LYAPUNOV METHODS FOR SAFE INTELLIGENT AGENT DESIGN

A Dissertation Presented

by

THEODORE J. PERKINS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2002

Department of Computer Science

© Copyright by Theodore J. Perkins 2002

All Rights Reserved

LYAPUNOV METHODS FOR SAFE INTELLIGENT AGENT DESIGN

A Dissertation Presented

by

THEODORE J. PERKINS

Approved as to style and content by:

____

Andrew G. Barto, Chair

____

Roderic A. Grupen, Member

____

Christopher V. Hollot, Member

____

Paul E. Utgoff, Member

____

W. Bruce Croft, Department Chair
Department of Computer Science

This dissertation is dedicated to my parents:

Ann W. Perkins
David N. Perkins, Jr.

I hope I can be as good a parent
as they each have been for me.

ACKNOWLEDGMENTS

Completing a dissertation is hard—much harder than I expected almost eight years ago when I was 22 and fresh out of college. For my eventual success I am indebted to many people. My advisor, Andy Barto, has spent many years guiding my research, and teaching me how to do research and how to write. My other committee members, Rod Grupen, Kris Hollot, and Paul Utgoff all provided valuable feedback at various stages of my thesis work, and pushed me to improve the thesis in terms of both conceptual content and presentation. Sascha Engelbrecht, while a post-doc in my lab, first introduced me to the concept of a Lyapunov function. Before leaving to pursue other interests, he also co-authored the grant that has supported me for the past two years—National Science Foundation grant no. ECS-0070102.

Other members of the Adaptive Networks laboratory (lately, the Autonomous Learning Lab) have provided a stimulating research environment during my time here. In my unbiased opinion, ours has always been one of the most exciting research groups in the department. Everyone here is both open and eager to discuss research issues, as well other issues of importance such as darts and golf.

Finally, I must mention Doina Precup, who has been instrumental in helping me to complete this dissertation. She has always been willing to lend her expert ear to my latest research ideas. She has helped me by critiquing papers I have written and presentations I have made on the topics to be found here as well as on other work. She has probably proof-read as many pages of this document as I have, and even helped me to implement some of the necessary changes. Last, and not least, she has selflessly and tirelessly taken care of our daughter Maggie far more than her fair share, especially in recent months. I have dedicated this dissertation to my parents. My next book will be for her.

ABSTRACT

LYAPUNOV METHODS FOR SAFE INTELLIGENT AGENT DESIGN

MAY 2002

THEODORE J. PERKINS

B.A., CARLETON COLLEGE

M.Sc., UNIVERSITY OF WISCONSIN - MADISON

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

In the many successful applications of artificial intelligence (AI) methods to real-world problems in domains such as medicine, commerce, and manufacturing, the AI system usually plays an advisory or monitoring role. That is, the AI system provides information to a human decision-maker, who has the final say.

However, for applications ranging from space exploration, to e-commerce, to search and rescue missions, there is an increasing need and desire for AI systems that display a much greater degree of autonomy. In designing autonomous AI systems, or agents, issues concerning safety, reliability, and robustness become critical. Does the agent observe appropriate safety constraints? Can we provide performance or goal-achievement guarantees? Does the agent deliberate and/or learn efficiently and in real time?

In this dissertation, we address some of these issues by developing an approach to agent design that integrates control-theoretic techniques, primarily methods based on Lyapunov functions, with planning and learning techniques from AI. Our main approach is to use control-theoretic domain knowledge to formulate, or restrict, the ways in which the agent can interact with its environment. This approach allows one to construct agents that enjoy provable safety and performance guarantees, and that reason and act in real-time or anytime fashion. Because the guarantees are established based on restrictions on the agent's behavior, specialized "safety-oriented" decision-making algorithms are not necessary. Agents can reason using standard AI algorithms; we discuss state-space search and reinforcement learning agents in detail. To a limited degree, we also show that the control-theoretic domain knowledge needed to ensure safe agent behavior can itself be learned by the agent, and need not be known a priori. We demonstrate our theory with simulation experiments on standard problems from robotics and control.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In recent years, researchers in artificial intelligence (AI) have paid increasing attention to issues of safety and reliability [100, 80, 77, 59, 28, 68, 67]. These issues have become especially important since the rise of the agent-based view in AI, in which the goal is to produce an intelligent system capable of operating autonomously in complex, stochastic domains where there is potential for real harm as well as real good. If a chess-playing program plays a bad move, no one is going to get hurt. However, a malfunctioning navigation system can scuttle a multi-million-dollar space mission, and a malfunctioning autopilot for a car could cause a serious traffic accident. Safety and reliability are concerns especially for learning systems, which may come to behave in ways not expected by the designer and which must "try things out" in order to learn that they are bad.

There have been many successful applications of AI techniques to real-world problems ranging from medical diagnosis, to loan approval, to space mission planning and monitoring. However, in most applications the AI system has played an advisory role, with a human decision-maker having the final say [74, 1]. For both practical and aesthetic reasons, there is much interest in developing AI systems with greater autonomy. For example, one of the goals of NASA's recently-completed Deep Space One mission was to test new AI software that allowed the spacecraft to orient itself, navigate, and recover from simulated component failures rapidly and autonomously. By imbuing spacecraft with greater autonomy, NASA scientists hope to be able to support many more missions with fewer supervising personnel, and to support missions far from earth that require rapid decision-making, such as wide-ranging planetary exploration [21]. However, one need not look to domains as exotic as space exploration to observe the drive towards fielding intelligent autonomous agents. For example, Thrun et al. [92] report on a two-week experiment in which a tour-guide robot operated autonomously at the Smithsonian Museum of Natural History, guiding visitors and explaining exhibits. The safety challenges facing this robot included not running into people or exhibits—a difficult task in the crowded (and sometimes adversarial) museum environment—and not falling down an open escalator, which could be catastrophic for the robot as well as anyone standing below. In this thesis we address the issue of applying AI decision-making techniques while ensuring safety and reliability.

In fields more oriented toward producing and controlling real systems, such as hardware and software engineering, manufacturing design, and operations research, safety and reliability have always been important issues. This thesis borrows particularly from techniques developed in applied mathematics, control theory, and robotics for dealing with continuous-state and time control problems. Historically, and even to a significant degree today, people working on such control problems have often been concerned with qualitative goals, such as stabilizing a system around an operating point; keeping the system state in some range; tracking a desired system trajectory; or smoothing out or recovering from external disturbances to the system [83, 48, 96, 19]. In many important cases, these control problems can be solved analytically. This results not just in a control strategy that provably satisfies desirable performance goals, but also in valuable qualitative domain knowledge about how the problem can be solved.

The approach we pursue in this thesis is a fusion of qualitative control-theoretic ideas with AI techniques for sequential decision-making to achieve safe, reliable optimal or approximately-optimal control. We formalize what we mean by 'safe and reliable' in Section 3.3. Roughly speaking, we are interested in questions such as:

Does the agent always achieve its goal? Can the agent always achieve its goal? Does the agent keep the system it controls in a safe, acceptable subset of state space? For continuing tasks, is the long-term or asymptotic behavior of the agent reasonable? At the same time we want the agent's solution to be as cost-effective as possible.

For general infinite-state systems, questions such as these are undecidable, so it is beyond the capacity of any computational approach, AI or otherwise, to determine or ensure such properties. However, we show that it is possible to do so for certain classes of problems using domain knowledge and analytical techniques from mathematics, control theory, and robotics. We rely on a number of techniques to design agents, including feedback linearization, linear approximations to non-linear dynamics, and linear-quadratic regulator methods. Our focus is on Lyapunov-based methods. Lyapunov functions are a special form of domain knowledge that encode qualitative information about the connectivity of a state space, and are widely used for stability and convergence analyses [48, 96].

The primary approach we advocate is to incorporate Lyapunov domain knowledge into the very formulation of the problem posed to the agent. Based on the Lyapunov properties of the problem formulation and analyses of individual AI algorithms, we demonstrated how it is possible to establish that the agent can and will solve the problem posed, and that the solution generated will be safe. The list below summarizes the thesis chapter-by-chapter and identifies its major contributions. Novel material begins in Chapter 5 with definitions of Lyapunov functions that are useful for reasoning about AI algorithms. Chapter 6 introduces several example problems and provides Lyapunov analyses. These problems are used to demonstrate the process of taking Lyapunov domain knowledge into account during the problem formulation phase, and are used in simulation experiments in subsequent chapters. Chapters 7, 8, and 9 contain the main, new theoretical and experimental results of the thesis.

Thesis Outline

Ch. 1 Introduction.

Ch. 2 Related work is presented, briefly describing the state of the art in establishing safety and performance guarantees for AI algorithms, and related issues. Key concepts and approaches from control theory are also discussed.

Ch. 3 We present the optimal control framework we use to formally define the tasks AI agents face. We discuss models for an agent's environment, what constitutes a solution to an optimal control problem, what types of safety properties can be established using methods based on Lyapunov functions, and why standard optimal control algorithms alone are not sufficient for the task of ensuring safe control.

Ch. 4 We describe standard numerical methods for solving optimal control problems and their strengths and shortcomings.

Ch. 5 The reader is introduced to Lyapunov functions in general, and we define types of Lyapunov functions and descent properties that are useful for reasoning about and designing AI systems.

Ch. 6 We present several example problems that we use to demonstrate the general Lyapunov-based theory we develop, and that are the basis for simulation experiments in later chapters. In particular, we describe a deterministic pendulum swing-up and balance problem, a stochastic pendulum swing-up and balance problem, and a robot arm control problem with deterministic and stochastic variants.

Ch. 7 In the context of state-space search, we use Lyapunov functions to establish the existence of solutions and to provide sufficient conditions under which both optimal and suboptimal heuristic search algorithms find solutions. Experiments in the deterministic pendulum and robot arm domains are presented.

Ch. 8 We establish guarantees for much broader classes of agents, basing results almost entirely on Lyapunov properties of the problem formulation and not on the algorithm used by the agent. We particularly target reinforcement learning agents, due to the proven success of reinforcement learning methods in approximately solving optimal control problems and because of the inherent difficulties (and hence challenge) of analyzing such agents. Experiments are presented using all four of the problems introduced in Chapter 6.

Ch. 9 In the chapters above we assume that the agent faces a known problem and that a Lyapunov function for the problem is known. In Chapter 9, we partly lift this assumption by supposing that the designer of the AI system hypothesizes a set of Lyapunov function candidates. We describe an active exploration strategy for determining which of these (if any) are Lyapunov functions, and bound the total loss incurred by the learning process. We demonstrate the approach using a pendulum swing-up task in which the length of the pendulum is unknown to the agent.

Ch. 10 We draw conclusions and discuss directions for future work.

CHAPTER 2

RELATED WORK

In this chapter we describe previous work on establishing safety and performance guarantees for AI systems/controllers. In Section 2.1 we describe work on these topics from the AI community. In Section 2.2 we discuss some related ideas from control theory. We conclude in Section 2.3 by relating our uses of Lyapunov functions to research on methods for expressing domain knowledge in AI systems.

2.1 Safety and Reliability in Artificial Intelligence

Weld and Etzioni, in their paper "The First Law of Robotics: A Call to Arms," were among the first in the recent history of AI to stress the importance of safety considerations if one intends to loose autonomous, intelligent agents upon the world [100]. The primary concern of their paper was how "harm" could be formalized in a planning framework and how the potential for harm should be reconciled with the agent's desire to satisfy its goals. They defined two types of safety constraints: "do-not-disturb," meaning that the agent absolutely should not change some component of the state of its environment, and "restore," meaning that the agent is allowed to change part of the state of its environment, as long as the change is undone by the time the agent finishes its work. Related to Weld and Etzioni's restore condition is the maintainability property defined by Nakamura et al. [56]. *Maintainability* is the ability of an agent with partial control over a system to return the system to a desired set of states if it is taken out of that set by other agents/extraneous influences. While safety concerns seem not to have stirred much interest in pure planning research, there has been growing interest in methods for combining constraint-based reasoning with decision-theoretic reasoning (see, e.g., Walsh [97] for one example and further references). The main idea of this work is that constraints provide absolute guarantees on performance, while decision-theoretic planning optimizes behavior within those constraints.

