true of UCT or HOOT in the general case. Based on these results, we have a guarantee that is fairly similar to that of FSSS, which is that in the worst case it will behave as badly as uniform planning in terms of simple regret, but not worse. The distinction is that in addition to the formal theoretical results surrounding HOLOP, it also is superior in practice, as none of the algorithms outperform it in this chapter. As such, HOLOP is a planner that allows for what discrete planning algorithms currently do not provide, which is both strong theoretical guarantees as well as strong results in practice.

It is worth noting, however, that the goal of this chapter is not to show that HOLOP is the best continuous planner. The literature on fully continuous planners in general MDPs is still quite underdeveloped, and there is room for improvement both in terms of analytical and empirical results. The goal is instead to show that even an initial attempt into the field of native continuous planners (which is known in some ways to be suboptimal) can still easily

outperform the best discrete planners that are known to exist both in terms of analytical regret as well as empirical performance.

Chapter 5
Scalable Continuous Planning

In Chapter 4, it was demonstrated that, as compared to a wide range of discrete planners, HOLOP has lower sample complexity, produces higher quality solutions, and is more robust. These properties result from the fact that it does not require search over a discretization to produce effective policies, and more effectively uses acquired data. HOLOP is a natural baseline continuous planner by which to test other algorithms, as it has a number of desirable properties such as tight performance guarantees, a built-in exploration policy, good real-world performance, and parameter-free operation. Additionally, it was shown that HOLOP scales better than the discrete planners tested, which spanned a number of categories.

While it has already been made clear that discrete methods are unusable in even the medium-sized domains of Chapter 4, in this chapter, we consider planning in large domains, such as humanoid locomotion tasks that have up to 18 state dimensions and 7 action dimensions. When working in large domains where domain expertise is minimal, many trajectories are required to conduct planning. In this case, the linear memory requirements and super-linear computational requirements of HOLOP that were previously acceptable causes planning to be prohibitively expensive. Therefore, in this chapter, we restrict consideration to methods of planning that have planning and memory costs at most linear in N, and that are easily parallelizable in order to take

advantage of modern hardware. Algorithms in this chapter follow a model similar to HOLOP, where a continuous optimization algorithm is used to optimize a sequence of actions with respect to return, building on the insights gathered in Chapter 4.

5.1 Cross-Entropy Optimization

Originally designed for rare event simulation (Rubinstein, 1997), the cross-entropy method (CE) was extended to perform global optimization by casting high value points as the rare event of interest (Rubinstein, 1999). While the algorithm can be described generally at the cost of simplicity (Boer et al., 2005), we focus on the specific version described in Algorithm 12.

Briefly, the algorithm functions iteratively by: sampling a set of I actions $a_1 ... a_I$ from the distribution p based on p's current parameterization Φ_{g-1} (line 3); assigning rewards (or returns) to $r_1 ... r_I$ to $a_1 ... a_I$ according to the (potentially stochastic) evaluation function f (line 4); selecting the ρ fraction of "elite" samples (lines 5 and 8); and then computing the new parameterization of p $\hat{\Phi}_g$ used in the next iteration based on the elite samples (line 8). This occurs for Γ generations, with the total number of samples taken being $N = \Gamma I$.

The algorithm requires several items to be specified *a priori*. First among them is the type of distribution p, which defines main characteristics of how a is drawn. Although p can be any distribution, in this chapter, unless otherwise specified, we assume p is a Gaussian. Ideally, $p(\cdot|\hat{\Phi}_0)$ would be the distribution that generates optimal samples in the domain (although, if that were known, no optimization would be necessary). Since, generally, this distribution is not

Algorithm 12 Cross-Entropy

1: **function** OPTIMIZE($p, \hat{\Phi}_0, R, \rho, I, \Gamma$)
2: **for** $g = 1 \rightarrow \Gamma$ **do**
3: $a_1 ... a_I \sim p(\cdot|\hat{\Phi}_{g-1})$
4: $r_1 ... r_I \leftarrow R(a_1) ... R(a_I)$
5: sort actions according to descending reward
6: $\mu_g = \frac{\sum_{i=1}^{\lceil I\rho \rceil} a_i}{\lceil I\rho \rceil}$
7: $\sigma_g^2 = \frac{\sum_{i=1}^{\lceil n\rho \rceil} (a_i - \mu_g)^T (a_i - \mu_g)}{\lceil I\rho \rceil}$
8: $\hat{\Phi}_g = \langle \mu_g, \sigma_g^2 \rangle$
return a_1

known, or is difficult to sample from, other distributions are used. When domain expertise is limited, which is the case we consider here (Assumption 4), it should be ensured that $p(\hat{\Phi}_0)$ has good support over the entire sample space. Doing so helps to ensure that some samples will fall near the global optimum in the first generation. The update rule for the parameter vector $\hat{\Phi}_g$ in line 8 is defined as the maximum likelihood estimate for producing the elite samples in the current generation (although other rules can be used as well).

There are a number of other parameters for CE. The parameter ρ determines the proportion of samples that are selected as elite, and is important because it impacts the rate at which $\hat{\Phi}_g$ changes from generation to generation, as well as the final solution Goschin, Littman, and Ackley (2011). The variable I defines the number of individuals per generation; it is important that this number be large enough so that the early generations have a good chance of sampling near the optimum. The number of generations evaluated by the algorithm is defined by Γ.

These parameters must all be set sensibly to find a good solution for the domain of interest. While doing so, tradeoffs in solution quality, computational requirements, convergence rates, and robustness must all be considered. To

simplify the application of CE to various domains, there are methods that automatically adjust (or remove) these parameters. While we use a fixed number of generations in optimization, another common method is to examine the stability of $\hat{\Phi}$ or $r_1, ..., r_I$ from generation to generation, and terminate when the change of the variable drops below a threshold. The fully automated cross-entropy method (Boer et al., 2005) further reduces the need to manually define parameters to CE by adjusting them automatically during execution.

CE has a number of beneficial properties. By attempting to perform global optimization, it avoids getting trapped in local optima; if attempting to find near-optimal solutions, local search methods require shaping functions to initialize search near the global optimum, making them inapplicable in our setting as shaping functions require domain expertise. Another property of CE is that its computational costs are linear in the number of samples, and the method parallelizes trivially within a generation, meaning the optimization itself is not computationally intensive.

While CE has guaranteed convergence to optimal results in some discrete (Costa et al., 2007) and continuous domains (Margolin, 2005), the conditions required in existing proofs are fairly strong and are violated in the experiments discussed here. To our knowledge, there are no guarantees in terms of the rate of convergence or sample complexity.

As compared to HOLOP, there are two primary drawbacks involved with using CE. Firstly, HOLOP is a parameter-free planning algorithm. CE, on the

other hand has a wide range of parameters, and the algorithm is generally sensitive to changes in their values. Additionally, extra modifications are sometimes needed in practice, such as a temperature schedule that prevents premature convergence to local optima (Szita and Lörincz, 2006). Therefore in practice, whereas HOLOP can simply be applied to a problem with no additional work, parameter optimization and other modifications are a necessary component of producing quality results in CE. Secondly, whereas HOLOP has strong formal guarantees, the lack of such guarantees for CE in the setting considered in this chapter is a item worth considering.

5.1.1 Cross-Entropy Optimizes for Quantiles

When using CE, an item to keep in mind is that instead of optimizing for expectation, it optimizes for quantiles (Goschin, Littman, and Weinstein, 2013). The formal proof is related to one that was produced showing the same behavior in genetic algorithms (Goschin et al., 2011), but will not be covered here. Instead, this section presents an intuitive argument as well as a concrete case study outlining when optimizing for quantiles can cause poor behavior in practice and how to correct this behavior when using CE.

In standard CE, an all-or-nothing approach is used to define elite samples (defining the top ρ-fraction as elite and discarding the rest). Doing so means all data of non-elite samples are simply ignored when constructing $\hat{\Phi}_g$. Consider a 2-armed bandit, with $R(a_1)$ producing 1 with probability 0.2, and -1 with probability 0.8, and $R(a_2)$ producing 0.5 with probability 1. In this case, of course $\mathbb{E}[R(a_1)] = -0.6$, and $\mathbb{E}[R(a_2)] = 0.5$. CE can be used for this bandit task by defining a Bernoulli distribution p, selecting either a_1 or a_2 (when

$a \sim p = 0$ or 1), with $\hat{\Phi}_0 = 0.5$, initialized to chance selection. Assume additionally that $\rho = 0.1$ and that I is some large value. In the first generation, the samples where $R(a_1) = 1$ will constitute the elite samples, and thereafter $\hat{\Phi} = 0$, sampling only from a_1, even though it is poorer in expectation.

The change that causes CE to maximize expectation instead of quantiles is simple, and is accomplished by weighing each sample proportionally to its value, as opposed to the standard threshold (0 or 1 weight) (Goschin, Littman, and Weinstein, 2013). We call this simple variant of the algorithm CE-Proportional (CEP). This modification has the added benefit of simplifying the algorithm by removing the parameter ρ. CE used in this manner can be related to a broader family of optimization algorithms (Stulp and Sigaud, 2012).

In a larger, more real world example of this phenomenon, we discuss two variants of the game of blackjack and describe how differences in mechanics can lead to changes in policies when optimizing for quantiles or expectation. The first variant of the game reduces the game to its most important dynamics, as described in Sutton and Barto (1998). At the start of play, the dealer is dealt one visible card, and the player is dealt two cards from a infinitely large shoe made of standard decks. At each point in time, the player can choose to either *hit* (add another card to his hand), or *stick* (cease to add cards and pass play to the dealer). If the sum of the player's hand surpasses 21 points, he *busts*, losing the round. Once the player sticks, the dealer hits until his hand sums to 17 or more points.

Likewise, the dealer busts if his hand sum reaches greater than 21 points. Assuming neither busts, the winner is the player with the hand closest to 21 points (if both hands are equal valued, play ends in a draw). Each numeric card has points equal to its number, with face cards having a value of 10. The

ace can be valued at 1 or 11 points. If the ace can be valued at 11 points without busting, it is scored as such and is said to be *usable*. We also adopt the policy representation of Sutton and Barto (1998). The state is represented by the dealer's showing card, the sum of the player's hand, and whether or not the player holds a usable ace. On hand values less than 12, the player automatically hits, because there is no chance of busting. Therefore, the game can be represented with 200 states with 2 actions of hitting or sticking (the distribution over which is binomial). CE begins with a uniform distribution over all pure policies, represented by 200 individual binomial distributions initialized with $\hat{\Phi}_0 = 0.5$. As such, each sample from the space optimized over by CE is a point in policy space of size 2^{200}.

Using CE in this manner is equivalent to classical policy search. Although in this chapter we focus on CE for performing policy search over open-loop policies, the algorithm does have a history of use in standard policy search. CE has had significant empirical success in a number of settings, among them buffer allocation (Alon et al., 2005), scheduling, and vehicle routing. More references and applications are described in the standard CE tutorial (Boer et al., 2005). The first paper to apply the CE method formally in the context of RL for policy search was Mannor et al. (2005). The idea of using CE to search in a parameterized policy space was subsequently used to obtain results that were orders of magnitude better than previous approaches in the challenging RL domain of Tetris (Szita and Lörincz, 2006, Szita and Szepesvári, 2010, Goschin, Littman, and Weinstein, 2013).

Returning to blackjack, the experiment is run with $G = 2000$, and $I = 10000$. Each experiment is repeated 10 times. Figure 5.1(a) shows the average reward per generation over each of the 10 executions of CE with various

selection methods. As can be seen, policy improvement occurs most rapidly with $\rho = 0.5$, but levels off quite rapidly. It is then surpassed by CEP, which produces the highest quality policies for the rest of the experiment. The distribution of rewards according to strategy is depicted in Figure 5.2(a), with error bars displaying the standard deviation of the average of the 10 final populations in each experiment. While CEP produces the best policy, the difference between it and standard CE is minimal.

In the second variant tested, the option to *double* is introduced. This action causes the player to double the wager (after which payoffs can be only $-2, 0$, or 2), hit, and then stick. All other details (with changes in resulting state and action spaces) are identical to the first setting, and the dealer is not able to apply this action. The performance of the various CE variants is rendered in Figure 5.1(b). Whereas in blackjack without the *double* option all parameterizations of CE and CEP improved over time, once the double option is presented, only CEP results in consistent improvement over time. Both CE with $\rho = 0.2$ and $\rho = 0.5$ initially improved but later degraded, with $\rho = 0.5$ being essentially equal to chance performance by the end of the experiment, and all other policies produced by non-proportional selection being worse than chance. The reason for the difference in average performance is because without proportional selection, CE maximizes for quantiles, and therefore prefers the double action as it occasionally produces higher reward even though it is worse on average. As can be seen in the distributions over rewards in Fig. 5.2(b), CEP exercises the double action less than 10% of the time, and has an action distribution markedly different from the other strategies. In particular, CE with $\rho = 0.1, 0.2$ both performed the worst, and doubled the most (almost 95% of the time), and lost almost $1/3$ of all bets where doubling was used, resulting in