Safety and performance guarantees are of great interest to researchers that study learning agents for at least two reasons. First, the behavior of learning agents is often difficult to predict, even in abstract, theoretical settings. Second, learning agents typically explore both successful and unsuccessful behaviors in order to learn to separate one from the other. In on-line learning scenarios, it is often important that the agent not spend too much time exploring unsuccessful behaviors.

Singh et al. [80] provide the first example of using Lyapunov domain knowledge to design safe reinforcement learning agents. They study robot motion planning problems—kinematic problems in which an agent chooses the direction in which the robot should move next. Connolly and Grupen had previously studied the application of Lyapunov function methods to this problem [18], using harmonic functions. There are two natural types of harmonic function for this problem. Singh et al. [80] propose a formulation of the problem in which movement directions that the learning agent can choose are all convex combinations of the directions suggested by these two types of Lyapunov functions. Restricting action choice in this way ensures that the agent does not collide with objects in its workspace and that it reaches a specified goal configuration. In general, combining two different Lyapunov functions does not retain the beneficial properties of either one. In Chapter 8 we explore a related approach which ensures safety using a single Lyapunov function. In this approach the agent chooses among a set of actions that all descend on the same Lyapunov function, but in different ways.[1]

Huber and Grupen [33, 34, 35] apply reinforcement learning to quadrupedal robotic walking problems in which the system state is defined by the status of a set of lower-level closed-loop controllers. Reinforcement learning is used to choose which low-level controllers to activate, but actions that violate basic safety constraints (such as lifting all four legs at once) are eliminated from the set of admissible actions from the beginning. In spirit, this is similar to our approach; domain knowledge is used to constrain the agent to a set of safe behaviors.

Kretchmar [44] has recently proposed a method that combines reinforcement learning with robust control theory to achieve safe learning of controllers for systems with partially-unknown dynamics. In simulation, his approach works well for realistic regulation problems. However, the forms of system dynamics and learning agent that are allowed by his theory are limited. In Chapter 9 we propose an alternative approach that allows much broader classes of system dynamics and learning agents, but which offers weaker guarantees. Gordon [28] has proposed a method in which each control improvement suggested by a learning component is first verified for safety using model-checking techniques. Like the work of Kretchmar, her approach uses on-line verification to ensure that learning never results in an unsafe system.

Schneider [77] and Neuneier and Mihatsch [59] propose learning methods that attend to the variability of outcomes, resulting in learning controllers that are risk averse. Although there are no theoretical guarantees on the performance of these learning algorithms, their goal of safer, more reliable learning matches the general theme of the work mentioned above.

[1] The idea that there is more than one way to descend on a given Lyapunov function, and that this freedom can be used to optimize secondary criteria, is not a novel idea, dating back at least to Kalman's and Bertram's early survey of Lyapunov function methods [36]. Its application in reinforcement learning is new, as far as we know.

Other work in reinforcement learning has focused on reliability issues more than safety. Koenig [42] proved the first result that provides a guarantee on the performance of an agent during learning.[2] He studied finite-state minimum cost-to-goal control problems, and showed that a Q-Learning agent solving such a problem is guaranteed to find a goal state in time bounded by a polynomial in the size of the state set. In ground-breaking work, Kearns and Singh [40, 41] showed that with high-probability, a learning agent can find a near-optimal policy in a finite Markovian decision problem in time that is polynomial in the number of states. (See Brafman and Tennenholtz [16] for a simpler proof of some of these results and an extension to two-player constant-sum Markov games.) Kearns and Koller [37] proved that for problems represented in a compact, factored form, high-probability near-optimal learning is possible after an amount of experience that is polynomial in the size of the compact description. This description is potentially exponentially smaller than the number of problem states. (Note that it is not known whether the computations necessary for choosing the right polynomial amount of experience can themselves be achieved in polynomial time.) While important theoretically, these results have not yet had a strong impact on how reinforcement learning is usually put into practice, since even polynomial time complexity is large for the applications that reinforcement learning researchers would like to solve. The results also do not address continuous-state problems. Although similar reasoning was used to establish that near-optimal control can be achieved in problems with general state sets by using forward search, the search trees required by the theory are impractically large for most problems of interest [38, 39].

Still farther afield, but important to note because of the example domains we consider, are the strong theoretical results that have been obtained for dynamic pro-

[2] Note that the date of publication is at least five years later than when the result was initially obtained.

gramming approaches to solving continuous-state, continuous-time optimal control problems described by systems of differential equations [47, 25, 8, 9]. These methods work by laying a grid of discrete points over a compact subset of the state space and solving a finite-difference approximation to the control problem, similar to how partial differential equations are often solved numerically. The general character of the theoretical results obtained for this type of approach is that as the resolution of the grid grows arbitrarily fine, the value functions or policies computed by the finite-difference approximation approach the optimal value functions or policies for the continuous-state and time control problem. In practice, one often selects a grid that is as fine as memory and computational limits allow. Simulation experiments can help determine whether the solution computed is of satisfactory quality.

Pareigis [61, 62] and Munos [54] have extended these ideas to reinforcement learning systems that start with no prior knowledge of the system dynamics. However, even with the best adaptive resolution tricks these grid-based do not scale well to high dimensional problems. For example, state spaces of the demonstration domains used in the most recent work of Munos [54, 55] have dimension no more than five. We study a six-dimensional robot arm control problem in Chapters 7 and 8. In pilot experiments, we attempted to solve the problem using simple uniform-spaced grids. Even using millions of grid-points, solutions were of poor quality or failed outright to bring the arm to desired goal configurations.

2.2 Control Theory

For much of its history, and to a considerable extent today, control theory has been concerned first and foremost with safety and reliability. Typical problems include stabilization, trajectory tracking, and keeping the state of a system in some range. Techniques for achieving tasks such as these include Lyapunov-based methods, feedback-linearization, proportional-integral-derivative control, and a host of techniques designed for controlling linear systems [83, 48, 96]. Although these techniques may produce successful stabilization or tracking controllers, such controllers do not always score well on other performance measures of interest. A prime example is feedback linearization-based control of robotic manipulators. Using feedback linearization, a robot manipulator can be moved to a desired configuration or can be made to track a desired trajectory. However, the movements are inefficient and slow, "robotic," and do not take advantage of the natural dynamics of the manipulator to move quickly or smoothly. Indeed, the whole point of feedback linearization is to eliminate natural dynamics so that the analytical derivation of a controller becomes possible.

Of course, control theorists study optimal control as well, but there one runs into the same problems as one does in AI: exactly solving optimal control problems is extremely hard. Even solving them near-optimally, usually by numerical methods, is challenging and may not retain the safety properties implicit in an optimal solution. There have been attempts to infuse analytical methods, which guarantee safe, reliable control, with greater sensitivity to cost. We mention two highlights in this genre of work that focus on Lyapunov methods.

We formally introduce Lyapunov functions in Chapter 5. For the present moment, one can intuitively think of a Lyapunov function as a distance-to-goal function. In Lyapunov optimizing feedback control, a Lyapunov function is specified, and a controller is developed that causes the system to descend on that function and approach the goal[96]. In *steepest descent*, for example, control choices are made so that the system follows the gradient of the Lyapunov function with respect to the state variables as closely as possible. (This assumes the state of the system is described by a vector of real-valued variables.) In *quickest descent*, control choices are made so that the system descends on the Lyapunov function as quickly as possible. In *minimum cost-descent control*, costs are associated with each control choice, and controls are chosen to minimize a weighted combination of immediate cost and descent on a Lyapunov function. An appropriate weighting instills some sensitivity to performance requirements, while still ensuring desirable safety properties [96]. The main method we propose in Chapter 8 for guaranteeing safety properties of a learning agent can be viewed as a form of Lyapunov optimizing feedback control. In our case, controls are not chosen to optimize some quantity involving a Lyapunov function. Rather, in the simplest case, controls are chosen to optimize estimated long-term performance subject to the constraint of descending on a given Lyapunov function.

Also important is the relatively recent development of backstepping, forwarding, and related procedures for stabilization problems, including procedures for robust stabilization problems, in which the dynamics of the system are incompletely known [45, 26, 78]. These procedures have increased the range of problems for which Lyapunov designs are possible and have improved the quality of designs for problems for which stabilization methods already existed. In particular, these procedures can be shown to construct controllers that are inversely optimal—optimal with respect to some "reasonable" performance metric. Interestingly, not all stabilizing controllers are inversely optimal. Those that are have superior stability properties, such as being able to stabilize the system despite systematic disturbances in the control signal or system dynamics. Further, the methods described by Freeman and Kokotović [26] include free variables that allow the designer to choose, within some range, what cost function the stabilizing controller should optimize. This work is quite promising, and is also important for popularizing the notion of a control Lyapunov function—a function on which the system can be made to descend by an appropriate choice of control action [82]. In Chapter 5 we define several new types of control Lyapunov functions which are particularly suited to the task of analyzing AI decision algorithms such as state-space search, dynamic programming, and reinforcement learning.

2.3 Domain Knowledge

One of the most basic discoveries of research in artificial intelligence is the important role that domain-specific knowledge plays in making problem solving tractable. One view of Lyapunov functions is that they are a form of domain knowledge. In addition to allowing safe control, they can play all the usual roles of domain knowledge: making the computations necessary for solving the problem more time or space efficient, guiding the solution process, increasing robustness (especially when unforeseen or previously unexperienced situations occur), and allowing a learning agent to perform at a satisfactory level even at the beginning of learning.

A certain amount of domain knowledge goes into formulating an optimal control problem—that is, in choosing how the state of the agent's environment is represented, how the agent interacts with the environment, and how the agent's performance is measured. In addition, specific algorithms for solving control problems can be imbued with prior knowledge in various ways. In state-space search, domain knowledge can be expressed in the definitions of the search operators and in heuristic functions [64, 74]. In Chapter 7 we use Lyapunov domain knowledge in both of these capacities.

In the field of reinforcement learning, there has been much effort in developing methods for expressing domain knowledge. Various means for giving"advice" to reinforcement learning agents have been explored [101, 95, 17, 49, 93, 50, 27, 51, 76, 2, 20]. Typically, the agent is instructed on which actions are desirable or which actions are to be avoided in certain situations. Researchers have also experimented with incorporating suitably expressed prior knowledge directly into value function approximators [93, 27, 50, 51].

Sometimes, prior knowledge can be expressed simply by the formulation of control choices for a reinforcement learning agent. As mentioned above, Singh et al. [80] suggested a parameterization of control choices for motion planning in which the controller chooses a convex combination of the gradient directions of two distinct Lyapunov functions. More generally, recent work in modularity and temporal abstraction has provided mathematical foundations for dynamic programming-based control and learning when a single control choice can cause the execution of an entire sequence of control actions [63, 69, 23]. This allows a designer to specify intelligent courses of action, which simplifies learning to solve the control problem. Our use of Lyapunov functions in the context of dynamic-programming based methods, including reinforcement learning algorithms, has this basic character of constraining (or constructing) the agent's control choices.

CHAPTER 3

OPTIMAL CONTROL FRAMEWORK

In this chapter we define the general optimal control framework we use to describe the problem faced by an agent. We define Markov processes and Markov decision processes in Section 3.1. (See Bertsekas [8] or Sutton and Barto [88] for other presentations.) In Section 3.2 we define two types of solutions to Markov decision processes, action sequences (a.k.a. open-loop controllers) and policies (a.k.a. closed-loop controllers). In Section 3.3 we define a number of general safety and reliability properties of interest, and in Section 3.4 we consider whether or not the optimal control framework presented in this chapter is sufficient for addressing safety and reliability concerns.

3.1 Markov Processes and Markov Decision Processes

A *Markov process* $\langle S, F_0, F, \Omega \rangle$ models a system evolving on an arbitrary state set S at discrete points in time. The state of the system at time $t = 0$ is $s_0 \in S$, which is determined stochastically according to start state distribution F_0. For $t \in \{1, 2, 3, \ldots\}$, the state of the system is determined by the dynamics equation:

$$s_{t+1} = F(s_t, w_t) \ ,$$

where the w_t are independent "random disturbances" distributed according to Ω. At this level of generality, rigorously defining probabilities and expectations requires a measure-theoretic treatment. The general Lyapunov-based theorems we present later

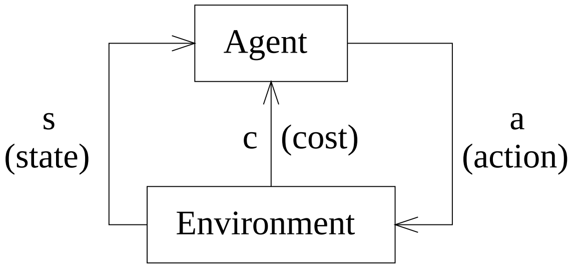

Figure 3.1. The interaction of an agent and its environment.

are phrased to be applicable in such a general setting, as long as the probabilities and expectations to which the theorems refer are well-defined. Since it is rather tangential to our topic, we do not describe the measure-theoretic foundations of Markov processes (or Markov decision processes) on general state sets. Rather, we simply assume that any quantities of interest are well-defined. We refer the reader to Bertsekas and Shreve [10] and Meyn and Tweedie [52] for background on the general case. In the example problems studied in the thesis, S is either finite or a closed, convex subset of $\mathbb{R}^n$, and the functions F_0 and F are piecewise smooth. In such cases, the random disturbances can be taken to be uniform random variables on the interval [0,1], and the well-definedness of the relevant probabilities and expectations is immediate.

We model the environment of an agent as a *Markov decision process* $\langle S, A, F_0, F, C, \Omega, G \rangle$. The interaction between the agent and its environment is depicted in Figure 3.1, and proceeds as follows. The state of the environment at time $t = 0$ is $s_0 \in S$, which is determined stochastically according to start state distribution F_0. At times $t \in \{0, 1, 2, \ldots\}$, the agent chooses an action, a_t, from a set of allowed actions, $A(s_t)$. The immediate cost that the agent incurs, $c_t \in \mathbb{R}$, and the next state of the environment, $s_{t+1} \in S$, are determined according to the dynamics equations:

$$\begin{aligned} c_t &= C(s_t, a_t, w_t) \ , \\ s_{t+1} &= F(s_t, a_t, w_t) \ , \end{aligned}$$

where w_t is a random disturbance distributed according to Ω. A particular sequence of states, actions, and costs that occurs as an agent interacts with its environment, $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots \rangle$, is called a *trajectory.* We say that a trajectory $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots \rangle$ is *possible* if there exists $\langle w_0, w_1, w_2, \ldots \rangle$ such that for all $t \in \{0, 1, 2, \ldots\}$, $c_t = C(s_t, a_t, w_t)$ and $s_{t+1} = F(s_t, a_t, w_t)$.

A *deterministic* Markov decision process is one for which the initial state, s_0, is fixed and for which the dynamics equations are deterministic. That is, the dynamics equations do not depend on a random disturbance and can be written simply as:

$$\begin{aligned} c_t &= C(s_t, a_t) \ , \\ s_{t+1} &= F(s_t, a_t) \ . \end{aligned}$$

In some cases, we are interested in controlling a deterministic system from a range of possible start states. In accordance with the definition above, we take the view that each start state defines a separate Markov process. Whether different start states really need to be treated separately depends on the solution method one intends to use. For example, in a simulated robot arm control problem presented below, we consider a set of nine start states. In Chapter 7, we use state-space search to find a separate solution for each start state. In Chapter 8, we use reinforcement learning to find a single solution that works for all nine start states.

Some Markov decision processes include *goal states*, which are terminal, or absorbing, states. If, at some time t, s_t is a goal state, then the agent incurs no further costs and there is no further opportunity for the agent to take actions. For example, if the control problem is to use a robot arm to move an object to a given location, after the object has been placed at the location, no further control is needed. In such a case, the trajectory describing the agent-environment interaction takes the form $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots, s_{t-1}, a_{t-1}, c_{t-1}, s_t \rangle$. The set of all goal states, which may be empty, is called the *goal set* and is denoted by G. A *minimum cost-to-goal* MDP is one for which G is non-empty and all immediate costs are non-negative.

3.2 Solutions to Markov Decision Problems

What constitutes a solution to an MDP? We consider two types of solutions: action sequences and policies.

We use action sequences as solutions only for deterministic, minimum cost-to-goal MDPs. For a deterministic MDP, an action sequence, $\langle a_0, a_1, \ldots, a_n\rangle$, uniquely determines a trajectory $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots, s_n, a_n, c_n, s_{n+1}\rangle$, by the rules $c_t = C(s_t, a_t)$ and $s_{t+1} = F(s_t, a_t)$, for $t \in \{0, 1, 2, \ldots, n\}$. Such an action sequence is considered to be a *solution* if $s_{n+1} \in G$. The *total cost* of an action sequence solution is the sum of the immediate costs incurred by the agent: $\sum_{t=0}^{n} c_t$. An action sequence solution is optimal if its total cost is no more than that of any other action sequence solution. For some problems, there may be no solutions—all goal states may be unreachable from the start state. On the other hand, there may be infinitely many solutions, but no optimal solution, as depicted in the example in Figure 3.2. In this diagram, the circles represent possible states of the environment. An arrow between two circles means that there is an action that causes the state of the environment to change from one state to the other. Each arrow is labeled with the immediate cost that the agent incurs upon choosing the corresponding action. The reader can observe that there are action sequence solutions of total cost: 3, 2, $1\frac{2}{3}$, $1\frac{5}{9}$, ... There is no solution that has total cost less than or equal to that of every other solution. Establishing the existence of solutions is one of the benefits of Lyapunov domain knowledge, as discussed in Chapter 7. When we discuss search algorithms for finding optimal action sequence solutions, further Lyapunov arguments or other assumptions are used to ensure that optimal solutions exist.

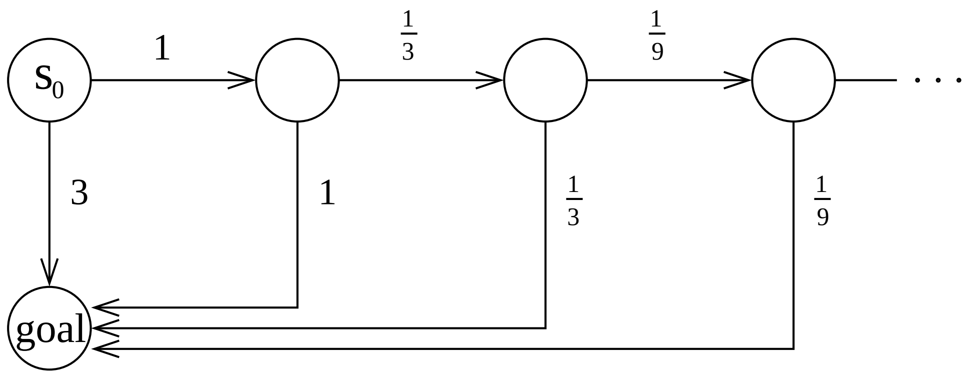

Figure 3.2. A deterministic, minimum cost-to-goal MDP with no optimal action sequence solution.

The second type of solution we consider is a policy. A policy π maps each non-goal state s to an allowed action $\pi(s) \in A(s)$. A policy completely specifies how the agent behaves.

To define what it means for a policy π to be optimal, we first define the value of π with respect to a particular start state of the environment, s. A number of different definitions of this value have been proposed. One common choice is the expected undiscounted return:

$$J_1^\pi(s) = \lim_{n \to \infty} E \left\{ \sum_{t=0}^{n} c_t \mid s_0 = s \right\} ,$$

where the expectation is with respect to the random disturbances, w_t, and is taken over all possible trajectories. Implicitly, the expectation is conditioned on the fact that the agent chooses actions according to π. On a particular trajectory, if $s_t \in G$, then for $i \geq t$, c_i should be considered zero in the sum above. The quantity $J_1^\pi(s)$ need not be finite or even well-defined in general; this issue is discussed further below.

Other common definitions for the value of a policy with respect to a start state include the expected discounted return:

$$J_\gamma^\pi(s) = \lim_{n \to \infty} E \left\{ \sum_{t=0}^{n} \gamma^t c_t \mid s_0 = s \right\} , \ \gamma \in [0, 1) ,$$

and the average cost:

$$J^{\pi}_{ave}(s) = \lim_{n\to\infty} E\left\{ \frac{1}{n}\sum_{t=0}^{n} c_t \mid s_0 = s \right\} .$$

Any of these three definitions can be applied to MDPs with or without goal states, though the limits may not be well-defined or may be infinite for some MDPs and some policies. When S is finite there are well-known sets of conditions that are sufficient to ensure well-definedness [8]. Under the same conditions, it is also known that there is always at least one policy π^* that is optimal in the sense that $J^{\pi^*}(s) \leq J^{\pi}(s)$ for any other policy π and all states s. This is true for any of the three definitions of policy value above. When S is infinite, there need not be an optimal policy even if $J^{\pi}(s)$ is well-defined for all $s \notin G$—just as there need not be an optimal action sequence for a deterministic, minimum-cost-to-goal MDP.

3.3 Safety and Reliability Properties

In this section we formally define several basic safety and reliability properties, which we use to make concrete, precise claims about the safety and reliability of different agent designs discussed in subsequent chapters. By no means should this list of properties be considered exhaustive. Rather, it reflects common kinds of safety and reliability concerns and illustrates the sort of properties that can be established using methods based on Lyapunov functions.

Suppose that an agent interacts with an environment that is modeled as an MDP with state set S. Let $T \subset S$. Recall that a trajectory can have the form $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots\rangle$ or $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots, s_{n-1}, a_{n-1}, c_{n-1}, s_n\rangle$, where $s_n \in G$. Let $I(X)$ be the set of all time indices that occur in trajectory X—either $\{0, 1, 2, \ldots\}$ or $\{0, 1, 2, \ldots, n\}$. Suppose that for some reason T is considered a set of "safe" or "desirable" states. For example, consider a driver (the agent) driving his car on a highway (the environment). The driver might consider safe states to be those in which the car in front of him is at least 5 meters away. One of the simplest questions we may ask is, "Does the state of the environment remain in T?"

Definition 3.1 *In trajectory X, the state of the environment remains in T if $s_t \in T$ for all $t \in I(X)$.*

For a deterministic MDP, an action sequence results in a unique trajectory. Thus, we can say that an action sequence is safe if the state of the environment remains in T in the resulting trajectory. For an arbitrary MDP, many trajectories may result when the agent follows a particular policy. Still, we can say that a policy is safe if the state of the environment remains in T in all possible trajectories (or perhaps for all trajectories in a probability-one subset of all possible trajectories). Even stronger, we can say that the MDP itself is safe if the state of the environment remains in T in all trajectories that are possible under any agent behavior.

Another natural question is, "Does the state of the environment reach T?"

Definition 3.2 *In trajectory X, the state of the environment reaches T if there exists $t \in I(X)$ such that $s_t \in T$.*

Note that the definition does not require that the state of the environment stay in T forever. One natural application of this notion is to the case $T = G$—that is, the question of whether the state of the environment reaches the goal. Recall that an action sequence qualifies as a solution if it causes the state of the environment to reach G. For agents seeking policies, every policy is usually considered a feasible solution to the problem. However, it is reasonable to consider that a policy be a solution to a minimum cost-to-goal problem only if the state of the environment reaches G in all trajectories that are possible under that policy. If the MDP is formulated such that the state of the environment reaches G under any agent behavior, then we have a basic reliability guarantee—that of goal-achievement.

A more basic consideration is the reachability of G. Is there an action sequence or is there a policy that results/can result in a trajectory in which the state of the environment reaches G? This question is undecidable for general, infinite state systems, and it is of practical as well as theoretical concern. For example, in continuous-state, continuous-time control problems, such as robot control problems, one often discretizes time and approximates continuous control sets with a finite number of alternatives. Whether or not a given part of state space is reachable under such a discretization becomes a non-trivial question.