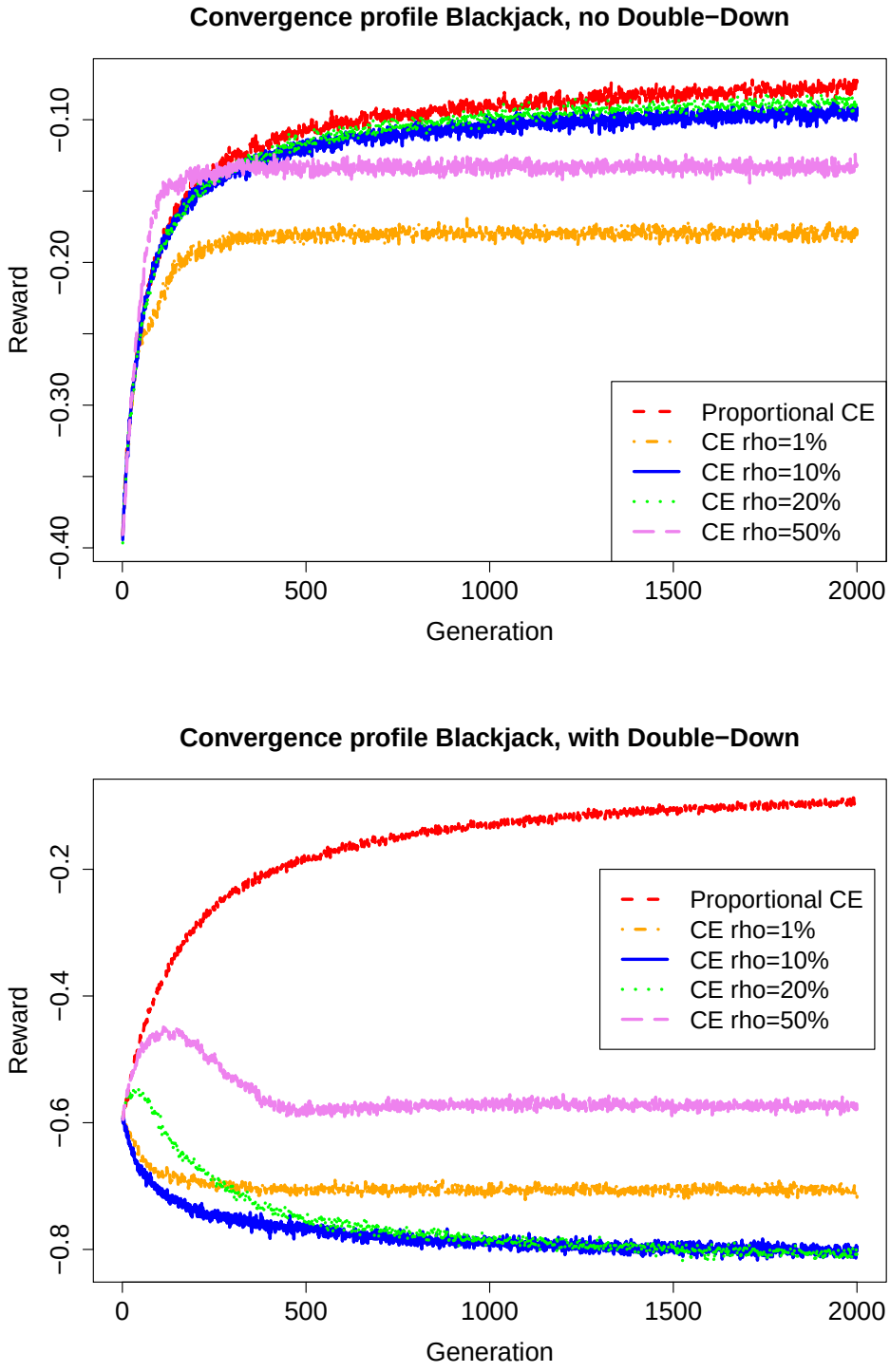


Figure 5.1: Average payoff of CE with various ρ and CEP.

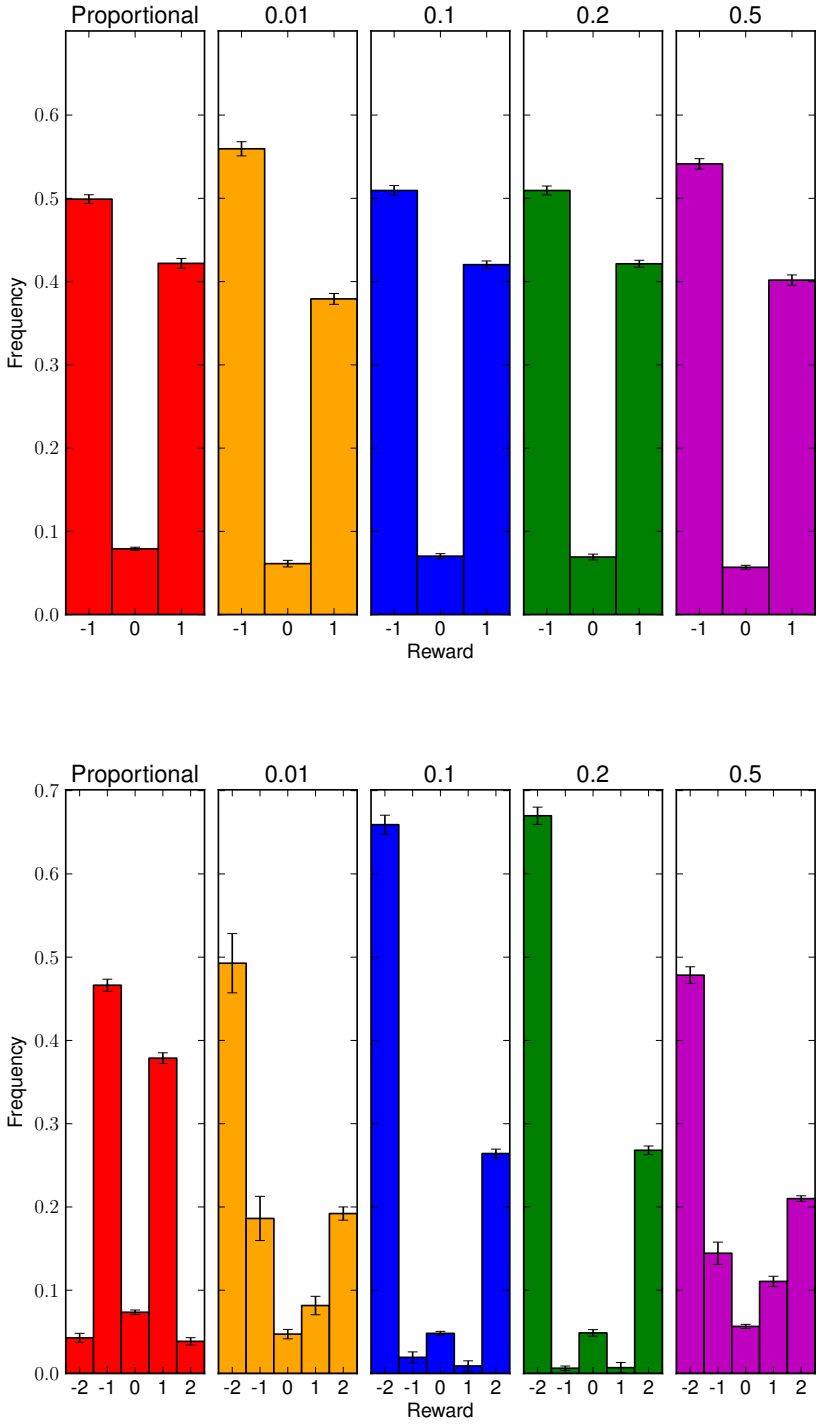


Figure 5.2: Payoff distributions of CE with various ρ and CEP.

very poor performance. Essentially, the CEP variant both removes a parameter that must otherwise be searched over, while improving the performance of CE drastically in cases where optimizing for quantiles produces a different result than optimizing for expectation.

5.2 Open-Loop Planning with Cross-Entropy

In this section, we discuss the application of CE to open-loop planning. Essentially the approach is the same as HOLOP, with the distinction that CE replaces HOO for optimization. This planning algorithm will be called cross-entropy open-loop planning (CEOLP). As compared to HOLOP, the advantages of CEOLP are memory costs independent of N and planning times only linear in N. As an open-loop planner underpinned by a global optimization algorithm, CEOLP shares properties with HOLOP, such as having planning costs are independent of $|S|$. Additionally, it operates identically in continuous, hybrid, and partially observable domains. This final property is extremely important when considering the humanoid walking problems that will be presented later in this chapter, as physical domains with hard contacts (such as feet contacting the floor) make the state space a discrete-continuous hybrid, significantly complicating planning in closed-loop methods. CEOLP is shown in Algorithm 13.

Applying CE to local planning does not require any changes to the algorithm itself; only the manner it is used changes. The evaluation function f now performs a complete rollout according to the action sequence and evaluates the sequence according to its return (lines 4, 9-16); parameters necessary for the rollout are bound by the agent, and f is left to accept $\mathbf{a}$ as a parameter during optimization in CE (line 4). For the fixed planning horizon H, optimization is now conducted over length-H action sequences (vectors of length $|A|H$)

Algorithm 13 Open-Loop Planning with CE

1: **function** AGENT($G, H, s_0, p, \hat{\Phi}_0, \rho, I, \Gamma$)
2: $s \leftarrow s_0$
3: **loop**
4: $f \leftarrow \lambda \mathbf{a}$. ROLLOUT($\mathbf{a}, s, 0, H, G$)
5: $\mathbf{a} \leftarrow$ OPTIMIZE($p, \hat{\Phi}_0, f, \rho, I, \Gamma$)
6: $a' \leftarrow \mathbf{a}_1 ... \mathbf{a}_{|A|}$
7: $s' \sim G_{T(s,a')}$
8: $s \leftarrow s'$
9: **function** ROLLOUT($\mathbf{a}, s, h, H, G$)
10: **if** $h \geq H$ **then**
11: **return** EVALUATE(s)
12: $R \leftarrow G_R, T \leftarrow G_T, A \leftarrow G_A, \gamma \leftarrow G_\gamma$
13: $a' \leftarrow \mathbf{a}_{h|A|} ... \mathbf{a}_{(h+1)|A|}$
14: $r \leftarrow R(s, a')$
15: $s' \sim T(s, a')$
16: **return** $r + \gamma$ ROLLOUT($s', \mathbf{a}, h + 1, H, \Gamma$)

with respect to their returns (line 5). After optimization is complete, the first action in the sequence (line 6) is performed in the true domain (line 7), and the process repeats anew from the resulting state (line 8).

Cross-Entropy has been used for local planning in MDPs to search over sequences of states in domains as complicated as simulated helicopter control and navigation (Kobilarov, 2011). Additionally, CE has been been used in place of a uniform distribution to improve the performance of rapidly exploring random trees (Kobilarov, 2012). This previous work, however, is concerned with domains that are deterministic, and planners that have access to more domain knowledge than we assume (Assumption 4), such as inverse kinematics and precomputed motion primitives.

5.3 Empirical Results

This section presents a detailed comparison of performance between HOLOP and CEOLP, using benchmark domains from Chapter 4. It should be noted that the goal of this section is not to make the claim that CEOLP is a better algorithm than HOLOP, as it is somewhat unfair to compare them. While it will be shown that CEOLP does outperform HOLOP in terms of policy quality, memory usage, and computation time, the differences in assumptions made by the two algorithms must be kept in mind. Theoretical results (which CEOLP essentially does not have) aside, HOLOP is an entirely parameter free algorithm, while CEOLP requires the following parameters specified:p, $\hat{\Phi}_0$, I, Γ, and (if CEP is not used) ρ. The argument that will be made, however, is that properties of HOLOP make the algorithm more difficult to use when dealing with large planning domains, and in such settings other algorithms such as CEOLP are more practical.

In the second set of experimental results, CEOLP is then selected for planning in a number of large domains involving humanoid locomotion. These domain have $|S| = 18$ and $|A| = 7$, and are difficult planning problems for a number of reasons. Although characteristics and difficulties involved in the domain will be discussed at length, it is high dimensional domain with sparse effective policies, the space of which has many discontinuities with respect to return (Tassa and Todorov, 2010, Erez, 2011, Erez et al., 2011, Tassa et al., 2012).

5.3.1 Double Integrator

In this section, performance of CEOLP is presented in the double integrator domain, introduced in Section 4.2.1. In the first part of the section, performance and costs are compared to HOLOP, and the second set of results shows a detailed comparison to optimal performance by a linear quadratic regulator (Sontag, 1998).

CEOLP and HOLOP

In the first set of empirical results with CEOLP, performance is compared to HOLOP. Whereas Chapter 4 focused primarily on the ability of algorithms to scale to high-dimensional domains, in this section we consider results from that setting as well as differences in planning costs as N increases. In CEOLP, N is divided into 10 evenly sized generations, with $\rho = 1/4$, and p is a distribution over $H = 50$ independent Gaussians.

Figure 5.3 presents scaling results for HOLOP and CEOLP, which are equivalent to those presented in Section 4.3.2. As can be seen, although the algorithms have essentially identical performance when $D = 1$, increasing D causes a great deal more performance degradation for HOLOP than CEOLP. In fact, the performance of CEOLP at $D = 5$ is not statistically significantly different from that of HOLOP at $D = 2$. These results are essentially the same in the inverted pendulum, although in that domain CEOLP at $D = 5$ outperforms HOLOP at $D = 2$ with statistical significance (not shown).

A reason for the disparity in performance of the two algorithms is that the impact of new data is smaller in HOLOP than in CEOLP. Whereas CEOLP updates the distribution over all dimensions over the entire sample space after each generation, in HOLOP each new sample can only refine the policy in one

region of one dimension. Therefore, in practice refining the policy requires more samples for HOLOP compared to CEOLP.

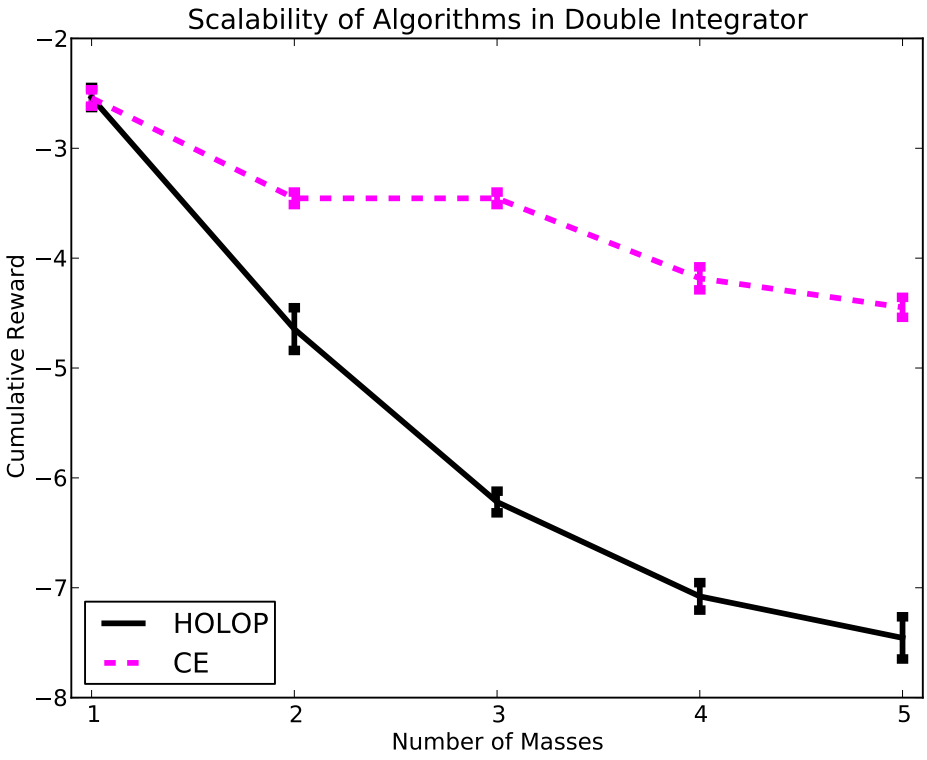


Figure 5.3: Performance of planning by HOLOP and CEOLP in the double integrator.