Combining the two properties defined above we have:

Definition 3.3 *In trajectory X, the state of the environment reaches and remains in T if there exists $\tau \in I(X)$ such that for all $t \in I(X)$, $t \geq \tau \Rightarrow s_t \in T$.*

In the highway-driver example above, for instance, we might imagine that at some point in time the state of the environment is not in the set of "safe" states. (That is, there is less than 5 meters to the car ahead, perhaps because of a busy merge or because the car ahead decelerates suddenly.) If the driver behaves such that the reach-and-remain property holds for all possible trajectories, then he is assured of returning to a safe state of driving.

The question of whether there exists a policy that causes the state of the environment to reach and remain in T in any possible trajectory from any $s_0 \notin T$ is similar to the maintainability property studied by Nakamura et al. [56]. Maintainability additionally requires that the time to reach T be bounded over all $s_0 \notin T$.

In many problems, causing the state of the environment to reach a given T and remain there forever may be unrealistic or unachievable. A weaker condition is that the state of the environment s_t is in T for an infinite number of time steps t.

Definition 3.4 *In trajectory X, the state of the environment spends infinite time in T if $s_t \in T$ for infinitely many distinct $t \in I(X)$.*

The final property we consider describes the state of the environment converging to T over time. Let $\delta_T : S \to \mathbb{R}$ satisfy $\delta_T(s) = 0$ for $s \in T$ and $\delta_T(s) > 0$ for $s \notin T$. Intuitively, δ_T can be thought of as a "distance" to T.

Definition 3.5 *In trajectory X, the state of the environment asymptotically approaches T if* $\lim_{t\to\infty} \delta_T(s_t) = 0$.

Properties such as the first four defined above are often studied in the model-checking literature. Indeed, Gordon [28] has demonstrated the relevance of model-checking techniques for verifying learning systems. Asymptotic behavior is usually studied in control theory, and is one of the primary questions addressed by Lyapunov methods. Since our MDP framework allows for problems with infinite state sets, all of these properties are undecidable. Interestingly, even quite innocent-looking finite-dimensional continuous-state systems can implement Turing machine computations. (See Blondel and Tsitsiklis [13] for a survey of complexity results from a control theory perspective.) Nevertheless, we demonstrate that control-theoretic techniques are quite useful in establishing safety and reliability properties of agents operating in such environments.

3.4 Does the Optimal Control Framework Adequately Address Safety and Reliability?

Certainly one can define optimal control problems that reflect various kinds of safety properties. For example, suppose we are given an MDP, but that we additionally want the agent to keep the state of the environment in a set T for safety reasons. Suppose that there is at least one policy that does so, and suppose that policy has finite expected return or average cost for any initial state $s_0 \in T$. If we modify the cost function of the MDP so that actions taking the state of the environment out of T have infinite cost, then clearly the optimal policy is the best policy that does not cause the state of the environment to leave T.[1,2] For goal-achievement problems, the cost function is usually designed so that optimal behavior causes the state of the environment to reach the goal set with probability one. If not, then one would probably argue that the problem is misformulated.

However, even if we formulate an optimal control problem so that it properly reflects safety concerns, the agent still needs to solve the problem. Since the properties described in the previous section are undecidable, formulating the problem to reflect safety does not (cannot) ensure that the agent ends up with a solution that is safe. Some algorithms [47, 39, 60] promise near-optimal solutions, but by necessity, only for classes of problems for which near-optimality does not ensure safety. Formulating the problem such that optimal or near-optimal policies are safe also does not address the issue of how safely or reliably a learning agent performs during learning. For these reasons, we favor ensuring safe, reliable agent a priori, using domain knowledge.

[1]More precisely, the optimal policy keeps the state of the environment in T with probability one.

[2]Strictly speaking, we should not assign infinite cost to any action. However, we can create the same effect by redefining the dynamics of the MDP so that any action taking the state of the environment out of T brings it to a special infinite chain of states, resulting in an infinite return or average cost. For example, if costs are undiscounted, the chain of states need merely result in the cost sequence: $1, 1, 1, \ldots$ Other sequences can be designed to ensure that the discounted return or average cost are infinite.

CHAPTER 4

SOLUTION METHODS

In this chapter we discuss standard methods for solving, exactly or approximately, optimal control problems formulated as MDPs. In Section 4.1, we discuss state-space search algorithms for finding action sequence solutions. Dynamic programming-based approaches, including several reinforcement learning algorithms, are covered in Section 4.2. We close the chapter with a brief discussion of direct optimization approaches.

4.1 State-Space Search for Action Sequences

In this section, we consider state-space search methods for finding action sequence solutions to deterministic, minimum cost-to-goal MDPs [64, 74]. Recall that in such problems there is a start state s_0 and a non-empty goal set G. For each $s \notin G$ and $a \in A(s)$, the agent incurs a non-negative cost $C(s, a)$ and the state of the environment becomes $F(s, a)$. A solution is a sequence of actions that causes the state of the environment to enter G, when applied starting from s_0. A solution is optimal if the sum of the costs of the actions is no more than the total cost of any other solution. We assume that $A(s)$ is finite for all $s \notin G$.

In principle, state-space search methods are applicable to a wide range of problems, from the classical puzzle games (which were the first test applications) to certain navigation and robotics problems [74, 14]. State-space search methods are also widely used for more abstract problems, such as controlling the execution of constraint-satisfaction and planning algorithms [74]. State-space search is usually applied to

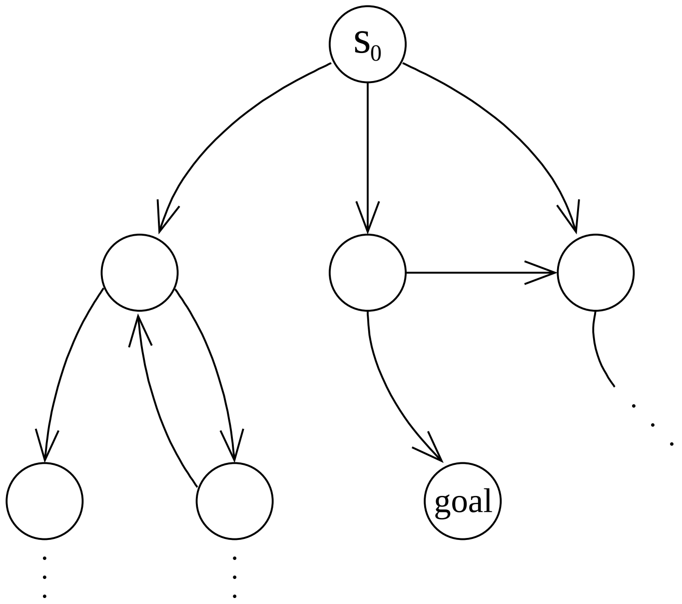

Figure 4.1. Example of a search graph.

problems with finite state sets, but is applicable to infinite state-set problems as well. We discuss several of the more important state-space search algorithms and theoretical results, and highlight some of the difficulties that can arise when the state-set of the environment is infinite.

Because there is a single initial state and a finite number of actions available from any state, only a countable[1] set of states is reachable from s_0. The *search graph*, depicted in Figure 4.1, has a vertex corresponding to each reachable state and directed edges (arcs) corresponding to the effects of actions. Search algorithms differ in how they explore the search graph, looking for a solution. We focus on four algorithms of practical and theoretical importance: uniform-cost search; best-first search, including the special case of A*; depth-first branch-and-bound; and limited look-ahead search.

Before considering solution algorithms, however, note that if there is an infinite number of reachable states, then certain problems can arise. In Section 3.2, we

[1]A set is *countable* if it is finite, or if its elements can be put in one-to-one correspondence with the natural numbers $\{1, 2, 3, \ldots\}$.

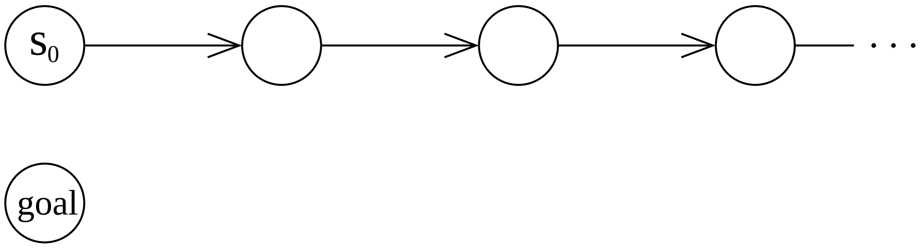

Figure 4.2. A search graph in which the goal is not reachable.

demonstrated that even if a search graph contains solutions, it may not contain an optimal solution. In this case, the optimal control problem is ill-formed. Both in finite and in infinite-state problems, it is possible that a problem has no solutions. For instance, Figure 4.2 depicts an infinite search graph with no solutions. In finite search graphs, an exhaustive search can eventually determine that there is no solution. However, in infinite search graphs, almost any standard search algorithm does not terminate. Indeed, an algorithm that is *complete*—i.e. guarantees to find a solution if one exists—cannot also terminate on all problems that have no solutions. The problem of deciding whether a solution exists is undecidable, so such a search algorithm is impossible.

In the following subsections, we present uniform-cost search, best-first search, depth-first branch-and-bound, and limited look-ahead search. We present these algorithm in forms in which the search graph is treated as if it is tree-structured. So for example, if a state s appears more than once on a path, or if two different paths lead to s, then those appearances of s are treated as if they refer to different states. This makes the algorithms easier to describe and matches our implementations of the algorithms, which we used for the experiments presented in Chapter 7. However, the theoretical results that we present in Chapter 7 apply equally well to more sophisticated versions of these algorithms which attempt to exploit graph structure for the sake of efficiency or completeness.

For a search algorithm that treats the search graph as if it is tree-structured, any state s considered during search results from a unique path through the graph from s_0. Let $\hat{g}(s_0, s)$ denote the total cost of the actions along that path. Let $h(s)$ be the cost of the optimal solution from s to some state in G, if an optimal solution exists, and $+\infty$ otherwise. The algorithms we present use a *heuristic evaluation function* $\hat{h}$, which can be any mapping from S to the non-negative reals. The function $\hat{h}$ is often thought of as an estimate of h. If $\hat{h}(s) \leq h(s)$ for all s, then $\hat{h}$ is called *admissible*. The *total evaluation* of state s is $\hat{f}(s) = \hat{g}(s_0, s) + \hat{h}(s)$. For admissible $\hat{h}$, the total evaluation of s is a lower bound on the cost of an optimal solution from s_0 to some $g \in G$ that passes through s.

4.1.1 Best-First Search and Uniform-Cost Search

There are several algorithms that go by the name "best-first search." In Figure 4.3, we present what could be called best-first search using $\hat{f} = \hat{g} + \hat{h}$. For the sake of brevity, we simply refer to it as best-first search from now on.

The algorithm maintains a list of states, Q, which it keeps ordered by increasing $\hat{f}$. On each iteration of the while loop, best-first search removes the first state, s, from the front of the list. It checks if s is a goal state. If so, then the path from s_0 to s is returned as the solution. Otherwise, the states immediately reachable from s are placed in the list, sorted according to $\hat{f}$. This process is called *expanding* state s.

The name A* refers to the special case of best-first search in which $\hat{h}$ is admissible. A* has several important theoretical properties. First, if A* returns a solution, then that solution is optimal. Second, if S is finite and if either all costs are positive or the algorithm is modified slightly to detect and avoid zero-cost loops in the search graph, then A* is guaranteed to terminate—returning a solution if any exists, and returning "no solution" otherwise. Under the same conditions, best-first search using an inadmissible $\hat{h}$ is complete and is guaranteed to terminate. However, there is no

Inputs: initial state s_0 and a heuristic evaluation function $\hat{h}$.
Outputs: a solution (path from s_0 to G) or "no solution".

Let Q be a list of states, initialized to hold s_0.
while Q is not empty **do**
  Remove the first state from Q. Call it s.
  if $s \in G$ **then**
    return the path from s_0 to s.
  else
    for all $a \in A(s)$ **do**
      Add $F(s, a)$ to Q, keeping the states in Q sorted by increasing $\hat{f} = \hat{g} + \hat{h}$.
    end for
  end if
end while
return "no solution".

Figure 4.3. Best-first search using $\hat{f} = \hat{g} + \hat{h}$.

guarantee that a solution produced by such a search is optimal. A third important property of A* is that it is *optimally efficient*. Roughly speaking, this means that A* expands no more states than any other optimal heuristic search algorithm using the same heuristic function [64].

Uniform-cost search is identical to the best-first search algorithm in Figure 4.3 except that it uses no heuristic evaluation function. States are evaluated and kept sorted simply by $\hat{g}$. Like A*, a solution returned by uniform-cost search is optimal. Uniform-cost search terminates and is complete under the same conditions mentioned above for best-first search.

If S is infinite, uniform-cost search and best-first search need not terminate, regardless of whether solutions exists (optimal or otherwise). For best-first search, the admissibility of $\hat{h}$ also does not ensure termination. The reader may verify, for instance, that uniform-cost search search does not terminate on the search graphs depicted in Figures 4.2 and 3.2. Figure 4.4 shows a search graph on which uniform-cost

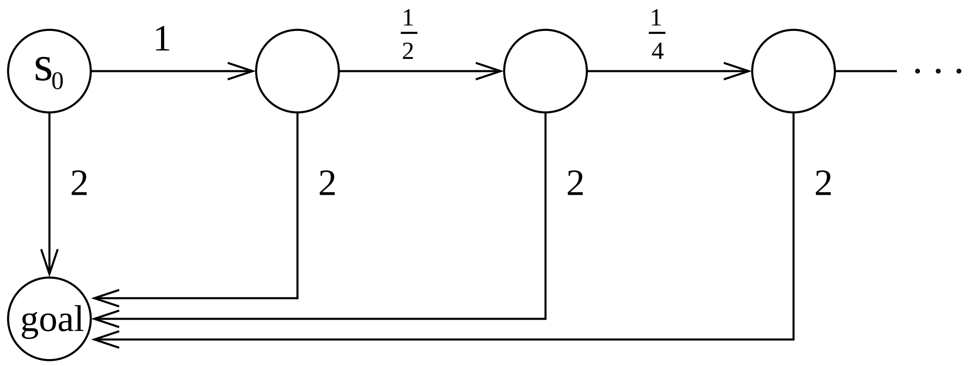

Figure 4.4. A search graph with an optimal solution on which uniform-cost search does not terminate. Arcs are labeled with the action costs.

search does not terminate, despite there being an optimal solution. Similar examples can be constructed for best-first search. What remains true in the case of infinite S is that if uniform-cost search or A* return a solution, then that solution is optimal. In Section 7.2.1, we provide sufficient conditions based on Lyapunov functions for ensuring the termination of best-first search and uniform-cost search in problems with infinite state sets.

4.1.2 Depth-First Branch-and-Bound

Best-first search maintains a list of states to expand, Q, which can grow quite large. In practice, maintaining this list in memory is often the limiting factor in solving large problems. The depth-first branch-and-bound algorithm (DFBnB), depicted in Figure 4.5, typically uses memory far more sparingly. DFBnB relies on a heuristic evaluation function, $\hat{h}$. Usually an admissible $\hat{h}$ is used, although the we allow for inadmissible $\hat{h}$ as well.

DFBnB executes a depth-first traversal of the search graph, always keeping track of the best solution it has found so far. It uses a heuristic evaluation function, $\hat{h}$, to help decide when to prune a search path. Search does not progress beyond state s if $\hat{g}(s_0, s) + \hat{h}(s) \geq C$, where C is the cost of the best solution found up to that point.

Inputs: initial state s_0 and heuristic evaluation function $\hat{h}$.
Outputs: P holds the best solution found, if any, and C holds its cost.

$P =$ "no solution".
$C = +\infty$.
DFBnBHelper(s_0)
return (P,C).

DFBnBHelper(s)
  if $s \in G$ **then**
    if $\hat{g}(s_0, s) < C$ **then**
      $P =$ the path from s_0 to s.
      $C = \hat{g}(s_0, s)$.
    end if
    return
  else
    if $\hat{g}(s_0, s) + \hat{h}(s) \geq C$ **then**
      return
    else
      for all $a \in A(s)$ **do**
        DFBnBHelper($F(s, a)$)
      end for
    end if
  end if

Figure 4.5. The depth-first branch-and-bound algorithm.

Inputs: initial state s_0 and heuristic evaluation function $\hat{h}$.
Outputs: a solution, P.

P = an empty list of actions.
while $F(s_0, P) \notin G$ **do**
  Perform a limited-complexity search, rooted at $F(s_0, P)$.
  Let s be the leaf state that minimizes $\hat{g}(F(s_0, P), s) + \hat{h}(s)$.
  Append to P the first action on the path from $F(s_0, P)$ to s.
end while
return P.

Figure 4.6. The limited look-ahead search algorithm.

The reasoning is that if $\hat{h}$ is admissible, then no solution through state s can cost less than the solution already found.

DFBnB is not guaranteed to terminate when S is finite, because the depth-first traversal may get trapped in a loop in the search graph. If DFBnB is augmented to check for revisits to a state along a path (which is a non-standard augmentation), then it terminates if S is finite. It returns "no solution" if there is no solution and returns some solution otherwise. Like A*, if $\hat{h}$ is admissible and if DFBnB returns a solution, then that solution is optimal. If S is infinite, then termination is not guaranteed even with loop-checking.

4.1.3 Limited Look-Ahead Search

Limited look-ahead search [24, 43, 74], also known as staged search and receding horizon control, is used when time or space limitations make finding an exact solution to a problem impossible. The algorithm attempts to construct a solution incrementally, one action at a time, using a series of limited-complexity searches. The algorithm is presented in Figure 4.7. The variable P holds the growing action sequence solution. The notation $F(s_0, P)$ should be interpreted as the state that

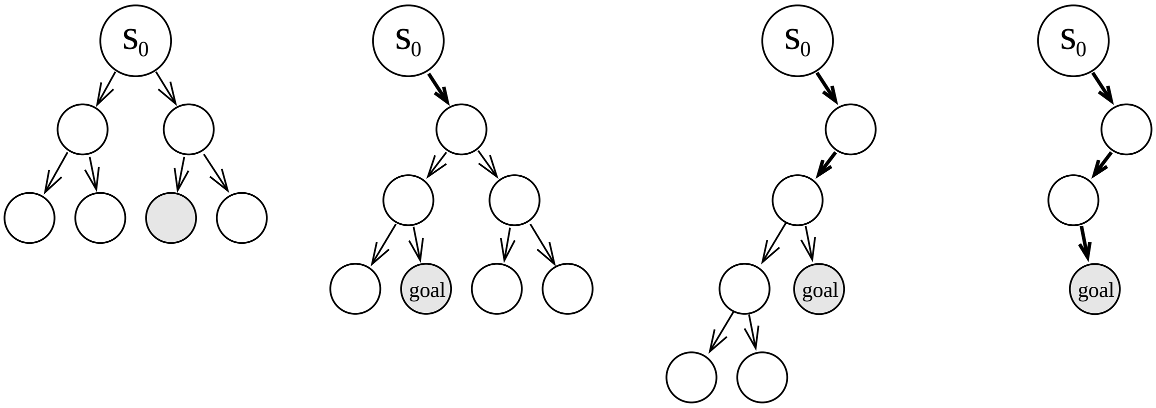

Figure 4.7. Example of a limited look-ahead search run.

results when the initial state of the environment is s_0 and the agent performs the actions listed in P. Figure 4.7 provides an example of what a run might look like. At each major iteration, a search is performed to assess which action would best extend the current partial solution. The first action on the path to the best leaf in the search tree, shaded in gray in Figure 4.7, is appended to P. This process is repeated until a complete solution is constructed (if that ever happens).

A limited-complexity search could be, for example, an exhaustive search out to some depth limit or a best-first search which terminates when a certain node limit is reached. If a depth-limited DFBnB search is used at each iteration, we call the algorithm repeated fixed-depth search (RFDS).

Limited look-ahead search is well-suited to real-time control problems, in which there are typically tight deadlines for choosing what action to take next. One can control the complexity of the searches in each major iteration in order to respond to deadlines and still, hopefully, produce a path to goal. In general, there is no guarantee that a solution will be constructed, and certainly no guarantee of producing an optimal solution. Even in settings that do not demand real-time control, limited look-ahead search is important because it allows one to achieve different trade-offs between search effort and solution quality. Generally, by putting more effort into the searches that happen at each iteration, a lower-cost solution results. One of the important theoretical contributions in Chapter 7 is providing sufficient conditions, based on Lyapunov functions, that guarantee that limited look-ahead search constructs a solution. This allows one to trade off search effort for solution quality, without worrying whether the algorithm will find a solution at all.

4.2 Dynamic Programming-Based Methods

Dynamic programming methods are more general than heuristic search methods because they can be applied to general, stochastic MDPs. We begin by describing the standard theory and algorithms for finite state-set MDPs, which are used in numerous applications in control, operations research, manufacturing, and many other fields. We then describe reinforcement learning methods, both for finite MDPs and for MDPs with state sets that are intractably-large for exact solution (including infinite). Our presentation is based mostly on Bertsekas [8], Bertsekas and Tsitsiklis [11], and Sutton and Barto [88]. Any result or idea we present in this section that is not explicitly cited can be found in one of these three books.

4.2.1 Dynamic Programming for Finite MDPs

A finite MDP is an MDP in which the state set S is finite and the set of actions available for any $s \in S$, $A(s)$, is finite. For the following discussion, we assume that in any goal state there are no actions available, and that a summation or minimization over zero actions is zero.

Recall that at time t the state of the environment s_t and the agent chooses an action $a_t \in A(s_t)$. The immediate cost that the agent incurs is $c_t = C(s_t, a_t, w_t)$ where w_t is a random disturbance. The next state of the environment is $s_{t+1} = F(s_t, a_t, w_t)$. The random disturbances are independent and identically distributed for all t. Thus, the expected immediate cost that the agent incurs when the state of the environment is s and the agent takes action a can be defined, independently of t, as:

$$c_s^a = E_w \left\{ C(s, a, w) \right\} \ ,$$

where the expectation is with respect to a random variable, w, which has the same distribution as the w_t. Similarly, one can define the probability that the next state of the environment is s' given that its present state is s and the agent chooses action a as:

$$p_{ss'}^a = Prob_w(F(s, a, w) = s') \ .$$

We consider the cases of undiscounted or discounted costs, which can be treated uniformly by allowing $\gamma \in [0, 1]$. A thorough discussion of the average cost case can be found in Puterman [70].

Recall that in Section 3.2 we defined the expected return of policy π when the environment starts in state s as:

$$J^\pi(s) = \lim_{n \to \infty} E \left\{ \sum_{t=0}^{n} \gamma^t c_t \mid s_0 = s \right\} \ .$$

We will refer to $J^\pi(s)$ as the *value* of state s under policy π. The Bellman equations allow us to express the state values recursively as:

$$J^\pi(s) = c_s^{\pi(s)} + \gamma \sum_{s'} p_{ss'}^{\pi(s)} J^\pi(s') \ , \forall s \ . \tag{4.1}$$

An optimal policy, π, is defined to be one for which $J^\pi(s) \leq J^{\pi'}(s)$ for all states s and all policies π'. In a finite MDP there is always at least one optimal policy, often denoted π^*. Its state values, J^{π^*}, or just J^*, are the unique solution to the Bellman Optimality equations:

$$J^*(s) = \min_{a \in A(s)} \left(c_s^a + \gamma \sum_{s'} p_{ss'}^a J^*(s') \right) \;\; \forall s \; . \tag{4.2}$$

A policy is optimal if and only if the state values under that policy satisfy equation 4.2. Equivalently, π is optimal if and only if, for all s, $\pi(s) \in \arg\min_{a \in A(s)}(c_s^a + \gamma \sum_{s'} p_{ss'}^a J^\pi(s'))$. If J^* is known, then an optimal policy can be constructed simply by assigning an action in $\arg\min_{a \in A(s)}(c_s^a + \gamma \sum_{s'} p_{ss'}^a J^*(s'))$ to each s.