In the next series of results, we compare the costs of planning as N increases. Cost of both algorithms are effectively invariant to changing D, so results for fixed D are not presented. Figure 5.4 shows the change in memory use as N increases. As expected, the memory use of HOLOP is linear in N, as each trajectory results in one additional constant-size node being stored in the HOO tree decomposing the space of action sequences. CEOLP, on the other hand, has constant memory memory costs. This property is important, and holds because between generations the only variables the algorithm must maintain are

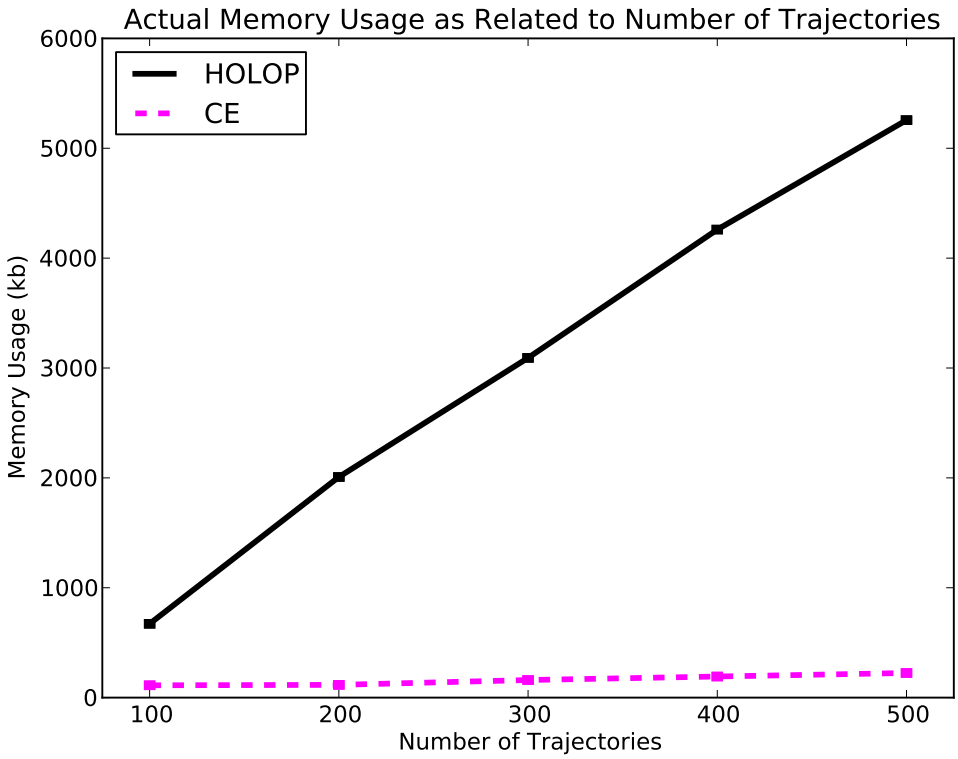


Figure 5.4: Comparison of memory usage of HOLOP and CEOLP as the number of trajectories increases.

the current generation g and $\hat{\Phi}_g$, which in this setting has a memory requirement linear in $|A|H$ (or $|A|H^2$ if multivariate Gaussians are used). Even within a generation, only $a_1, ..., a_I$ and $r_1, ..., r_I$ are required, which is slightly larger, but also invariant to changes in N.

A comparison of actual running time for one step of planning in each algorithm is presented in Figure 5.5. Again, consistent with analytical results, the running time of HOLOP is super-linear in N. This point, compounded with the degraded performance shown in Figure 5.3 as D increases means that the number of samples required and resulting planning time in large domains becomes prohibitively expensive. CEOLP, on the other hand, has running times that are exactly linear in N, and more efficiently uses samples to refine plans in

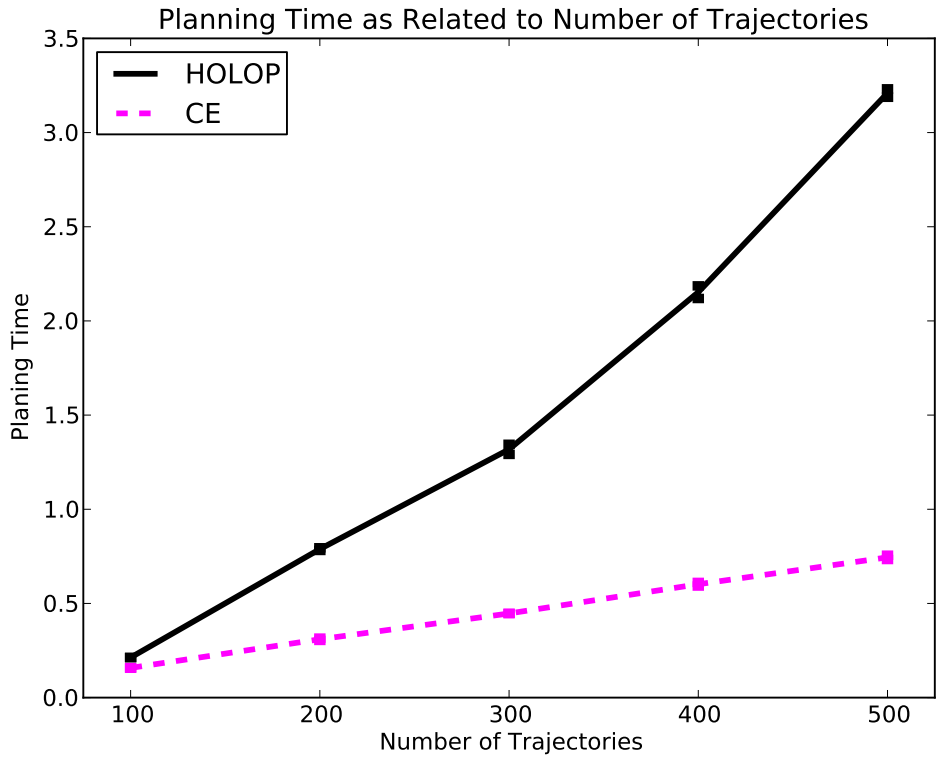


Figure 5.5: Comparison of actual running time of HOLOP and CEOLP as the number of trajectories increases.

large domains.

CEOLP and Optimal Performance

In this section, a comparison between the performance of CEOLP and optimal performance by a linear quadratic regulator (LQR) in the double integrator is conducted. The cumulative reward per episode obtained by CE for varying numbers of trajectories (ΓI) per planning step, as well as the LQR solution appear in Figure 5.6. Here, $\rho = 0.1$ and $\Gamma = 30$, with I divided evenly across generations. The mean cumulative reward of CE with $\Gamma I = 7,000$ is not statistically significantly different from optimal ($p < 0.05$). In fact, in some episodes the solution found by CE is slightly better than the "optimal" LQR solution.

Such a result can occur because the LQR solution assumes the domain is time continuous, which is not actually the case.[1]

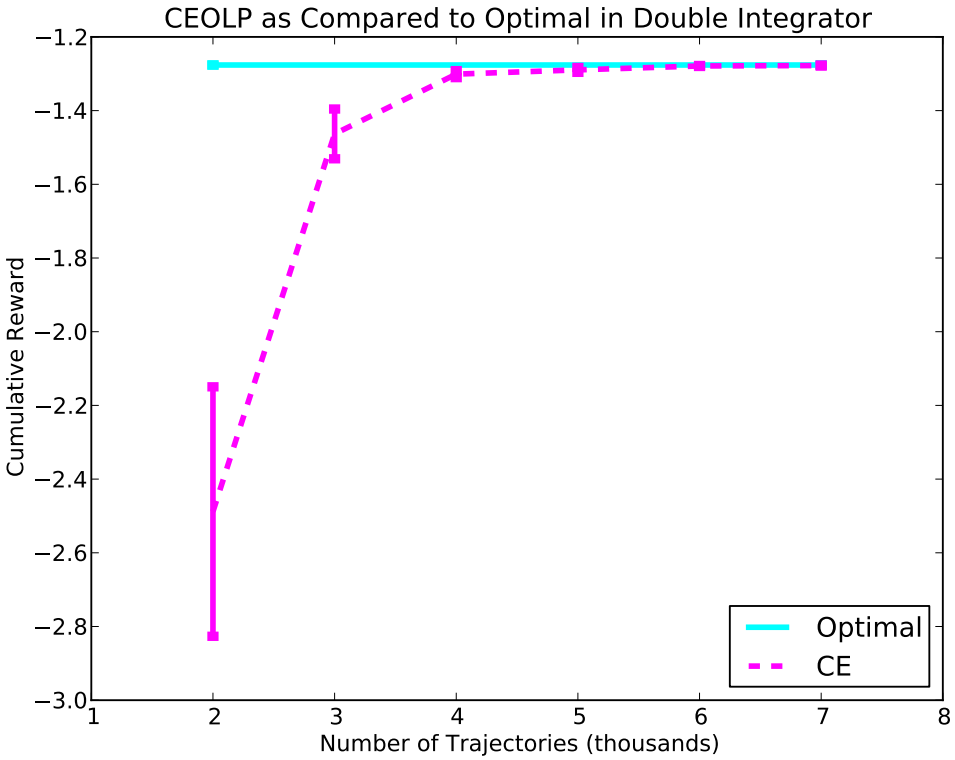


Figure 5.6: Performance of planning in the double integrator with cross-entropy compared to optimal.

A visualization of the planning performed by CEOLP in the double integrator from s_0 is rendered in Figure 5.7. Alternating in shades of red and blue, trajectories are grouped according to increasing generation during optimization, in planes of increasing height along the vertical axis. These planes correspond to the trajectories developed in generations 0, 7, 14, 21, and 28. The policy at generation 0 (lowest on the vertical axis) is simply a random walk in the domain. The trajectory rendered highest along the vertical axis in black is

[1] An optimal finite-horizon discrete-time solution is possible, but not tested (Chow, 1975).

the trajectory selected by LQR, and the start state s_0 is represented by the vertical cyan line. As can be seen, the basic shape of the policy forms quickly, with the later generations performing minute refinements.

It is worth mentioning that the trajectories from generation 28 and the LQR policy (2^{nd} to highest, and highest, respectively, in the plot) are initially similar, but differ later in the trajectories. While differences from the LQR policy should be regarded as errors, CE can still achieve very high performance due to the fact that it uses receding-horizon planning. Because planning is completely restarted during each step in the episode, the planner only needs to produce a near-optimal *initial* action each time planning is conducted.

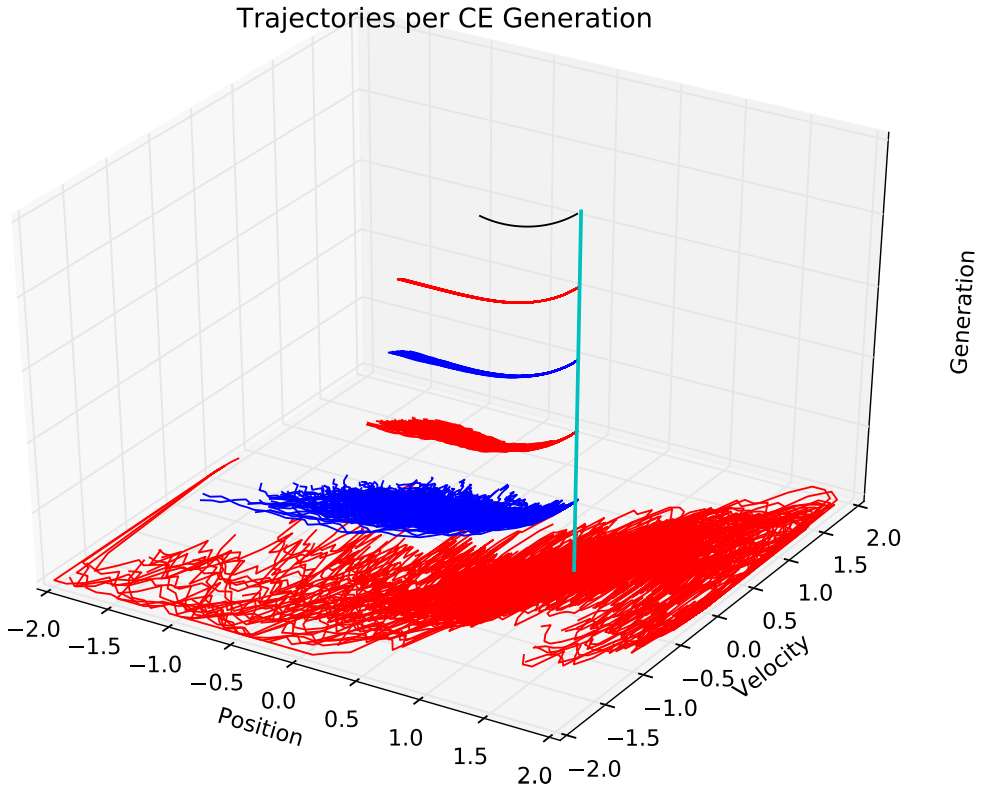


Figure 5.7: Trajectories from selected generations in cross-entropy applied to the double integrator from the episode start state.

5.3.2 Humanoid Locomotion

In this series of experiments, we demonstrate the ability of open-loop planning by CEOLP to produce effective strategies for humanoid locomotion in high-dimensional domains that are deterministic, stochastic, or even contain a flawed generative model.

As modeled here, the walker's planar body consists of a pair of legs, a pair of arms, and a torso. The legs have upper and lower segments, while the arms and torso are made of one segment. At each of the 7 joints, a desired angular joint speed is set at each time step, so the action space for planning is these 7 values. In a compact representation, the state is 18 dimensions, as one part of the body must be specified in terms of location and velocity, and all other points can be described in terms of its angle and angular velocity from the joint it connects to. The reward function is the velocity of the hip along the x-axis. Any state where a part of the body other than a foot contacts the floor is terminal, with a large penalty. We use $\rho = 0.2$ and CE is allowed 10,000 trajectories of planning per time step distributed evenly across 30 generations, with $I = 30$. Aside from parallelization, the simulation and planning algorithms were not optimized, with the simulation run in PyODE (PyODE, 2010). The sampling distribution p is a multivariate Gaussian.