There are various methods for computing J^*. One method is to solve the linear program:

minimize: $\sum_s J(s)$

subject to: $J(s) \geq c_s^a + \gamma \sum_{s'} p_{ss'}^a J(s')$ for all $s \in S$ and $a \in A(s)$.

where $J(s)$ for $s \notin G$ are the free variables. For $s \in G$, $J(s)$ should be taken as zero. Since linear programming problems can be solved in polynomial time, the same is true for finding optimal policies for finite MDPs.

Another method for computing J^*, along with an optimal policy, is the policy iteration algorithm shown in Figure 4.8. The algorithm starts with initial policy π_0 and computes its state values by solving the linear system of Equations 4.1. The algorithm then constructs a new policy, π_1, that is "greedy" with respect to the state values of policy π_0, computes the state values under π_1, and so on. It can be shown that $J^{\pi_i} = J^{\pi_{i-1}}$ if and only if the policies are optimal. Otherwise, it can be shown that $J^{\pi_i}(s) \leq J^{\pi_{i-1}}(s)$ for all s and that for some s', $J^{\pi_i}(s') < J^{\pi_{i-1}}(s')$. In other words, the sequence of policies generated by policy iteration improves at each step until reaching an optimal policy. Since there are finitely many deterministic policies, policy iteration terminates after a finite number of iterations.

Inputs: initial policy π_0.
Outputs: optimal state values, J^*, and an optimal policy, π^*.

$i \leftarrow 0$
Solve the system of Equations 4.1 to compute J^{π_0}.
repeat
  $i \leftarrow i + 1$
  For all s, let $\pi_i(s) \in \arg\min_{a \in A(s)}(c_s^a + \gamma \sum_{s'} p_{ss'}^a J^{\pi_{i-1}}(s'))$.
  Solve the system of Equations 4.1 to compute J^{π_i}.
until $J^{\pi_i} = J^{\pi_{i-1}}$
$J^* \leftarrow J^{\pi_i}$
$\pi^* \leftarrow \pi_i$
return (J^*, π^*)

Figure 4.8. The policy iteration algorithm.

Inputs: initial state values J_0.

for $i = 1, 2, 3, \ldots$ **do**
  For all s, let $J_i(s) = \min_{a \in A(s)}(c_s^a + \gamma \sum_{s'} p_{ss'}^a J_{i-1}(s'))$
end for

Figure 4.9. The value iteration algorithm.

For most problems, policy iteration is more efficient than the linear programming solution. However, repeatedly solving the system of equations 4.1 is still a big computational burden. The value iteration algorithm, shown in Figure 4.9, tends to be faster, and also has the advantage of having an easy implementation. The initial state values, J_0, can be any assignment of real numbers to states. A good choice is $J_0 = 0$. Successive J_i are generated by iteratively applying the Bellman Optimality equations 4.2 as update rules. Under a variety of conditions, $\lim_{i\to\infty} J_i = J^*$.

As written, the algorithm does not terminate. Of course, in reality one must terminate the algorithm at some point. There are formulas bounding the difference between J_i and J^* as a function of i or other quantities, and various rules for determining when to stop. J_i converges to J^* exponentially quickly, so one does not need to wait too long to have a good estimate of J^*. It is also not necessary to store the J_i separately. One can keep a single vector of estimates, J, and repeatedly update it as: $J(s) \leftarrow \min_{a\in A(s)}(c_s^a + \gamma \sum_{s'} p_{ss'}^a J(s'))$. If all states are updated infinitely many times, then the values converge to J^*. This approach is closest in design to the reinforcement learning methods presented in the next section.

4.2.2 Reinforcement Learning for Finite MDPs.

Linear programming, policy iteration and value iteration are all model-based algorithms—they assume the expected costs, c_s^a, and transition probabilities, $p_{ss'}^a$, are known. By contrast, many reinforcement learning algorithms assume data about the environment comes as "experiences" of the form $(s, a) \to (c, s')$, meaning that the environment was in state s and the agent took action a, which resulted in immediate cost c and next state s'. One advantage of this assumption is that it allows reinforcement learning agents to be interfaced with real or simulated environments for which the expected costs and transition probabilities are not known explicitly.

The algorithms we consider below rely on a slightly different way of writing value functions than presented in the previous section. Instead of computing the values of states, these algorithms compute the values of state-action pairs. The value of a state-action pair (s, a), denoted $Q^\pi(s, a)$, is defined as the expected return if the environment starts in state s, the agent begins by taking action a, and follows policy π thereafter [99]:

$$Q^\pi(s, a) = \lim_{n \to \infty} E\left\{ \sum_{t=0}^{n} \gamma^t c_t \mid s_0 = s, a_0 = a \right\} .$$

Q^π is related to J^π, and to itself, by a linear system of Bellman equations:

$$\begin{aligned} Q^\pi(s, a) &= c_s^a + \gamma \sum_{s'} p_{ss'}^a J^\pi(s') \\ &= c_s^a + \gamma \sum_{s'} p_{ss'}^a \sum_{a' \in A(s')} \pi(s', a') Q^\pi(s', a') \ \forall s, a \ . \end{aligned}$$

The Bellman Optimality equations for action-values, whose solution is unique in most cases, are:

$$Q^*(s, a) = c_s^a + \gamma \sum_{s'} p_{ss'}^a \min_{a' \in A(s')} Q^*(s', a') \ \forall s, a \ . \tag{4.3}$$

If Q^* is known, an optimal policy can be easily identified by letting $\pi^*(s) \in \arg\min_{a \in A(s)} Q^*(s, a)$.

One of the best-known reinforcement learning algorithms, Q-learning [99], is presented in Figure 4.10. For each experience $(s_i, a_i) \rightarrow (c_i, s_i',)$, the quantity $c_i + \gamma \min_{a' \in A(s_i')} Q(s_i', a')$ can be viewed as a possibly-biased stochastic sample, or estimate, of the return expected from taking action a_i in state s_i. In other words, it is a sample of the right-hand side of the Bellman Optimality equation (4.3). $Q(s_i, a_i)$ is modified to be closer to this estimate, with a step size parameter α_i determining the "weight" of the update. If the experience sequence contains an infinite number of independent samples of the outcomes of each state-action pair, and if the step size parameters α_i converge to zero at an appropriate rate, then the estimates Q converge to

Inputs: initial action values Q and an infinite sequence of experience $\{(s_i, a_i) \rightarrow (c_i, s'_i)\}$, $i \in \{0, 1, 2, \ldots\}$.

for i=0,1,2,... **do**
  $Q(s_i, a_i) \leftarrow (1 - \alpha_i)Q(s_i, a_i) + \alpha_i(c_i + \gamma \min_{a' \in A(s'_i)} Q(s'_i, a'))$.
end for

Figure 4.10. The Q-learning algorithm.

Q^* with probability one [88]. The relative number of samples of different state-action pairs is not important for convergence.

How is the experience used by Q-learning generated? As long as the outcome of each (s, a) is sampled infinitely many times, it does not matter from a convergence point of view. In practice, learning is often based on experience from simulated or real trajectories. In either case, intelligent selection of experience can greatly increase the speed of learning. When a real environment is being controlled and real costs are being incurred, there is even more reason for the agent to collect its experience wisely. The ϵ-greedy Q-learning algorithm combines Q-learning updates with a simple method for choosing actions (and thus generating experience) in simulation or in reality. This algorithm is presented in Figure 4.11. At each time step, with probability ϵ, a random action is selected. This ensures that each action is tried infinitely many times and that each reachable state is visited infinitely many times. With probability $1 - \epsilon$, an action currently estimated to have minimal expected long-term cost is selected. This focuses the experience along "good" trajectories and keeps down incurred costs, which can be important for on-line learning situations. Again, if the step size parameters, α_i, go to zero at an appropriate rate, Q converges to Q^*. Note that in the update equation, if $s_{i+1} \in G$, that last term is taken to be zero, so that the update is really just $Q(s_i, a_i) \leftarrow (1 - \alpha_i)Q(s_i, a_i) + \alpha_i c_i$.

Inputs: initial action values Q, exploration rate ϵ, and an environment in initial state s_0.

---

for $i = 0, 1, 2, \ldots$ **do**
  Choose action a_i:
  Let p be uniformly random in [0,1].
  if $p < \epsilon$ **then**
    Choose a_i uniformly randomly from $A(s_i)$.
  else
    Choose a_i from $\arg\min_{a \in A(s_i)} Q(s_i, a)$.
  end if

  Execute the action:
  Apply action a_i to the environment.
  Observe c_i and s_{i+1}.

  Update Q:
  $Q(s_i, a_i) \leftarrow (1 - \alpha_i)Q(s_i, a_i) + \alpha_i(c_i + \gamma \min_{a' \in A(s_{i+1})} Q(s_{i+1}, a'))$.

  Reset environment, if necessary:
  if $s_{i+1} \in G$ **then**
    Reset s_{i+1} according to a start state distribution or by some other means.
  end if
end for

Figure 4.11. The ϵ-greedy Q-learning algorithm.

Another popular reinforcement learning algorithm is Sarsa (the algorithm is due to Rummery and Niranjan [73]; the name is due to Sutton [87]). Sarsa updates action-value estimates on the basis of experiences of the form: $(s, a) \rightarrow (c, s', a')$. This notation means that the state of the environment was s, the agent chose action a, incurred immediate cost c, the next state of the environment was s', and the agent chose action a'. The Sarsa update rule for such an experience is:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(c + \gamma Q(s', a')) \ .$$

The ϵ-greedy Sarsa algorithm is just like the ϵ-greedy Q-learning algorithm except that the Sarsa update is used instead of the Q-learning update. Note that if ϵ is small, a' is usually in $\arg\min_{a \in A(s')} Q(s', a)$, thus $c + \gamma Q(s', a') = c + \gamma \min_{a \in A(s')} Q(s', a)$. That is, the Q-learning and Sarsa updates are identical most of the time. It has been shown that if α_i and ϵ are taken to zero at appropriate rates, then Q converges to Q^* under the ϵ-greedy Sarsa algorithm [79].

It is not immediately obvious why Sarsa might be preferred over Q-learning, and for finite MDPs there often is no significant difference. However, a generalization of the Sarsa algorithm, called Sarsa(λ), has variance-reduction properties which usually allow much faster learning than either Q-learning or Sarsa. Further, when generalizing function approximators, such as neural networks, are used to estimate Q-values for MDPs with large or infinite state sets, Sarsa(λ) has been observed to have much better stability properties than Q-Learning, Sarsa, and other algorithms [87, 88]: the approximations are much less likely to not converge. The next section describes how Q-learning and Sarsa may be combined with generalizing function approximators such as neural networks. The more complex Sarsa(λ) algorithm, which we use in our experiments in Chapters 8 and 9, is described in Appendix B.

There are many other important reinforcement learning algorithms. The TD and TD(λ) algorithms are methods for learning J^π for a given π based on experience

[86]. There are several versions of Q(λ) which give alternate methods for learning Q^* [99, 65]. The advantage learning algorithm estimates both J^* and Q^* [4]. There are also approaches to reinforcement learning in which experience is used to learn estimates of the MDP's immediate costs, c_s^a, and transition probabilities, $p_{ss'}^a$. From this information, optimal values or policies can be computed by the dynamic programming methods discussed in Section 4.2 [88].

4.2.3 Reinforcement Learning with Differentiable Value Function Approximators.

When an MDP's state set is very large or infinite, one simply cannot store J- or Q-values for every state. The typical reinforcement learning approach to this problem is to use a tractably-sized, parameterized function approximator such as a neural network to approximate the value function. That is, one assumes an action-value function of the form $\hat{Q}(s,a,\omega)$ where ω is a vector of free parameters to be adjusted by the learning algorithm.

When $\hat{Q}$ is differentiable in ω, a common strategy for tuning ω is to use a gradient-based version of one of the standard reinforcement learning algorithms, such as Q-learning or Sarsa. For example, consider the standard Q-learning update based on the experience $(s,a) \rightarrow (c,s')$:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(c + \gamma \min_{a' \in A(s')} Q(s',a')) \ .$$

This can also be written as:

$$\begin{aligned} Q(s,a) \quad &\leftarrow \quad Q(s,a) + \alpha(c + \gamma \min_{a' \in A(s')} Q(s',a') - Q(s,a)) \ , \\ \text{or } \Delta Q(s,a) \quad &= \quad \alpha(c + \gamma \min_{a' \in A(s')} Q(s',a') - Q(s,a)) \ . \end{aligned}$$

The right hand side, after the "α", can be viewed as an error. For Q^* the error is zero on average for all states. If the error is positive, $Q(s,a)$ needs to be increased in order to reduce the error. If the error is negative, $Q(s,a)$ should be decreased. When using a differentiable function approximator, $\hat{Q}$, one effects a change in $\hat{Q}(s,a,\omega)$ by changing ω. Thus, in the gradient-descent version of Q-learning, the free parameters are adjusted as:

$$\Delta\omega \;=\; \alpha(c + \gamma \min_{a' \in A(s')} \hat{Q}(s', a', \omega) - \hat{Q}(s,a,\omega))\nabla_\omega \hat{Q}(s,a,\omega)$$

Similarly, the gradient-descent version of a Sarsa update, based on the experience $(s,a) \rightarrow (c, s', a')$, is:

$$\Delta\omega \;=\; \alpha(c + \gamma \hat{Q}(s', a', \omega) - \hat{Q}(s,a,\omega))\nabla_\omega \hat{Q}(s,a,\omega)$$

Though this approach to approximating value functions has been shown to be unsound in general [5, 11, 94], it has generated the greatest successes of the field. For example, Tesauro's celebrated backgammon player TD-gammon used the TD(λ) algorithm with a neural network to approximate J^* in a manner similar to how Sarsa(λ) approximates Q^* [91]. The approach has also been used successfully to solve a number of dynamical-system control problems (see, e.g., Sutton [87] or Santamaria et al. [75]). In Chapters 8 and 9 we use the Sarsa(λ) algorithm along with a special kind of linear function approximator to learn good policies for pendulum and robot arm control problems.

4.3 Direct Policy Optimization

A third major class of approaches to solving MDPs is direct policy optimization. In this approach, a single numerical value is attached to each policy in some set, and optimization algorithms are used to find the best policy in that set.

Recall that in Section 3.2 we offered several definitions for the value of a policy with respect to a given initial state. A common approach to assigning a single numerical value to a policy is to average the policy's value with respect to each possible initial state, weighed by the initial state distribution of the MDP. That is, the value assigned to the policy is just the return or average cost that can be expected *before* the initial state is observed.

Much of the research in this area has focused on using optimization algorithms based on gradient-descent to search in a continuous space of stochastic policies—policies which associate a probability distribution over the allowed actions to each state. Early work of this sort in the reinforcement learning community includes the REINFORCE algorithm [102] and certain actor-critic methods (see, e.g., Gullapalli [30, 29]). Lately, there has been a resurgence of interest in this approach due to its superior convergence properties compared to the reinforcement learning approaches described in Section 4.2.3 [6, 89, 7].

Other approaches to policy optimization include evolutionary algorithms (e.g., Ram et al. [71], and Moriarty et al. [53]) and various versions of local search (e.g., Rosenstein and Barto [72], and Strens and Moore [85]).

We do not use direction optimization methods in this thesis, although we certainly could. Indeed, such approaches are often at their best when a "reasonable" class of possible policies can be identified from the outset. This is precisely what Lyapunov domain knowledge and other control-theoretic techniques provide.

CHAPTER 5

LYAPUNOV FUNCTIONS

Lyapunov functions are a fundamental tool for studying the stability and convergence of systems that evolve through time. The concept was originally devised by A. M. Lyapunov for studying the stability of systems of differential equations. Since then, Lyapunov functions have been adopted and used in many other fields. In control theory and robotics, Lyapunov arguments are often used to validate control designs by showing that the dynamics of the controlled system are stable [96, 19, 48]. Lyapunov analyses are used in designing controllers for queues in network routing problems, where the aim is to ensure stability of the network traffic (see, e.g., Basar [3]). More esoteric uses of Lyapunov functions include establishing the convergence of stochastic approximation algorithms [11] and recurrent neural networks [32].

Lyapunov functions are often pictured as energy functions. Although various definitions are used in different settings, a Lyapunov function is generally a positive, scalar function of the state of a system that decreases monotonically along system trajectories. Intuitively, the system "dissipates" the "energy" represented by the Lyapunov function. A dissipative system, of course, settles into a state of locally minimal energy—that is, it stabilizes. Another view of Lyapunov functions is as generalized distance-to-target or error functions. This view is common in robotics, for example, where Lyapunov functions are used to prove that a robot approaches a desired target configuration from any initial state or that it tracks a desired trajectory.

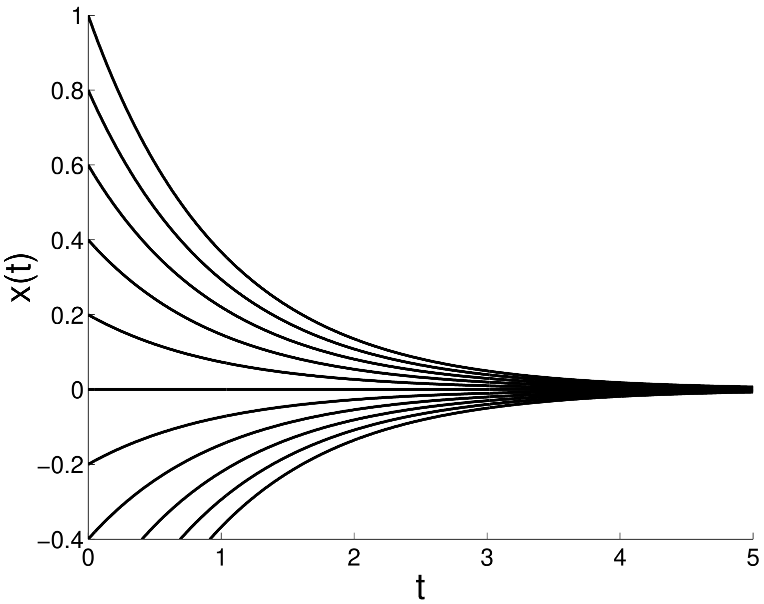

Figure 5.1. Sample trajectories of the system $\dot{x} = -x$.

5.1 Two Examples

We present several precise Lyapunov function definitions in Section 5.2. To build further intuition about Lyapunov functions, we first present stability proofs for two simple, uncontrolled, continuous-time systems.

For the first example, imagine that a particle is moving along the real line, and that its position as a function of time is denoted by $x(t)$ for $t \in [0, \infty)$. Suppose the particle starts at some $x_0 = x(0) \in \mathbb{R}$ and that the particle's motion is described by the differential equation:

$$\dot{x} = -x \ .$$

This is one of the most elementary differential equations, and can readily be solved to yield a closed formula for the particle's position at any time: $x(t) = x_0 e^{-t}$. Examining the solution, we see that for any x_0, the particle approaches the origin as $t \to \infty$, since $\lim_{t\to\infty} x_0 e^{-t} = 0$. Figure 5.1 plots trajectories of the system for several initial conditions.

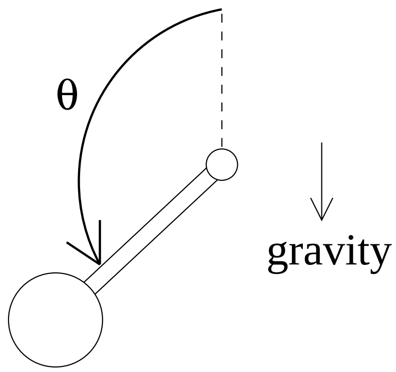

Figure 5.2. The single-link pendulum.

The same asymptotic result can be established using a Lyapunov argument. Let $L(x) = \frac{1}{2}x^2$, for $x \in I\!R$. L is the Lyapunov function we use to analyze the asymptotic behavior of the system. Define $\dot{L}(x) = \frac{dL}{dx} \cdot \frac{dx}{dt} = x \cdot (-x) = -x^2$. Since $\dot{L}(x) \leq 0$ for all $x \in I\!R$, along any trajectory $x(t)$, $L(x(t))$ is monotonically non-increasing. Since $L(x) \geq 0$ for all $x \in I\!R$, $L(x(t))$ must have some lower limit: $\lim_{t \to \infty} L(x(t)) = l \geq 0$. Lastly, since $L(x(t)) \to l$, it must be that $\dot{L}(x(t)) \to 0$. That only happens at the origin, so we can conclude $x(t) \to 0$.

One of the benefits of the Lyapunov approach is that it does not require explicitly solving the differential equation, which in general is quite difficult. The Lyapunov argument above uses only differentiation and simple limiting arguments. Of course, the Lyapunov approach does not give as much information as a complete solution to the differential equation. The Lyapunov argument identifies the limiting behavior of a trajectory, and thus provides a qualitative description of the dynamics of the system. Questions about the state of the system at precise times are not answered by the this approach.

As a second example, consider the single-link pendulum depicted in Figure 5.2. The state of the pendulum is given by an angular position, θ, and an angular velocity, $\dot{\theta}$. We can model the acceleration of the pendulum swinging under the influence of gravity and slowed by friction as:

$$\ddot{\theta} = \sin(\theta) - c\dot{\theta} \ ,$$

where c is a positive constant. This assumes the pendulum has unit length and mass and assumes a gravitational constant of 1. The $\sin(\theta)$ term is due to gravity and the $-c\dot{\theta}$ term is due to friction.

The mechanical energy of the pendulum is $\mathrm{ME}(\theta, \dot{\theta}) = 1 + cos(\theta) + \frac{1}{2}\dot{\theta}^2$, the first two terms being the potential energy and the last term being the kinetic energy. The time derivative of ME along system trajectories is $\dot{\mathrm{ME}}(\theta, \dot{\theta}) = -\sin(\theta)\dot{\theta} + \dot{\theta}\ddot{\theta} = -c\dot{\theta}^2$. So, ME is non-negative for all system states and non-increasing along all system trajectories. From any initial state, then, ME converges to some $l \in I\!R$ and $\dot{\mathrm{ME}} \to 0$. Since $\dot{\mathrm{ME}} = 0$ only when $\dot{\theta} = 0$, the pendulum must converge to a state with $\dot{\theta} = 0$. Since $(\theta, \dot{\theta}) = (k\pi, 0)$ for $k \in \{0, \pm 1, \pm 2, \ldots\}$ are the only equilibrium points with $\dot{\theta} = 0$, the pendulum must converge to one of those points. These are zero velocity states with the pendulum either perfectly upright or hanging straight down. Since the upright positions are not stable equilibria, we can conclude that for "most" initial conditions, the pendulum converges to the zero-energy, hanging-downward position.

5.2 Lyapunov Functions for Discrete-Time Processes and Control

Markov processes and MDPs describe systems evolving in discrete time on general state sets. Our definitions of Lyapunov functions are aimed at this sort of environment, and thus differ in detail from the traditional definitions of Lyapunov functions for continuous-state, continuous-time systems. Below, we define Lyapunov functions for Markov processes and control Lyapunov functions for MDPs. We also describe two types of descent: guaranteed descent and probabilistic descent. Our definition of the state of a Markov process descending on a Lyapunov function is very similar to one given in Meyn and Tweedie [52]. The other definitions are new, to our knowledge.