In the simplest experiment, called the deterministic walker, we demonstrate the ability of CE to find an effective policy when the domain is deterministic. A stroboscopic rendering of the policy is shown in Figure 5.8, with locomotion going from right to left. It is worth noting that the gait is highly dynamic; there are entire classes of planning devoted to legged locomotion that are unable to produce this type of behavior, such as classical zero moment

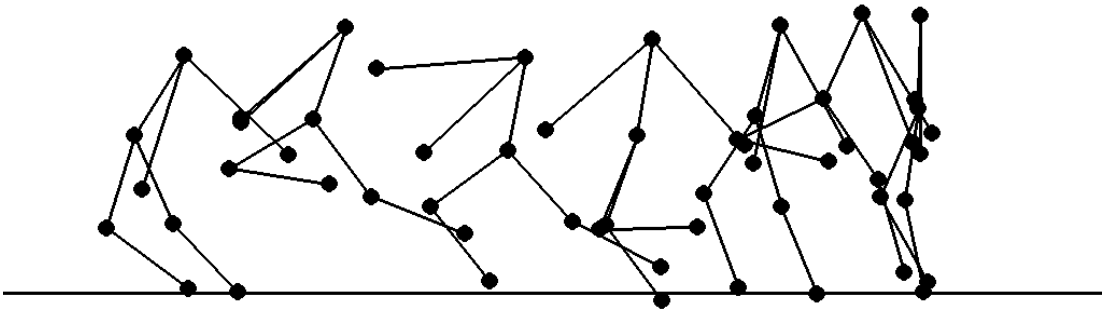


Figure 5.8: The deterministic walker, renderings from every 10^{th} frame.

point controllers. As compared to such controllers, those that can move dynamically both appear more natural and are more efficient (Manchester et al., 2011). Because there are phases where the body is ballistic, it is more accurate to call the form of locomotion running, as opposed to walking. The increasing distance between each successive rendering from right to left indicates that the agent is accelerating throughout the experiment, increasing reward per time step as the episode progresses. In contrast, the policy constructed by UCT in the same domain is rendered in Figure 5.9. The last frame with the body rendered in red is the last step in the episode, as a part of the body has contacted the ground. In this domain, UCT with $\alpha = 5$ has to consider $78,125$ different actions, much more than what is allowed by the budget of $N = 10,000$. Failure of UCT in this large domain is consistent with results in Chapter 4.

In the next experiment, called the stochastic walker, the basic setting is expanded by introducing stochasticity in the form of random "pushes". These pushes are applied to one joint selected uniformly at random at each time step, are of a uniformly random magnitude, and are always toward the right (opposite the direction of desired locomotion). This type of noise makes planning

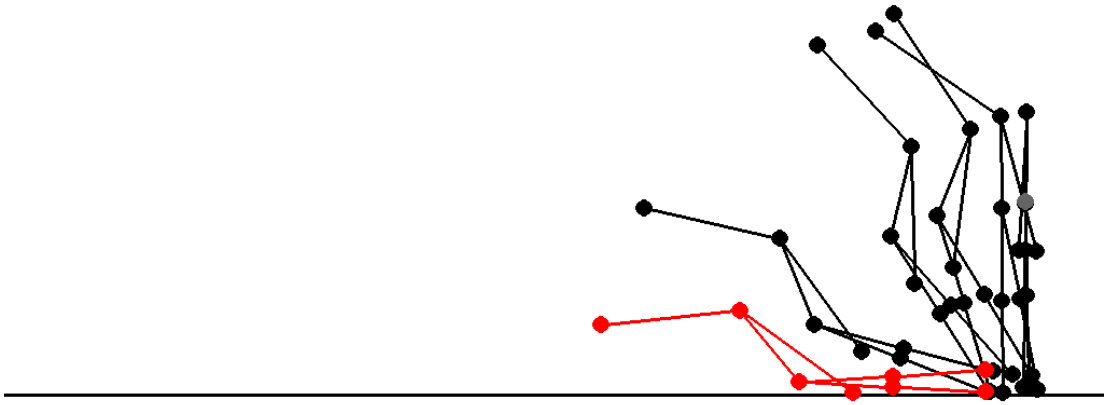


Figure 5.9: Performance of UCT in the deterministic walker, renderings from every 10^{th} frame.

difficult because it is not zero-mean, and perturbs an individual joint strongly at each time step (as opposed to an equivalent force being distributed evenly over the body). Figure 5.10 is a rendering of the performance of CE planning in this noisy domain. As compared to the deterministic walker, locomotion takes roughly 3.5 times as long to cover the same distance. Because of the difficulties associated with balance in the presence of external forces, methods that explicitly compensate in simpler settings (more constraints on the type of forces, balance of a lone leg) have been proposed, and require precomputing policies just to maintain balance (Liu and Atkeson, 2009). In contrast, the method of planning here does not rely on precomputation, but is still able to produce full-body strategies for walking (as opposed to simply balancing) that are robust to these challenging forms of noise.

Another experiment in this domain is to test the importance of replanning. In a modification of the planning scenario, the entire planning sequence of 30 steps is used before replanning is conducted again (as opposed to replanning at every step in the other experiments). In order to keep the amount of data

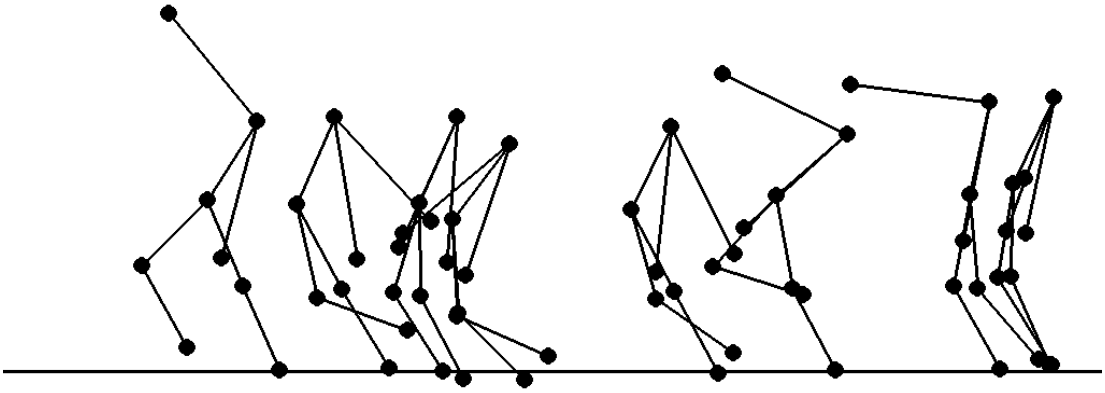


Figure 5.10: The stochastic walker, renderings from every 30^{th} frame.

available to the algorithm constant, CEOLP is given access to $N = 30 * 10000$ trajectories. The stroboscopic results of this experimental setting are presented in Figure 5.11, with renderings from every 10^{th} frame. When planning is conducted in this manner, by the end of execution of the the first 30 steps, the body is already in a configuration that cannot be recovered from, leading to an inevitable fall. The terminal state is reached somewhere around step 60, either just before the end of the second action sequence (steps 30-59), or the beginning of the third (steps 60-89), depending on stochasticity. These results show that replanning is necessary in stochastic domains, as action sequences that maximize expected reward are in general highly suboptimal once a long period of time has passed and stochasticity begins to strongly influence the trajectory.

In the final, most difficult walking experiments, we demonstrate the ability of CE to plan in settings where the generative model used for planning is erroneous. In one of the experiments, the domain itself is deterministic (without any external forces), but the model incorrectly introduces stochastic pushes (Figure 5.12). In the other experiment, the situation is reversed, where the domain is deterministic, but the model incorrectly includes stochastic pushes

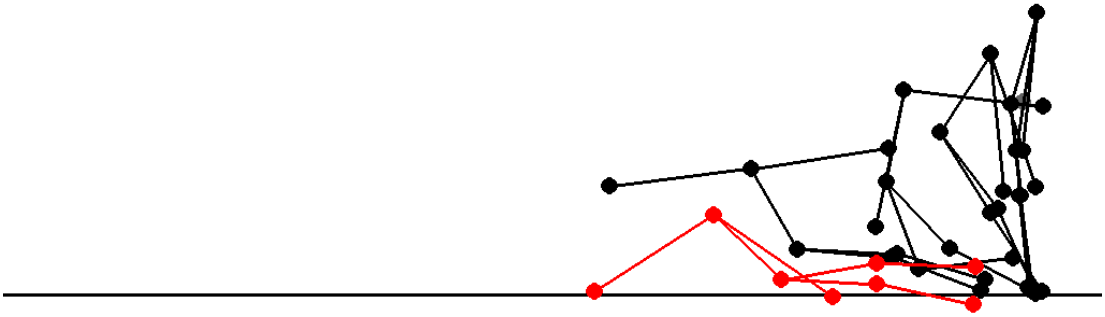


Figure 5.11: The stochastic walker, with replanning every 30 steps. Renderings from every 10^{th} frame.

(Figure 5.13). Locomotion is also performed successfully in these more difficult settings, albeit at a slower pace. In the deterministic walker, using a false stochastic generative model results in locomotion that takes 1.7 times as long to reach a fixed point as when the correct generative model is used. The policy developed for this experiment is characteristically more conservative than the policy developed for the normal deterministic walker with correct model. In particular, in this experiment the body spends almost the entire episode with at least one foot in contact with the ground, whereas in the fully deterministic walker the body spends a fair amount of time airborne, completely free of any ground contact. Compared to the stochastic walker, planning with an incorrect deterministic model produces locomotion taking approximately 1.4 times as long as planning with a correct model. Similarly, in this experiment the agent takes a policy that is somewhat better suited to the model, as opposed to the actual environment, taking a more risky policy that spends less time with ground contact as compared to the policy constructed for the stochastic walker when the correct model is provided.

The purpose of the final experimental domain is to demonstrate the ability

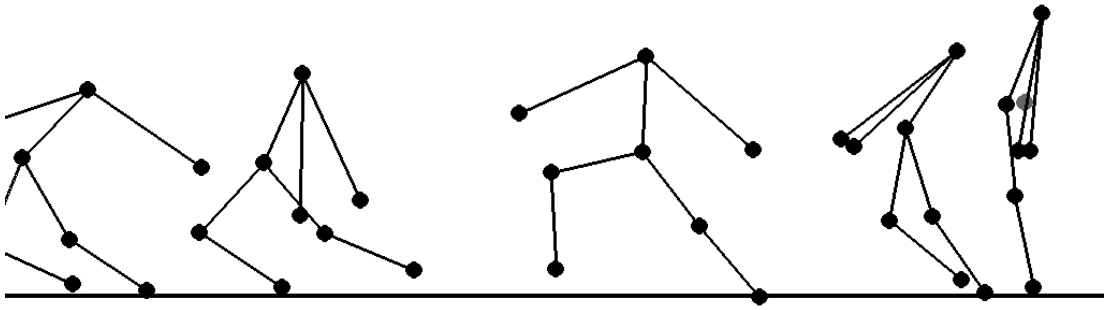


Figure 5.12: Performance of CEOLP in the deterministic walker, with an *incorrect* stochastic model. Renderings from every 30^{th} frame.

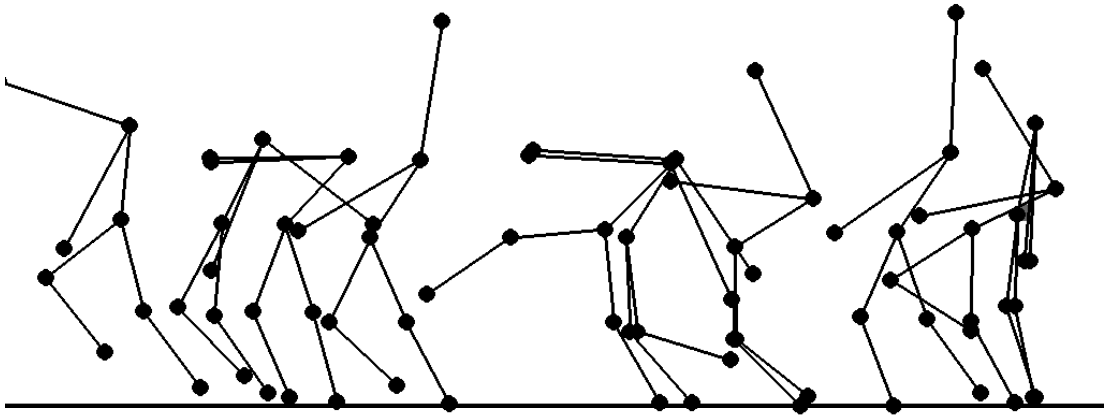


Figure 5.13: Performance of CEOLP in the stochastic walker, with an *incorrect* deterministic model. Renderings from every 30^{th} frame.

of open-loop planning with CE to enable a humanoid to traverse uneven terrain. In particular, the final task requires the descent of a flight of stairs. Aside from the addition of stairs, the setting is identical to the deterministic walker. The solution to the task found by CE appears in Figure 5.14. As can be seen,

the policy found is to run to the edge of the top step and then take a leap that clears the entire flight of stairs. The landing is performed successfully and running proceeded after the landing (not rendered). We anticipated a policy that would walk down the steps, but because the goal is to maximize the velocity of the hip along the horizontal axis, jumping across the the flight of stairs is superior to deliberately walking down it step by step, as long as the landing can be performed without falling.

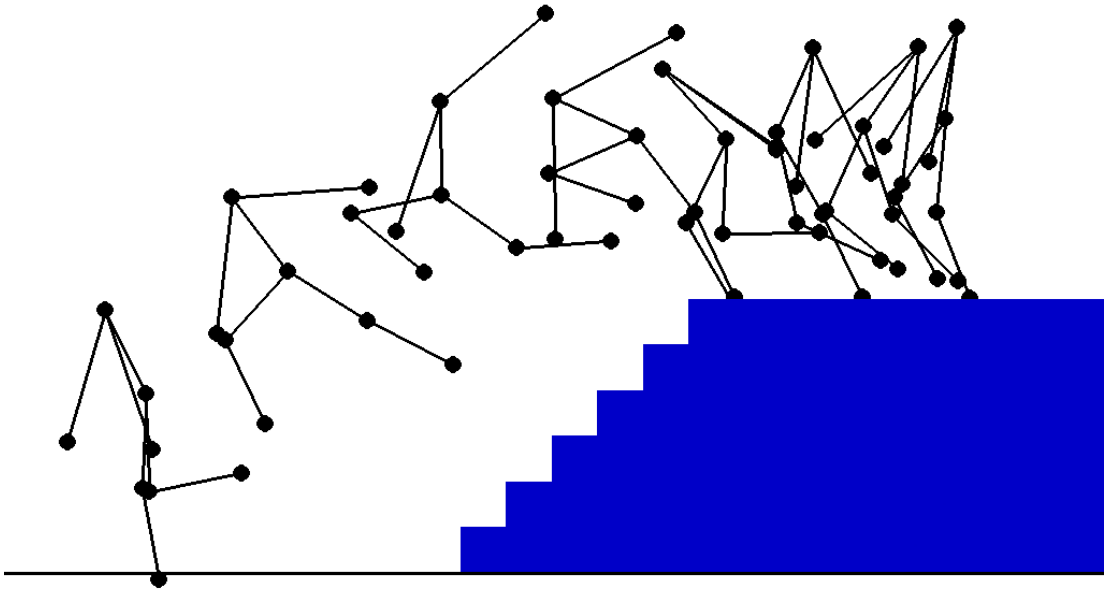


Figure 5.14: Cross-Entropy finding a creative solution to the stair-descent problem. Renderings from every 10^{th} frame.

5.4 Iterative Greedy Rejection

CEOLP is able to plan effectively in large domains because of its ability to generalize over different sequences of actions. In some settings, however, the similarity between different sequences of actions has almost no implication on the similarity of returns. This is the case, for example, in pure navigation tasks that give a large reward for reaching the goal, but zero reward elsewhere. In this type of domain, because reward is very sparse, small changes in policy can be the difference between zero and nonzero returns. As a result, the landscape of returns versus policy is highly unsmooth, violating common assumptions of smoothness.

The infinite-armed bandit problem (Goschin, Weinstein, Littman, and Chastain, 2012) defines a problem where an agent is allowed either to sample arms from an unknown distribution D, $a \sim D()$, or to resample an arm previously drawn from D, $r \sim R(a)$. Arms may be drawn from continuous or discrete spaces, making the approach applicable to domains with discrete or continuous action spaces with the only modification being to D. Based on the PAC formalism, the algorithms for use in the infinite-armed bandit problem require the specification of the common ϵ and δ parameters, along with a minimum acceptable reward, r_0. Given these values, a bound on the minimal sample complexity based on the Hoeffding/Bernstein bounds is $\Omega\left(1/\epsilon^2\left(1/\rho + \log 1/\delta\right)\right)$, with ρ being the probability of sampling an arm with expected reward of at least r_0, or $\rho = P_{a\sim D}(\mathbb{E}[R(a) \geq r_0])$.

In the original presentation of the problem (Goschin, Weinstein, Littman, and Chastain, 2012), three algorithms are discussed that have polynomial time bounds similar to the bound just described. Empirically, the algorithm called iterative greedy rejection (IGR) achieves the best performance, even though the

theoretical bounds provided are slighly inferior to the other two methods. IGR operates by greedily rejecting arms at any point that their sample mean drops below r_0, and only accepts an arm once it is possible to prove $\mathbb{E}[R(a) \geq r_0 - \epsilon]$ with probability $1 - \delta$, according to the Hoeffding bound. While this may result in a large number of rejected good arms, the algorithm compensates by rejecting poor arms very quickly (for example, if the arm has a Bernoulli payoff, a reward of 0 on the first pull will result in the arm being rejected immediately).

IGR and related approaches are shown to be effective planning algorithms in a number of domains. The most complex and difficult domain tested is the videogame *Pitfall!*, which will be discussed at length here. Prior to the application of IGR and related methods, the only known algorithm able to construct policies in this domain solved only the very simple initial screen. In addition, that method requires significant expert knowledge to define the types of interactions that can occur between objects (Diuk et al., 2008).

The success of IGR as the state of the art for planning in *Pitfall!* is due to the minimal assumptions made by the algorithm. In fact, the only assumption made is that a successful sequence of actions can be drawn from D (a uniform distribution over action sequences in the experimental setting). The lack of other assumptions is crucial because *Pitfall!* has many properties that violate standard assumptions of other planning algorithms, which are discussed briefly here.

Because the game runs in a complex emulator, it is not possible to reach arbitrary states at any point in time - only an episodic generative model is available. Not only is it impossible to reach arbitrary states at any time, but even defining state is quite difficult in *Pitfall!* The state is only fully defined by the entire configuration of memory, which is enormous at 8^{128}, which the agent

does not access to. Therefore, the domain is partially observable (Kaelbling et al., 1998), as agents only have access to image frames and not contents of memory. The game is non-Markovian on the level of on screen pixels, as some important information is conveyed only at rare intervals. Although the domain is a POMDP, open-loop planners such as IGR can plan effectively in *Pitfall!*, as state is ignored, and the required reset to s_0 is available through the emulator. This an example of how making minimal assumptions about the domain lead to more general planning algorithms. Aside from partial observability, smoothness between action sequences and returns does not exist because the difference in one pixel location anywhere along a trajectory hundreds of steps long can mean the difference between success and failure, making straightforward generalization impossible. Finally, in the experiments run, stochasticity is introduced by occasionally randomly changing the requested action, meaning that planners must tolerate noise.

Given all these complications of huge state space, partial observability, no domain expertise, and stochasticity, planning in *Pitfall!* objectively seems extremely difficult. IGR, which makes almost no assumptions about the domain, however, is able to plan sequences of actions hundreds of steps long (the entire hypothesis space as designed in the experiment is 8^{500}) in even the most difficult screens reliably with only a few thousand trajectories (Goschin, Weinstein, Littman, and Chastain, 2012). The best policy found for three screens are shown in Figure 5.15. Although IGR as tested operates on a domain with a discrete action set, because it only samples arms from D, it can be used with no modification to plan in domains that have continuous state or action spaces.

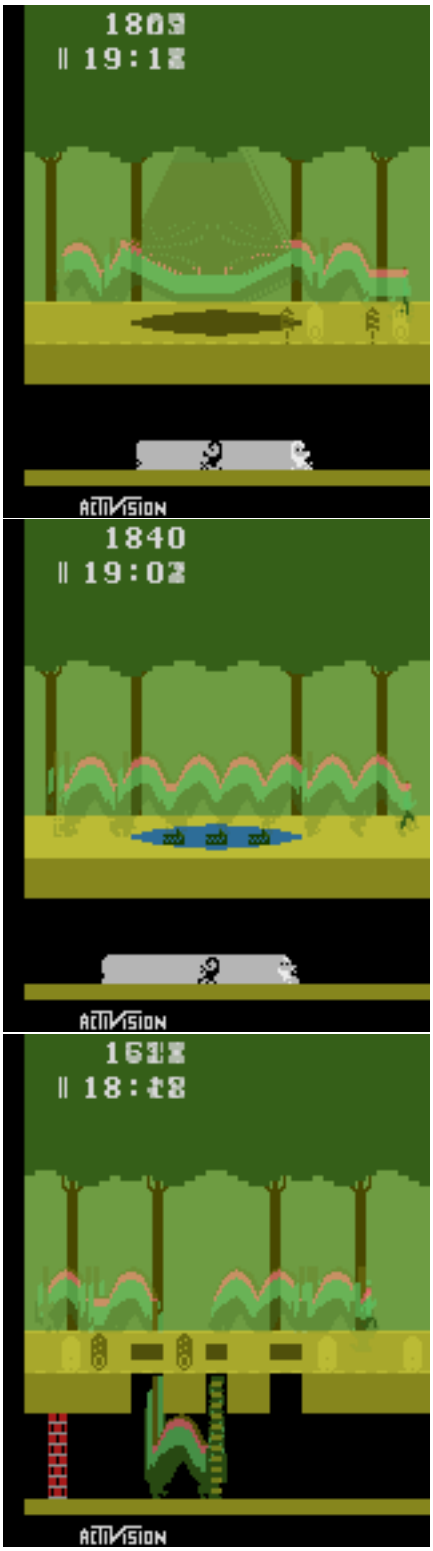
5.5 Discussion

In this section, two planning algorithms are presented that produce effective policies in large planning domains. The first algorithm, CEOLP, is an open-loop planner in the same form of HOLOP but underpinned by cross-entropy optimization. Direct comparison between the two is difficult as HOLOP is a parameter-free algorithm with strong performance guarantees, and CEOLP is sensitive to a wide number of parameters and does not have known theoretic guarantees for the setting considered. That in mind, results show that when attention is paid to its parameterization, CEOLP is more efficient with increasing sample size than HOLOP. In the most complex domain considered in this work, CEOLP is used to produce effective walking and running gaits in a high dimensional humanoid locomotion task even in the face of uneven terrain, noise, and minimal domain knowledge, granted only an episodic generative model. This is the case even though the formal study legged locomotion (approached from which are not used here) is a complex field of research to itself, generally requiring significant amounts of expert knowledge.

In addition, IGR is also introduced. Following in the pattern of evaluating planning algorithms in complex domains, IGR is shown to be an effective planner in the game of *Pitfall!* The success of IGR there is partially due to the generality of the method, as it produces policies far superior than the previous state of the art, which makes a number of hard assumptions and is only capable of completing the simple first screen (Diuk et al., 2008). Although the domain has a discrete action space, its is extremely challenging due to the high-dimensional state space, partial observability, and stochasticity, which are problems that can also exist in continuous domains. Likewise, the method can

be used in domains with continuous action spaces simply by changing the distribution D from a discrete to continuous distribution (in most cases, uniform distributions over action sequences suffice). One particular strength of IGR is that the only assumption made is that "good" arms or action sequences are produced by D with some minimal likelihood. This means that it can be applied to almost any planning setting. Even in the absence of knowledge of what a good return is, methods such as the doubling trick (Bubeck et al., 2010) may be used to produce acceptable plans.

Figure 5.15: Successful policies found by IGR in three challenging levels in *Pitfall!*



Chapter 6
Conclusion

Although reinforcement learning is a very general problem setting, research in the area has traditionally focused on a small scope of problems, considering domains of small, finite size. Even though most canonical RL domains are actually continuous, due to focus on this small subset of problems, it has become accepted practice to apply a coarse discretization to the original continuous domain and plan in the resulting discrete domain. This work has focused on raising a critical view of the approach.

Chapter 1 introduced the general problem setting, the thesis statement, and some intuitive arguments as to why the thesis statement may be correct. Chapter 2 then discussed background information on discrete planning, including why global planning is impractical in the case considered here. In response to this, the motivation and history of local planning is presented, concluding with to the state of the art of planning in discrete domains. Chapter 3 introduced planning in continuous domains, demonstrating that, from the perspective of regret, optimizing according to a discretization is meaningless. Following that point, a number of full planning algorithms are presented, finishing with HOLOP, which has tight performance guarantees, is state agnostic, and plans natively in continuous action domains. Chapter 4 empirically compares the state of the art in discrete planning with HOLOP, showing that the latter has significantly better performance than all discrete planners across the many

domains tested. In addition to cumulative reward, other important factors of memory and computational requirements were considered, showing that, from all perspectives, discrete planners do not scale to even medium sized domains. Chapter 5 then moved consideration from small and medium sized domains to significantly more difficult large high dimensional and even partially observable domains, showing that methods that reason natively in the planning domain without modification are the state of the art.

For the remainder of this chapter, we will change focus to related algorithms not yet discussed as well as potential extensions to the planners considered as future work. Finally, some remaining concluding remarks are presented.

6.1 Additional Related Work

Although this work is ultimately concerned with planning, HOLOP, CEOLP, and IGR have all focused on algorithms that cast planning as an optimization problem. As such, we will discuss a number of global optimization algorithms as well as pure planning algorithms. Algorithms introduced here overlap with a number of fields of study including operations research, control theory, and motion planning, so it is impossible to discuss all related work. Therefore, we restrict consideration to algorithms that are particularly related because of theoretical contributions, similarity in setting, or algorithmic similarity to previously discussed methods.

6.1.1 Optimization Algorithms

Because the field of optimization is very large, we focus on a restricted class of optimization algorithms that are useful for the setting considered. Specifically, because the MDPs considered have noise in the transition function, optimization algorithms (aside from DIRECT, which is discussed because of other novel characteristics) must tolerate noise (Assumption 5). Limited domain expertise (Assumption 4) introduces two other constraints that rule out large classes of optimization algorithms. First, algorithms must perform global optimization, as local optimization requires expert knowledge to initialize search near the global optimum. Secondly, many optimization algorithms rely on convexity; we do not provide any guarantees of convexity, and do not provide information in terms of the overall shape of any aspects of the problem such as R, T, or V.

Dividing Rectangles (DIRECT)

A common issue with branch and bound optimization algorithms based on the Lipschitz constant K, is that they require this constant to operate. When unknown, there are methods that attempt to estimate it, but estimates should be conservative to avoid prematurely removing areas from consideration. The DIRECT algorithm (Jones et al., 1993) operates without K, or any estimate of the value. Instead, the algorithm simultaneously searches over all values of K, creating ranges of possible values for each cell, and selecting any cell that is potentially optimal. The method assumes a deterministic domain, but because of the significance of the method it is being discussed, as it is often referenced in more recent work. Although it has been shown to work well in practice, it

does not have tight regret bounds and convergence is only guaranteed asymptotically (Bubeck et al., 2011b).

Zooming Algorithm

The zooming algorithm (Kleinberg, 2004, Kleinberg et al., 2008) combines the upper confidence bound technique with an adaptive step that uses samples to focus on regions near the possible optimum and to explore more thoroughly in these regions. The analytical results place a bound on what is called the covering dimension, which is related to the number of balls needed to cover a corresponding metric space. Based on the covering dimension d, the best possible regret at time step is $O(N^{(d+1)/(d+2)})$. Ultimately, the algorithm is complex and requires a "covering oracle" that determines if a region is covered by a set of points, and if not, reveals what regions need covering. Because of these complexities (in particular the assumption of a covering oracle which may not exist), the zooming algorithm is more interesting theoretically than it is practically useful, and is an influence in more practical optimization algorithms developed later.

UCB-AIR

The most common cause for the development of poor strategies when applying a discretization to a high dimensional continuous domain is the violation of the implicit assumption that $N >> |A'|$. Algorithms developed for this setting, such as IGR are a small exception to this rule. Another algorithm designed for use in this setting is UCB-AIR, or arm-increasing rule (Wang et al.,

2008). The primary difference between IGR and UCB-AIR is that in IGR a desired average reward is specified, whereas in UCB-AIR there is an assumption about the probability of sampling a near-optimal arm. The algorithm is also related to OLOP in the sequential planning setting, with UCB-Air having superior bounds in some cases, but also requiring more domain knowledge in terms of the number of near-optimal arms which OLOP does not require (Bubeck and Munos, 2010).

Estimating the Lipschitz Constant

When the Lipschitz constant K is unknown, one approach is to estimate bounds according to a range of values, as is the case in DIRECT. Another approach is to estimate K based on samples from the domain. Theoretical results show that optimal regret (even in comparison to the case where K is known) can be achieved by this approach (Bubeck et al., 2011b). The regret bounds according to the method are $O(K^{|A|/(|A|+2)}N^{(|A|+1)/(|A|+2)})$ in domains where $\mathbb{E}[R]$ and its derivative are Lipschitz. The algorithm works by performing a discretization, similar to that proposed in the Zooming algorithm (Kleinberg et al., 2008). Functioning in two phases, the algorithm first performs uniform exploration to produce an estimate of K that is tight enough to achieve the regret bound. Following this step, an exploration-exploitation phase is conducted, which can be performed by existing bandit algorithms. They also present the idea that such a branch and bound method may be applicable in settings outside pure optimization; it is likely that sequential planning is another potential application of this general approach.

6.1.2 Local Planners

Ultimately, this work is concerned with finding native continuous planners that outperform the state of the art in discrete planning when coupled with coarse discretization. Here, we will discuss a number of other algorithms outside of HOLOP, CEOLP, and IGR that function natively in continuous domains. Some algorithms discussed here violate the assumptions made in this work, but are still worthy of mention because of their contributions.

Differential Dynamic Programming

Differential dynamic programming (DDP) operates based on the assumption that the domain has locally quadratic dynamics and value function (Mayne, 1966). Although it is a fairly old method, it has been used to produce state of the art results in a number of domains. Perhaps the most significant success of the approach was its use in performing real-time control of highly complex acrobatic helicopter maneuvers (Abbeel et al., 2007, Coates et al., 2008, Abbeel et al., 2010). Another significant application of the approach was in the domain of simulated humanoid control (Tassa et al., 2007b, Tassa and Todorov, 2010, Erez, 2011, Erez et al., 2011, Tassa et al., 2012).

While, DDP has been shown to be extremely effective (especially in complex time sensitive domains), it does have a number of limitations. Firstly, the approach is fragile due to the strong assumptions made with respect to the form of dynamics of the MDP. Domains that have discontinuities in rewards or dynamics (which occur in any physical setting such as walking where hard contacts occur) need to have noise artificially introduced to smooth out discontinuities. This noise becomes a parameter that must be controlled, because

the introduction of noise reduces policy quality in the original domain (generally becoming more conservative), but introducing too little noise may result in failure (Tassa and Todorov, 2010). Another result of the hard dependence on quadratic dynamics is that regularization must used to prevent divergence by ensuring small policy changes between generations. Controlling the regularization variable leads to a trade off between safety from divergence and speed of convergence (Erez, 2011).

Although these issues require extra parameter manipulation to use the domain successfully in many domains, they do not prevent DDP from being used in the setting we consider. There are however, two more significant assumptions made when using DDP. Firstly, we assume access only to an episodic generative model (meaning that states cannot be selected arbitrarily during rollouts), but because of the approach of finite differences, DDP requires a full generative model. Secondly, DDP is a gradient method, meaning convergence is only to the locally optimal policy. As with other gradient methods, the impressive results produced by DDP requires significant domain expertise to initialize search in a manner that allows effective policies to be generated (Erez, 2011).

Binary Action Search

Binary action search (BAS) is a method of transforming a discrete action, continuous state planner into a continuous action, continuous state planner (Pazis and Lagoudakis, 2009). Similar to HOO, this method dynamically divides the continuous action space into regions of smaller size. It performs this transformation by creating an augmented state space that represents state in the traditional sense, as well as the current selected action. From this point, the discrete

action set becomes binary decisions in each original dimension in A, with each binary action causing an increase or decrease in the binary search of the corresponding action dimension. Because of the generality of the approach, it can be used either in pure planning algorithms such as those discussed here, or in the full learning setting where a generative model is not available. Results in the literature on the full learning setting show that HOLOP is empirically more effective at conducting decompositions: see Pazis and Lagoudakis (2009) and Weinstein and Littman (2012).

Optimistic Planning for Sparsely Stochastic Systems

One of the reasons for the high planning costs of sparse sampling and FSSS is that both algorithms must take enough samples to adequately factor in stochasticity in T. One way to reduce planning costs is to construct algorithms that can take advantage of particular features of the planning domain, such as limited stochasticity. In particular, the optimistic planning for sparsely stochastic systems algorithm leverages sparsity in the transition matrix, if it exists (Buşoniu et al., 2011). The algorithm in many respects is similar to FSSS, performing rollouts based on computed bounds estimated during rollouts. Differences are that it also requires access to a set of all possible next states, and analysis is with respect to simple regret, as opposed to a PAC-style result. Empirical results contain a comparison to OLOP, which is shown to be less effective in practice.

Tree Learning Search

Tree learning search (TLS) is another method of performing planning in continuous MDPs (Van den Broeck and Driessens, 2011). In TLS, data from rollouts are given to an on-line regression tree learner, which determines which regions of the action space may contain high reward. These estimates then inform the policy when performing rollouts at the next iteration. Unfortunately, TLS lacks any theoretical guarantees of performance, and was not shown to perform better than planners that used *a priori* discretization (Van den Broeck and Driessens, 2011).

Schepers (2012) compared TLS to HOLOP with regard to the form of the decomposition of continuous spaces conducted during planning, and places TLS and HOLOP in a family of related algorithms. Although, given a budget of samples from a generative model, TLS is outperformed by HOLOP and other discrete planners, given a budget of clock time, the algorithm is highly effective. In experiments, TLS is shown to be faster than HOLOP as well as UCT. This is because the data structure built by TLS only grows when data indicates that further decomposition would increase decision quality. As a result, TLS is generally able to build very compact representations of its policy, which is in contrast to HOLOP and UCT, which maintain data structures that grow in either the number of trajectories (HOLOP), or in the total samples from the generative model (UCT).

POWER

Policy learning by weighting exploration with the returns (POWER), is an algorithm intended for use in episodic continuous state and action MDPs (Kober and Peters, 2011). Instead of gradient methods, which require learning rates

and are vulnerable to noise and getting caught in local optima, POWER is instead based on the expectation-maximization (EM) paradigm (Dempster et al., 1977). The mathematical underpinning of the algorithm is based on analysis of the worst-case performance, as opposed to the more common consideration of the performance of policies in expectation. Exploration in POWER is performed locally, based on a temperature that decreases over time, and results in the original publication show that some other gradient methods can be derived from the analysis used to produce the algorithm. As a local-search method, the experimental results presented with the algorithm require expert initialization to start search near the global optimum.

Path Integral Policy Improvement and Related Algorithms

A particularly influential approach, path integral policy improvement (PI^2), (Kappen, 2005) is similar in some ways to differential dynamic programming, and has resulted in a whole family of related algorithms. A benefit of PI^2 in its original formulation is that it is not particularly complex (although more complicated than CELOP), and has a low risk of divergence because it does not perform matrix inversions or gradient estimations. Empirically, the approach has been successful both in simulation and real world obstacle avoidance tasks with the "little dog" robot (Theodorou et al., 2010). Aside from exploration noise, the method is mostly parameter free. One significant constraint is that knowledge of the dynamics is required and there are constraints on their form.

Chapter 5 focused primarily on cross-entropy for planning. Recent work has tied cross-entropy with the PI^2 family of algorithms (Theodorou et al., 2010) and the covariance matrix adaptation evolution strategy (Hansen and Ostermeier, 2001), with essential differences being how samples are weighed

(Stulp and Sigaud, 2012). One resulting benefit of creating this association is the creation of a new planning algorithm that attempts to take the most advantageous components of each algorithm, producing Path Integral Policy Improvement with Covariance Matrix Adaptation (PI^2-CMA). A primary advantage of the method is that it determines the magnitude of exploration noise.

Covariant Hamiltonian optimization for motion planning (CHOMP) is a rollout method that attempts to improve return according to gradient estimation (Ratliff et al., 2009). It is intended for use in motion planning problems, and, like PI^2, has impressive empirical results including uneven domain traversal with the little dog robot. As a gradient method, however, it runs the risk of converging to poor local optima (Theodorou et al., 2010). In response to the tendency of CHOMP to converge to poor policies, Stochastic trajectory optimization for motion planning (STOMP), a planning algorithm based on PI^2, was proposed (Kalakrishnan et al., 2011). Like CHOMP, it can plan smooth trajectories in motion planning tasks, simultaneously avoiding obstacles and optimizing constraints, which in traditional motion planning algorithms occurs in separate steps. An added benefit of the removal of gradients is that it allows for more general reward structures, and can optimize arbitrary terms in the cost function like motor efforts.

6.2 Extensions and Future Work

Although the computational costs of HOLOP, CEOLP, and IGR are relatively low, the actual running time of the algorithms depends heavily on the cost of performing the rollouts, and in particular sampling from R and T. In domains that have simple dynamics such as the double integrator, this is not a great

concern. For domains that are expensive to evaluate, such as full physics simulations or machine emulation as required in Chapter 5, actual running time can be long, or even prohibitive. Future research regarding improved responsiveness of the algorithms when coupled with expensive evaluation functions is warranted in settings with time sensitivity. Following the consideration of the general RL setting, we focus on general purpose extensions. Therefore, we ignore domain specific extensions such as the use of Poincaré maps and limit cycles which, is common in the literature of legged locomotion (Manchester et al., 2011).

Warm Starting

In all the planning settings considered, the only planning parameter that changes throughout execution is the starting state s_0. As such, the planner never benefits from previous experience, which is generally wasteful. Information from previous planning steps can be very useful for current decision making. A simple approach to reduce planning time is to "warm start" planning with results from the last round of planning. A simple example is $\hat{\Phi}_0$ in CEOLP. In the experiments, in Chapter 5, $\hat{\Phi}_0$ defines a uniform sampling of the entire action range, but another option would be to initialize $\hat{\Phi}_0$ to $\hat{\Phi}_G$ from the last round of planning. In particular, $\hat{\Phi}_0$ could be defined by shifting the previous trajectories forward, removing the just-taken initial action in the sequence, and replacing the last action with one that is randomly selected. This method is used in much of the DDP (Section 6.1.2) work to drastically reduce planning costs. A risk is that search during planning becomes much more local in nature, and additional noise should be included to maintain policy diversity. Informal results in CEOLP indicate both policy quality improvements along with drastic

savings in sample use.

Another option is to use learning algorithms to estimate an appropriate $\hat{\Phi}_0$ based on s_0 and previous experience in the domain. A similar method of reusing search data transformed UCT into its variant, UCT-RAVE, which was a key component of MOGO, the first computer Go agent to achieve master level play in 9x9 Go (Gelly and Silver, 2008).

Parallelization

When working in large domains, parallelization (Amdahl, 1967) of planning becomes increasingly important (Bourki et al., 2011). Based on the assumption of limited domain knowledge, higher dimensional domains must in general be met with increasingly large N. Additionally, the trend in modern processors has been to increase core count as opposed to speed, meaning that the improvement in planning time for a single core is decreasing with time.

A central problem in parallelization is the cost-granularity problem, meaning that the overhead of performing the actual parallelization (delegation of work, integration of results) must be lower than the actual cost of performing the work (planning, performing rollouts) in order for parallelization to actually improve running time. Fortunately, in most cases the costs of conducting planning is high with respect to the cost of delegation, and empirical results show parallelization of planning yields the most significant speedups in domains that are computationally expensive (Evans et al., 2007). Aside from the improvements in processing time, different forms of parallelization can serve to improve performance by increasing the diversity of planning (Chaslot et al., 2008), and can in some cases behave in a manner similar to bagging (Schapire, 1990) in standard supervised learning (Fern and Lewis, 2011).

By its nature, closed-loop planning is difficult to parallelize. In closed-loop planning, policies are commonly encoded by a tree or some other complex data structure. Such a data structure either requires shared memory or frequent communication to keep the data structure synchronized across processes. If shared memory or frequent message passing is to be avoided, the only form of parallelization that is usable in most closed-loop planners is fairly primitive root parallelization (Chaslot et al., 2008).

Parallelization of open-loop planners, on the other hand, is ver simple. One method is to simply have each process execute the following steps: query for an open-loop policy, execute the policy, and then report the return to the central process. This paradigm shares much in common with batch-mode optimization (Desautels et al., 2012). Open-loop planning therefore, is simpler to parallelize than closed-loop planning, and is also more efficient. Additionally, because shared memory is not required, algorithms can be trivially parallelized across physical machines with minimal communication, allowing for highly scalable parallelization.

Some open-loop planners can parallelize even more efficiently than at the level of individual sequences of actions. In CEOLP, for example, it is trivial to parallelize across a generation; the only data that needs to be passed is Φ_g. IGR is simpler, as it is essentially parallelizable without limit, and the only piece of data that must be sent is the distribution over actions D, with the first process that finds a good sequence simply returning it. Parallelization can also be performed with HOLOP; results (aside from time and memory measurements) in Chapter 4 used simple root parallelization (Weinstein and Littman, 2012). Another method for conducting batch sampling with HOO would be to pass the

action range for the leaf, $A(\nu)$, to each process, and allow a fixed number of actions to be executed from $A(\nu)$, as opposed to just one. For a large part of the planning phase, $A(\nu)$ is quite large, so it would not significantly impact sample diversity. Furthermore, theoretical guarantees remain intact when using this method.

Value-Function Approximation

In our rollout planners, the structure of Generic Rollout Planning (Algorithm 7) a function EVALUATE exists to assign a value to the final state of the trajectory. In all empirical results, we simply evaluate every function as 0, to demonstrate the performance of the planning algorithms in isolation. However, the EVALUATE function can be concretely instantiated as some form of VFA. Although there are significant problems that can arise when using VFAs (Section 3.3.1), there are a few reasons to consider their use in conjunction with rollout planners. When using a VFA, it is always better to use it as an evaluation function at the end of the rollout, as opposed to simply acting greedily according to the VFA. Theoretical results show that using rollouts in conjunction with a heuristic is always at least as good as the heuristic function alone (Bertsekas et al., 1997).

The use of VFAs as such evaluation functions have led to some of the most significant successes RL in game environments. Short rollouts rollouts followed by VFAs defined play in TD-Gammon, which achieved world-class play in backgammon (Tesauro, 1995). Heuristic UCT, a variant of UCT which incorporates VFAs was an important component in MOGO, which led to the first computer agent that achieved master level play in 9x9 Go (Gelly and Silver, 2008), and a similar approach is documented in Silver et al. (2008). The most

well-known application of this is in Watson, the Jeopardy! agent built by IBM, which used a VFA in a manner very similar to TD-Gammon to predict the probability of winning Jeopardy! based on differing wager amounts and confidence levels and other state information (Ferrucci et al., 2010).

Additionally, while standard VFAs are unsafe, there is a class of algorithms that performs VFAs based on the recorded returns of trajectories, as opposed to the standard dynamic programming approach (Stolle and Atkeson, 2006). Using trajectory libraries in such a manner was a key element in the success of recent applications of DDP (Liu and Atkeson, 2009, Tassa and Todorov, 2010).

LQR Trees (Tedrake, 2009) is another algorithm that helps guide search and has had a significant impact in the control and robotics literature. Like many approaches from control theory, the primary objective is to bring the system into stabilization around a goal state, as opposed to value maximization. As such, it does not provide any guarantees of optimality. While the first description of the algorithm is very complex and difficult to implement, a later extension (Reist and Tedrake, 2010) allows for much of the complex computations to performed based on simple estimates from the generative model. Planners discussed in this work are a natural fit with LQR-Trees, as the LQR-Trees do not directly define a policy; a continuous planner is needed in conjunction to actually formulate a policy.

A final note, however, is that attempting to use VFAs introduces costs in the full size of the domain. Some of the most impressive applications rely on sufficient domain knowledge to perform significant domain reduction to mitigate this issue (Tassa and Todorov, 2010, Erez et al., 2011). Without reduction, the costs of planning can increase very quickly with increasing domain size (Reist and Tedrake, 2010).

Model Building

This work considered the pure planning problem in RL where an episodic generative model is provided to the planner. Another possible setting is where an EGM is not known *a priori*. In this case, an EGM can be learned from direct interaction with the domain, using acquired $\langle s, a, r, s' \rangle$ tuples. In general, two components are necessary to build an accurate model in such an RL scenario: exploration and supervised learning.

Multi-Resolution Exploration (Nouri and Littman, 2008) or MRE is a method that can be used inside almost any RL algorithm to drive efficient exploration of a domain, assuming R and T are Lipschitz continuous. Originally concerned with domains of continuous state and discrete actions, the algorithm can be modified simply to explore the continuous space $S \times A$ (Weinstein and Littman, 2012). When used in this manner, MRE functions by decomposing $S \times A$ via a tree structure. Each leaf of the tree represents a region of $S \times A$ within which "knownness" $\kappa(s, a)$ is computed. As samples are experienced in the MDP, they are added to the corresponding leaf that covers the sample. Once the leaf contains a certain number of samples, it is bisected.

When a query is made for a predicted reward and next state, MRE may intervene and return a transition to s_{max} (a state with value $V_{\max}$) with probability $1 - \kappa(s, a)$. The presence of this artificial state draws the agent to explore and improve its model of that region. Doing so increases κ, which in turn means it will be less likely to be explored in the future. In terms of model building, the tree used by MRE can be used not only to drive exploration, but also as a regression tree. Instead of inserting only $\langle s, a \rangle$ samples, inserting $\langle s, a, r, s' \rangle$ allows estimates $\hat{R}$ and $\hat{T}$ to be constructed.

We note that other methods of exploration and model building are available. In terms of randomized exploration, the most popular method is ϵ-greedy, but as an undirected search method it can fail to explore efficiently in certain MDPs (Strehl and Littman, 2004), a limitation MRE does not have. In terms of model building, any supervised learning algorithm can be used to build estimates of R and T. Once MRE is chosen as the method of exploration, however, the same tree can also be used as a regression tree with only constant-time additional costs. Emprically, HOLOP combined with MRE for exploration and model building has been shown to outperform other discrete and continuous learning algorithms (Weinstein and Littman, 2012).

6.3 In Closing

Overall, this work has produced analytical as well as empirical results that demonstrate the veracity of the thesis statement:

When compared to algorithms that require discretization of continuous state-action MDPs, algorithms designed to function natively in continuous spaces have: lower sample complexity, higher quality solutions, and are more robust.

To be clear, arguments are not being made against discretization as a whole, as HOLOP is an algorithm that performs a discretization performed adaptively and automatically. Coarse discretizations, on the other hand, make the poor tradeoff of focusing too little on important parts of the search space while focusing too much on suboptimal regions. As the dimension of the problem increases, the need to carefully allocate samples becomes critical for effective planning. A simple idea is that planning algorithms should be suitable for the

domains they are applied to, and not vice versa.

Indeed, coarse discretization is one of the worst ways to attempt planning in continuous domains. There are a number of reasons why. From the perspective of generalization, a coarse discretization is simply a primitive form of function approximation; the fact that it is not used in general supervised learning is good evidence that the approach will not make good use of provided data. Because it performs poor generalization, coarse discretization leads to a difficult signal-to-noise problem, where, due to the curse of dimensionality, actions can only be sampled once and rollouts devolve into vanilla Monte Carlo estimates of quality. When this occurs it becomes impossible (in general) to select actions optimally, as actions are selected according to a chance, as opposed to near optimal, policy. When applied to large domains, coarse discretization leads to difficulties in operations that are generally taken for granted. For example, simply performing a $\max_{A'}$ becomes exponentially expensive in $|A|$, and even simple enumeration of $|A'|$ becomes prohibitively expensive.

Perhaps a reason that reliance on coarse discretization is so common is that, until recently, algorithms that function natively in continuous domains lead to significant problems. As discussed at length, VFA has a number of issues including the requirement that not only the final value functions, but all intermediate value functions be representable by the FA. The more significant issue is risk of divergence. Classical policy search also suffers from significant problems—functions representing the policy must be significantly complex to represent near-optimal policies while still having a simple parameter space. Additionally, most methods rely on gradient or local searches and converge only to local optima. Algorithms presented here greatly simplify the task of planning in continuous domains. They are both very simple to implement and

do not risk divergence of the value function. They perform an overlooked type of policy search without requiring domain expertise to initialize gradient search, or to determine efficient parameterizations of function approximators representing value functions or global policies.

Recall that the thesis statement does not claim that we have found an optimal continuous planning algorithm; such an algorithm does not exist for the general setting in RL. The claim is rather that there are continuous planners that outperform the best discrete planners that depend on a discretization provided *a priori*. A rather simple planning algorithm, HOLOP, is shown to be superior to a wide variety of state of the art discrete planners based on several important metrics. On the basis of theoretical performance, HOLOP achieves near-optimal cumulative regret and planners that rely on fixed discretizations have theoretically degenerate cumulative regret. On the basis of empirical performance, HOLOP outperforms all tested discrete planning algorithms in all domains considered. Additionally, on the basis of memory and computational costs, HOLOP significantly outperforms its discrete counterparts. Primarily, effective failure of discrete methods in even medium sized domains is due to the manner in which coarse discretization suffers from the curse of dimensionality.

Following the formal and empirical results demonstrating the inability of discrete planners to compete with native continuous methods, the discussion progressed from small and medium canonical RL domains to large domains that are still highly challenging for the state of the art. Two algorithms, CEOLP and IGR are presented that are both algorithmically very simple and produce effective policies for such domains. The simplicity of the approaches is partially derived from the minimal assumptions the algorithms make about the

planning domain, which makes them very general purpose; other methods generally require significant domain expertise to plan in the walking and *Pitfall!* domains of Chapter 5. Although no claims in particular are made about open-loop planning in the thesis statement, HOLOP, CEOLP, and IGR are all open-loop planning algorithms. Their ability to plan in the humanoid locomotion domain, which has hybrid state, and in *Pitfall!*, which is discrete but enormous and partially observable, is a testament to the advantages of the approach and the benefits of working directly from data and making minimal assumptions about the planning setting. The lack of assumptions also makes the algorithms less fragile as many algorithms make hard assumptions about domains and fail if those assumptions is violated, or if required provided information is incorrect.

Why is coarse discretization still favored?

In light of these results, it is worth asking the following question: *if coarse discretization has known unavoidable exponential costs as domains grow, and is known to fare so badly with respect to generalization, optimization, memory requirements, and computational costs, why is it still the case that RL literature continues to leverage this approach as both viable and state of the art?*

Part of the answer may lie in the results in Chapter 4. Consider again the results in Figure 4.13, which show that discrete planners are the fastest methods *only* for toy-sized domains (2 state dimensions and 1 action dimension). Additionally, the gap in performance between discrete and continuous planners in these smallest domains is still low. Domains of this size such as the inverted pendulum, acrobot, hillcar, double integrator, racetrack, and others,

are still the canonical baselines by which results are presented in the literature (Sutton and Barto, 1998). In most cases, experiments are not performed in larger domains; by doing so, simple and fundamental problems have simply been hidden, or worse, ignored.

Essentially, planning algorithms have come to overfit cannonical RL domains, which are toy domains unchanged for decades. The majority of even modern results in the RL literature, therefore, deal with domains that are easily and efficiently solvable with algorithms 20 years old (Moore and Atkeson, 1993). Comparing computational power of consumer CPUs between then and now shows a roughly 15,000-fold increase in computing power (Int, 2013a,b).

Just as other disciplines have had to develop new approaches in the move forward to more complex problems enabled by increasing computational power, if the field of reinforcement learning takes the challenge of working on high-dimensional domains, obsolete techniques will be discarded and new algorithms will be developed by necessity. On the positive side, the reinforcement learning competitions (Whiteson et al., 2010, Doe, 2013) have been providing new domains which can serve as baselines for comparing performance in non-trivial domains in the future. While there certainly is work in the literature that deals with planning in high dimensional domains, at the moment such work is the exception as opposed to the rule; moving forward it is hoped results in more complex domains will become the norm, leading to entirely new planning approaches that function natively in the domain of interest, whatever its properties.

Bibliography

Discontinued processors - intel microprocessor export compliance metrics. 2013a. URL http://www.intel.com/support/processors/sb/CS-020868.htm.

Intel core i7-3900 desktop processor extreme edition series. 2013b. URL http://download.intel.com/support/processors/corei7ee/sb/core_i7-3900_d_x.pdf.

Rl-competition. 2013. URL http://www.rl-competition.org/.

Pieter Abbeel, Adam Coates, and Andrew Y. Ng. Autonomous helicopter aerobatics through apprenticeship learning. *International Journal of Robotics Research*, 29(13):1608–1639, 2010.

Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *In Advances in Neural Information Processing Systems 19*. 2007.

Rajeev Agrawal. The continuum-armed bandit problem. *SIAM Journal on Control and Optimization*, 33(6):1926–1951, 1995.

G. Alon, D. P. Kroese, T. Raviv, and R. Rubinstein. Application of the cross-entropy method to the buffer allocation problem in a simulation-based environment. *Annals Operations Research*, 134(1):137–151, 2005.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies*, pages 483–485. 1967.

Chris Atkeson, Andrew Moore, and Stefan Schaal. Locally weighted learning for control. *AI Review*, 11:75–113, 1997.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81 – 138, 1995.

Richard E. Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6, 1957.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the Association for Computing Machinery*, 18(9):509–517, 1975.

Dimitri P. Bertsekas. Dynamic programming and suboptimal control: A survey from ADP to MPC. *Eur. J. Control*, 11(4-5):310–334, 2005.

Dimitri P. Bertsekas, John N. Tsitsiklis, and Cynara Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3:245–262, 1997.

Pieter-Tjerk De Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134, 2005.

Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hrault, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssire, and Ziqin Yu. Scalability and parallelization of Monte-Carlo tree search. In *Computers and Games*, volume 6515, pages 48–58. Springer Berlin / Heidelberg, 2011.

Craig Boutilier, Thomas L. Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research (JAIR)*, 11:1–94, 1999.

Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, pages 369–376. Cambridge, MA, 1995.

S. R. K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a Monte-Carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704, 2012.

Sébastien Bubeck and Rémi Munos. Open loop optimistic planning. In *Conference on Learning Theory*. 2010.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization of X-armed bandits. In *Advances in Neural Information Processing Systems*, volume 22, pages 201–208. 2008.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. X-armed bandits. *CoRR*, abs/1001.4475, 2010.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. *X*-armed bandits. *Journal of Machine Learning Research*, 12:1655–1695, 2011a.

Sébastien Bubeck, Gilles Stoltz, and Jia Yuan Yu. Lipschitz bandits without the Lipschitz constant. In *Algorithmic Learning Theory*, pages 144–158. 2011b.

L. Buşoniu, R. Munos, B. De Schutter, and R. Babuška. Optimistic planning for sparsely stochastic systems. In *Institute of Electrical and Electronics Engineers Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, pages 48–55. Paris, France, 2011.

Lucian Busoniu, Remi Munos, Bart De Schutter, and Robert Babuska. Optimistic planning for sparsely stochastic systems. In *Institute of Electrical and Electronics Engineers Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 48–55. Paris, France, 2011.

Guillaume M. J-b. Chaslot, Mark H. M. Winands, and H. Jaap Van den Herik. Parallel Monte-Carlo tree search. In *Proceedings of the Conference on Computers and Games 2008*. 2008.

G.C. Chow. *Analysis and control of dynamic economic systems*. Wiley, 1975.

Adam Coates, Pieter Abbeel, and Andrew Y. Ng. Learning for control from multiple demonstrations. In *International Conference on Machine learning*, pages 144–151. 2008.

Amanda Coles, Andrew Coles, Angel Garcia Olaya, Sergio Jimenez, Carlos Linares Lopez, Scott Sanner, and Sungwook Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1):1–8, 2012.

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. In *Uncertainty in Artificial Intelligence*, pages 67–74. 2007.

Andre Costa, Owen Dafydd Jones, and Dirk Kroese. Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters*, 35(5):573 – 580, 2007.

Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. Ph.D. thesis, Institut National Polytechnique de Grenoble, 2002.

Marc Peter Deisenroth and Carl Edward Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *International Conference on Machine Learning*, pages 465–472. 2011.

Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

Thomas Desautels, Andreas Krause, and Joel Burdick. Parallelizing exploration exploitation tradeoffs with Gaussian process bandit optimization. In *Proceedings of the 2012 International Conference on Machine Learning*. Edinburgh, Scotland, 2012.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *International Conference on Machine Learning*, pages 240–247. 2008.

Carlos Diuk, Lihong Li, and Bethany R. Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *International Conference on Machine Learning*, volume 382. 2009.

Paul F. Dubois, Konrad Hinsen, and James Hugunin. Numerical python. *Computers in Physics*, 10(3), 1996.

Michael O. Duff and Andrew G. Barto. Local bandit approximation for optimal learning problems. In *Advances in Neural Information Processing Systems*, volume 9, pages 1019–1025. 1997.

Tom Erez. *Optimal Control for Autonomous Motor Behavior*. Ph.D. thesis, Washington University in Saint Louis, 2011.

Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite-horizon model predictive control for periodic tasks with contacts. In *Proceedings of Robotics: Science and Systems*. Los Angeles, CA, USA, 2011.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Maching Learning Research*, 6:503–556, 2005.

Gareth E. Evans, Jonathan M. Keith, and Dirk P. Kroese. Parallel cross-entropy optimization. In *2007 Winter Simulation Conference*, pages 2196–2202. Institute of Electrical and Electronics Engineers, 2007.

Eyal Even-dar, Shie Mannor, and Yishay Mansour. PAC bounds for multi-armed bandit and Markov decision processes. In *Conference on Computational Learning Theory*, pages 255–270. 2002.

Eyal Even-Dar, Shie Mannor, and Yishay Mansour. Action elimination and stopping conditions for the multi-armed bandit and reinforcement learning problems. *Journal of Machine Learning Research*, 7:1079–1105, 2006.

Valerii Vadimovich Fedorov. *Theory of Optimal Experiments*. Academic Press Inc., 1972.

Alan Fern and Paul Lewis. Ensemble Monte-Carlo planning: An empirical study. In *ICAPS*. 2011.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building watson: An overview of the DeepQA project. *AI Magazine*, 31(3), 2010.

Raphael Fonteneau, Susan A. Murphy, Louis Wehenkel, and Damien Ernst. Model-free Monte Carlo-like policy evaluation. *Journal of Machine Learning Research - Proceedings Track*, 9:217–224, 2010.

Victor Gabillon, Alessandro Lazaric, Mohammad Ghavamzadeh, and Bruno Scherrer. Classification-based policy iteration with a critic. In *International Conference on Machine Learning*, pages 1049–1056. 2011.

Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer Go. In *Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*, pages 1537–1540. 2008.

Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *International Conference on Machine Learning*, pages 261–268. San Francisco, CA, 1995.

Sergiu Goschin, Michael L. Littman, and David H. Ackley. The effects of selection on noisy fitness optimization. In *Genetic and Evolutionary Computation Conference (GECCO)*. 2011.

Sergiu Goschin, Michael L. Littman, and Ari Weinstein. The cross-entropy method optimizes for quantiles. In *International Conference on Machine Learning*. 2013.

Sergiu Goschin, Ari Weinstein, Michael L. Littman, and Erick Chastain. Planning in reward-rich domains via PAC bandits. *Journal of Machine Learning Research Workshop and Conference Proceedings*, 14, 2012.

Joshua T. Guerin and Judy Goldsmith. Constructing dynamic bayes net models of academic advising. In *Proceedings of the 8th Bayesian Modeling Applications Workshop*. 2011.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. ISSN 1063-6560.

Verena Heidrich-Meisner and Christian Igel. Evolution strategies for direct policy search. In *Proceedings of the 10th international conference on Parallel Problem Solving from Nature: PPSN X*, pages 428–437. Springer-Verlag, 2008.

Feng-hsiung Hsu. IBM's deep blue chess grandmaster chips. *Institute of Electrical and Electronics Engineers Micro*, 19(2), 1999.

D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.

Leslie Pack Kaelbling. *Learning in embedded systems*. MIT Press, Cambridge, MA, USA, 1993.

Leslie Pack Kaelbling. Associative reinforcement learning: Functions in k-DNF. *Machine Learning*, 15, 1994.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *International Conference on Robotics and Automation*. 2011.

H J Kappen. Path integrals and symmetry breaking for optimal control theory. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(11), 2005.

M. Kearns, S. Mansour, and A. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *International Joint Conference on Artificial Intelligence*. 1999.

Robert Kleinberg, Aleksandrs Slivkins, and Eli Upfal. Multi-armed bandits in metric spaces. In *Symposium on the Theory of Computing*, pages 681–690. 2008.

Robert D. Kleinberg. Nearly tight bounds for the continuum-armed bandit problem. In *Neural Information Processing Systems*. 2004.

Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2011.

Marin Kobilarov. Cross-entropy randomized motion planning. In *Robotics: Science and Systems*. 2011.

Marin Kobilarov. Cross-entropy motion planning. *International Journal of Robotics*, 2012.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, pages 282–293. 2006.

Andrey Kolobov, Mausam, and Daniel Weld. LRTDP versus UCT for online probabilistic planning. 2012.

J. Zico Kolter and Andrew Ng. Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 521–528. 2009.

Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

Tze L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.

A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential Monte-Carlo methods. In *Advances in Neural Information Processing Systems*. 2007.

Lihong Li, Michael L. Littman, and Christopher R. Mansley. Online exploration in least-squares policy iteration. In *Autonomous Agents and Multiagent Systems*, pages 733–739. 2009.

Michael L. Littman. A tutorial on partially observable Markov decision processes. *Journal of Mathematical Psychology*, 53(3):119–125, 2009.

Michael L. Littman, Thomas Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Uncertainty in Artificial Intelligence*, pages 394–402. 1995.

Chenggang Liu and Christopher G. Atkeson. Standing balance control using a trajectory library. In *International Conference on Intelligent Robots and Systems*, pages 3031–3036. 2009.

IanR. Manchester, Uwe Mettin, Fumiya Iida, and Russ Tedrake. Stable dynamic walking over rough terrain. In *Robotics Research*, volume 70, pages 123–138. Springer Berlin Heidelberg, 2011.

Shie Mannor, Dori Peleg, and Reuven Rubinstein. The cross entropy method for classification. In *International Conference on Machine learning*. 2005.

Shie Mannor, Reuven Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *International Conference on Machine Learning*, pages 512–519. 2003.

Chris Mansley, Ari Weinstein, and Michael L. Littman. Sample-based planning for continuous action markov decision processes. In *International Conference on Automated Planning and Scheduling*, pages 335–338. 2011.

L. Margolin. On the convergence of the cross-entropy method. *Annals of Operations Research*, 134:201–214, 2005.

José Antonio Martín H. and Javier De Lope. Ex $< a >$: An effective algorithm for continuous actions reinforcement learning problems. In *Industrial Electronics Society*, pages 2063–2068. 2009.

David Mayne. A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems. *International Journal of Control*, 3(1):85–95, 1966.

Andrew Moore. *Efficient Memory-based Learning for Robot Control*. Ph.D. thesis, Computer Laboratory, University of Cambridge, 1990.

Andrew W. Moore and Christopher G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances in Neural Information Processing Systems 5*, pages 263–270. 1993.

Andrew W. Moore and Jeff G. Schneider. Memory-based stochastic optimization. In *Neural Information Processing Systems*, pages 1066–1072. 1995.

Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the Association for Computing Machinery*, pages 113–126, 1976.

Ali Nouri and Michael L. Littman. Multi-resolution exploration in continuous spaces. In *Neural Information Processing Systems*, pages 1209–1216. 2008.

Dirk Ormoneit and Saunak Sen. Kernel-based reinforcement learning. In *Machine Learning*, pages 161–178. 1999.

Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

Jason Pazis and Michail Lagoudakis. Binary action search for learning continuous-action control policies. In *Proceedings of the 26th International Conference on Machine Learning*, pages 793–800. Montreal, 2009.

Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the Association for Computing Machinery*, 25:559–564, 1982.

PyODE. version 2010-03-22. 2010. URL `http://pyode.sourceforge.net/`.

Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In *International Conference on Automated Planning and Scheduling*, pages 202 – 209. 2011.

Jette Randlov and Preben Alstrom. Learning to drive a bicycle using reinforcement learning and shaping. In *International Conference on Machine Learning*. 1998.

Nathan Ratliff, Matthew Zucker, J. Andrew (Drew) Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *International Conference on Robotics and Automation*. 2009.

Philipp Reist and Russ Tedrake. Simulation-based LQR-trees with input and state constraints. In *International Conference on Robotics and Automation*, pages 5504–5510. 2010.

Ioannis Rexakis and Michail G. Lagoudakis. Classifier-based policy representation. In *International Conference on Machine Learning and Applications*, pages 91–98. 2008.

Jacques Richalet. Model predictive heuristic control: Applications to industrial processes. *Automatica*, 14(5):413 – 428, 1978.

Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

Reuven Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operations Research*, 99:89–112, 1997.

Reuven Y. Rubinstein. The cross entropy method for combinatorial and continuous optimization. *Methodology And Computing In Applied Probability*, 1:127–190, 1999.

Juan Carlos Santamaría, Richard S. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 1996.

Jonathan Schaeffer. *One Jump Ahead: Computer Perfection at Checkers*. Springer Science and Business Media, LLC, 2009.

Jonathan Schaeffer, Yngvi Bjornsson, Neil Burch, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu, and Steve Sutphen. Solving checkers. In *International Joint Conference on Artificial Intelligence*, pages 292–297. 2005.

Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

Colin Schepers. *Automatic Decomposition of Continuous Action and State Spaces in Simulation-Based Planning*. Master's thesis, Maastricht University, 2012.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In *Proceedings of the International Conference on Artificial Neural Networks ICANN*. 2008.

Claude Shannon. XXII. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275, 1950.

Bruno O. Shubert. A sequential method seeking the global maximum of a function. *Siam Journal on Numerical Analysis*, 9, 1972.

D. Silver, R. Sutton, and M. Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th Annual International Conference on Machine Learning*, pages 968–975. 2008.

Eduardo D. Sontag. *Mathematical Control Theory*. Springer-Verlag, 1998.

Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *International Conference on Machine Learning*, pages 1015–1022. 2010.

Martin Stolle and Chris Atkeson. Policies based on trajectory libraries. In *International Conference on Robotics and Automation*. 2006.

Alexander L. Strehl and Michael L. Littman. An empirical evaluation of interval estimation for Markov decision processes. In *Tools with Artificial Intelligence (ICTAI-2004)*. 2004.

Freek Stulp and Olivier Sigaud. Path integral policy improvement with covariance matrix adaptation. In *International Conference on Machine Learning*. 2012.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, pages 1057–1063. 1999.

Istvan Szita and A. Lörincz. Learning Tetris using the noisy cross-entropy method. *Neural Computation*, 18(12):2936–2941, 2006.

Istvan Szita and Csaba Szepesvári. sztetris-rl Library. `http://code.google.com/p/sztetris-rl/`, 2010.

Y. Tassa and E. Todorov. Stochastic complementarity for local control of discontinuous dynamics. In *Proceedings of Robotics: Science and Systems*. Zaragoza, Spain, 2010.

Yuval Tassa, Tom Erez, and William Smart. Lagrangian analysis of the swimmer dynamical system. 2007a. URL `http://homes.cs.washington.edu/~tassa/papers/SDynamics.pdf`.

Yuval Tassa, Tom Erez, and William Smart. Receding horizon differential dynamic programming. In *Advances in Neural Information Processing Systems 20*, pages 1465–1472. 2007b.

Yuval Tassa, Tom Erez, and William Smart. The swimmer dynamical system. 2007c. URL `http://www.cs.washington.edu/homes/tassa/code/swimmer_package.zip`.

Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *International Conference on Intelligent Robots and Systems*, pages 4906–4913. 2012.

Russ Tedrake. LQR-trees: Feedback motion planning on sparse randomized trees. In *Proceedings of Robotics: Science and Systems*, pages 17–24. 2009.

Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8, 1992.

Gerald Tesauro. Temporal difference learning and TD-gammon. *Communications of the Association for Computing Machinery*, 38(3):58–68, 1995.

Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using Monte-Carlo search. In *Neural Information Processing Systems*, pages 1068–1074. 1996.

Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. Reinforcement learning of motor skills in high dimensions: A path integral approach. In *International Conference on Robotics and Automation*, pages 2397–2403. 2010.

Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School*. 1993.

John Tromp and Gunnar Farnebäck. Combinatorics of Go. In *Proceedings of the 5th international conference on Computers and games*, pages 84–99. 2006.

L. G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

Guy Van den Broeck and Kurt Driessens. Automatic discretization of actions and states in Monte-Carlo tree search. In *International Workshop on Machine Learning and Data Mining in and around Games*. 2011.

Hado Van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. In *Institute of Electrical and Electronics Engineers International Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007*, pages 272–279. 2007.

Guido van Rossum and Jelke de Boer. Linking a stub generator (AIL) to a prototyping language (Python). In *EurOpen Conference Proceedings*. 1991.

Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*. 2010.

H.O. Wang, K. Tanaka, and M.F. Griffin. An approach to fuzzy control of nonlinear systems: stability and design issues. *Institute of Electrical and Electronics Engineers Transactions on Fuzzy Systems*, 4(1):14–23, 1996.

Yizao Wang, Jean-Yves Audibert, and Remi Munos. Infinitely many-armed bandits. In *Proceedings of Advances in Neural Information Processing Systems*, volume 21, pages 1729 – 1736. MIT Press, 2008.

Ari Weinstein and Michael L. Littman. Bandit-based planning and learning in continuous-action markov decision processes. In *International Conference on Automated Planning and Scheduling*, pages 306–314. 2012.

Ari Weinstein and Michael L. Littman. Open-loop planning in large-scale stochastic domains. In *Association for the Advancement of Artificial Intelligence*. 2013.

Ari Weinstein, Michael L. Littman, and Sergiu Goschin. Rollout-based game-tree search outprunes traditional alpha-beta. *Journal of Machine Learning Research Workshop and Conference Proceedings*, 14:155–167, 2012.

Ari Weinstein, Chris Mansley, and Michael L. Littman. Sample-based planning for continuous action markov decision processes. In *International Conference on Machine Learning Workshop for Reinforcement Learning and Search in Very Large Spaces*. 2010.

Shimon Whiteson, Brian Tanner, and Adam White. The reinforcement learning competitions. *AI Magazine*, 31(2):81–94, 2010.

Ronald Williams and Leemon C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. In *Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems*. 1994.

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

Dmitry S. Yershov and Steven M. LaValle. Sufficient conditions for the existence of resolution complete planning algorithms. In *Workshop on the Algorithmic Foundations of Robotics*, volume 68, pages 303–320. Springer, 2010.

Sungwook Yoon, Alan Fern, and Robert Givan. FF-replan: A baseline for probabilistic planning. In *Seventeenth International Conference on Automated Planning and Scheduling*. 2007.

Hakan L. S. Younes, Michael L. Littman, David Weissman, and John Asmuth. The first probabilistic track of the international planning competition. In *Journal of Artificial Intelligence Research 24, Pages 851-887*. 2005. URL `http://public.jair.fetch.com/media/1880/live-1880-2554-jair.pdf`.

Zahra Zamani, Scott Sanner, and Cheng Fang. Symbolic dynamic programming for continuous state and action mdps. In *Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence*. 2012.