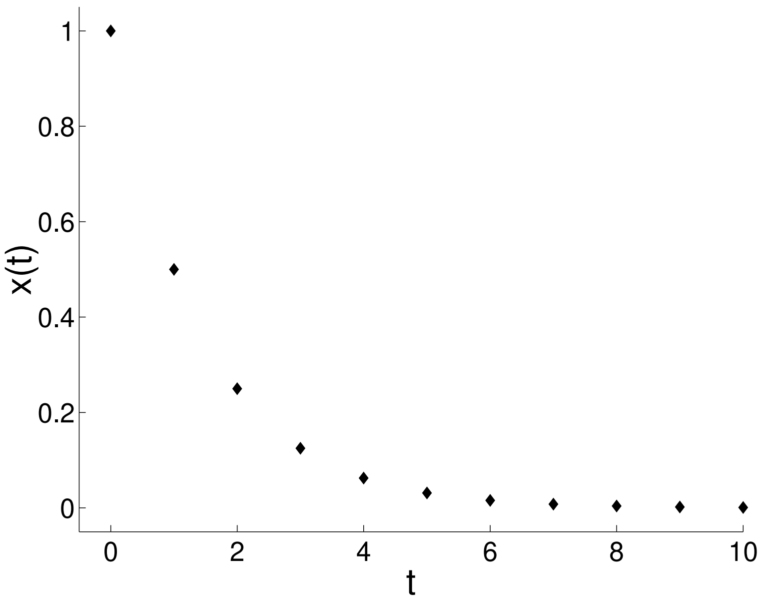

Figure 5.3. A sample trajectory of the system $x(t+1) = x(t)/2$.

Consider a Markov process with dynamics F and state set S. Let $T \subset S$ and let $L : S \to \mathbb{R}$ such that $L(s) > 0$ for all $s \notin T$.

Definition 5.1 *The state of the process descends on L outside of T if there exists $\delta > 0$ such that for all $s \notin T$ and all w, $F(s, w) \in T$ or $L(s) - L(F(s, w)) \geq \delta$.*

In other words, the state of the process descends on L if, from any $s \notin T$, the next state of the process is guaranteed either to be in T or to be lower on L by at least δ. Let us briefly consider an example.

Consider a Markov process with $S = \mathbb{R}$ and let $x(t)$ denote the state of the process at times $t = 0, 1, 2, \ldots$ Suppose the next state of the process is always given by the rule $x(t+1) = x(t)/2$. Figure 5.3 plots a trajectory of this process, starting from $x(0) = 1$. Let $L(x) = |x|$ for $x \in \mathbb{R}$ and suppose $T = \{0\}$. Does the state of this process descend on L? It does not. The state certainly approaches T from any initial condition, and $L(x(t)) \to 0$. However, the amount by which the state of the process descends on L at every step also decreases to zero, and the state never actually reaches T. If we suppose that $T = (-\epsilon, +\epsilon)$ for some $\epsilon > 0$, then the state of the process does descend on L. For this choice of T and for any $s \notin T$, the next state of the process is always in T or lower on L by at least $\epsilon/2 > 0$.

We find this guaranteed-descent definition most useful in studying deterministic systems. In most realistic stochastic systems, guaranteed descent is unlikely to hold. For this reason, we define probabilistic descent. Again, consider a Markov process with dynamics F and state set S. Let $T \subset S$ and let $L : S \to \mathbb{R}$ satisfy $L(s) > 0$ for all $s \notin T$.

Definition 5.2 *The state of the process descends probabilistically on L outside of T if there exists $\delta > 0$ and $p > 0$ such that for all $s \notin T$:*

$$Prob_w(F(s,w) \in T \text{ or } L(s) - L(F(s,w)) \geq \delta) \geq p \ .$$

In other words, on every time step the state of the process descends with probability at least $p > 0$. What happens the other $1 - p$ fraction of the time is left unspecified. We say that L is a Lyapunov function with target set T for a Markov process if the state of the process descends on L or descends probabilistically on L outside of T.

The reason we are interested in Lyapunov functions is that they provide some knowledge about how a process behaves over time. What conclusions can be drawn from knowledge of a Lyapunov function for a Markov process? The following two theorems provide answers to this question.

Theorem 5.3 *If the state of a Markov process descends on a Lyapunov function L outside of T, then from any $s_0 \notin T$, the state of the process enters T in at most $\lceil L(s_0)/\delta \rceil$ time steps.*

Proof: Suppose that the state of the process does not enter T within $k = \lceil L(s_0)/\delta \rceil$ time steps. Since the state of the process does not enter T, each successive state of the process is at least δ lower on L. Thus for $t \in \{0, \ldots, k\}$, $L(s_t) \leq L(s_0) - t\delta$.

In particular, $L(s_k) \leq L(s_0) - k\delta = L(s_0) - \lceil L(s_0)/\delta \rceil \delta \leq L(s_0) - L(s_0) = 0$. However, if $s_k \notin T$, then $L(s_k)$ must be positive. Thus, we have a contradiction. □

If one only knows that the state of a Markov process descends probabilistically on a Lyapunov function, then one cannot draw nearly as strong a conclusion. Although the state descends with probability at least p, with probability as much as $1 - p$ the state may do something totally different, such as move "far away" from T. However, if L is bounded above outside of T, so that the state of the process cannot get "too far" from T, then entering T eventually is certain.

Theorem 5.4 *If the state of a Markov process descends probabilistically on a Lyapunov function L outside of T, and if $L(s) \leq U$ for some $U \in I\!R$ and all $s \notin T$, then for any $s_0 \notin T$, the state of the process enters T eventually with probability one. The probability that the state of the process does not enter T by time τ is bounded above by a function that decays exponentially to zero in τ.*

Proof: Regardless of s_0, if the state of the process descends on L for $\lceil U/\delta \rceil$ steps in a row, then clearly it must enter T. The probability of this happening is at least $p_T = p^{\lceil U/\delta \rceil} > 0$, where p is the probability of descent on any time step. This means that the probability that the state of the process does not enter T in the first $\lceil U/\delta \rceil$ time steps is no more than $1 - p_T$. The probability that the state of the process does not enter T in the first $2\lceil U/\delta \rceil$ time steps in no more than $(1 - p_T)^2$. The probability that state of the process does not enter T in the first $k\lceil U/\delta \rceil$ times steps is at most $(1 - p_T)^k$. The probability that state of the process never enters T is $\lim_{k \to \infty}(1 - p_T)^k = 0$. This establishes both claims of the theorem. □

For MDPs there is added complexity because the environment receives a control input, or action, from the agent. We begin by defining what it means for an action to cause the state of the environment to descend, and then define control Lyapunov functions for MDPs.

Consider an MDP with dynamics F and state set S. Let $T \subset S$ and let $L : S \rightarrow I\!R$ such that $L(s) > 0$ for all $s \notin T$.

Definition 5.5 *For $s \notin T$, action $a \in A(s)$ causes the state of the environment to descend on L by an amount δ if for all w, $F(s, a, w) \in T$ or $L(s) - L(F(s, a, w)) \geq \delta$.*

Definition 5.6 *For $s \notin T$, action $a \in A(s)$ causes the state of the environment to descend on L by an amount δ with probability at least p if:*

$$Prob_w(F(s, a, w) \in T \text{ or } L(s) - L(F(s, a, w)) \geq \delta) \geq p \ .$$

Definition 5.7 *L is a control Lyapunov function (CLF) with target set T for an MDP if, for fixed $\delta > 0$, $p > 0$ and all $s \notin T$ there exists an action that causes the state of the environment to descend on L with probability at least p by amount δ.*

Definition 5.8 *If there exists $\delta > 0$ such that for all $s \notin T$, all $a \in A(s)$ cause the state of the environment to descend on L by an amount δ, then we say "all actions descend on L." If there exists $\delta > 0$ and $p > 0$ such that for all $s \notin T$, all $a \in A(s)$ cause the state of the environment to descend on L by an amount δ with probability at least p, then we say "all actions descend probabilistically on L."*

Deriving useful results from these definitions is mostly left for subsequent chapters. First, we make a few brief comments on the conditions under which Lyapunov functions exist and how they can be identified for particular systems.

5.3 Existence and Identification of Lyapunov Functions

For a Markov process, identifying a Lyapunov function establishes that the process can reach the set T (and will reach T, if descent is guaranteed or if L is bounded above.) Because for infinite-state processes, such reachability questions are undecidable in general, Lyapunov functions should be considered a powerful form of domain knowledge. This raises the question: under what conditions do Lyapunov functions exists, and how can they be found? The following discussion focuses on MDPs, though similar results hold for Markov processes.

For deterministic MDPs, it turns out that a CLF exists if and only if T is reachable from all $s_0 \notin T$. In this case, reachable means that there exists some finite sequence of actions which, applied from state s_0, brings the state of the environment to T.

Theorem 5.9 *Given a deterministic MDP with state set S and $T \subset S$, there exists a CLF L with target set T if and only if T is reachable from all $s_0 \notin T$.*

Proof: The forward implication holds because if L is a CLF, then the agent can always choose an action that descends. By reasoning similar to the proof of Theorem 5.3, T is reachable from s_0 in at most $\lceil L(s_0)/\delta \rceil$ steps. For the reverse direction, let $T_0 = T$ and define $T_i = \{s : s \notin \cup_{j=0}^{i-1} T_j$ and there exists $a \in A(s)$ such that $F(s, a) \in T_{i-1}\}$. That is, T_i is the set of states for which the minimum length path to T is of length i. Let $L(s) = \{i : s \in T_i\}$. L is well-defined since we assume T is reachable from every state, and one can easily check that L is a Lyapunov function. □

If one does not restrict attention to *deterministic* MDPs, then the existence of a CLF is not equivalent to the reachability of T. This is demonstrated by the example MDP in Figure 5.4. In this diagram, the circles represent possible states of the MDP. T contains only the left-most state. There is only one action available in each state outside of T, indicated by the branching arrows exiting the corresponding circles. With some probability, the action leaves the state of the environment unchanged. Otherwise the environment moves to the state immediately to the left. In this MDP, the environment eventually reaches T with probability one from any initial state $s_0 \notin T$. However, there is no CLF for this MDP. As indicated in the diagram, the probability that an action leaves the state of the environment unchanged grows for states farther to the right. Because this probability grows arbitrarily close to one, there can be no fixed probability $p > 0$ of descent for every state outside of T.

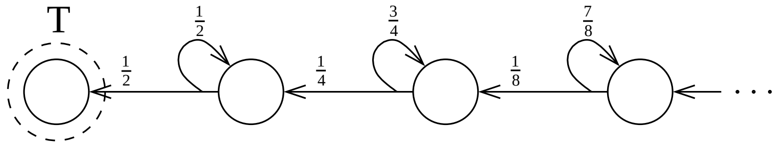

Figure 5.4. An MDP demonstrating that CLFs, as we have defined them, need not exist for stochastic systems even if T is reachable.

Actually determining a Lyapunov function for a given problem can be a difficult task. In practice, Lyapunov functions are usually identified through some combination of skill, intuition, and expertise. However, one does not always have to develop a Lyapunov function from scratch. For many general classes of environments, standard methods for constructing Lyapunov functions are known. Examples include: environments whose dynamics are linear or are feedback linearizable [96]; environments whose dynamics have only upper-triangular non-linearities, only lower-triangular non-linearities, or certain combinations of the two [45, 78]; and many more. Lyapunov functions have been developed for many important problem domains, including: robot manipulator control; robot navigation; satellite, ship, and airplane steering; network queue control; decentralized/multi-agent coordination; and chemical process control, to name a few [81, 48, 96, 57]. So, although finding Lyapunov functions is difficult in general, there are a number of options available to the practitioner who seeks one, and there have been many successful applications.

For many problems, the designer of the control system also has some degree of freedom in designing the system to be controlled. For example, robots used in industrial manufacturing are not off-the-shelf, generic items. Typically they are specially designed for the task they are supposed to perform. This makes the problem of controlling the robot to achieve the desired task much simpler. Similarly, the approach we advocate is to design the MDP that the agent has to solve using relevant domain knowledge, such that the control problem is tractable, and theoretical safety and performance guarantees can be established.

CHAPTER 6

EXAMPLE DOMAINS AND LYAPUNOV ANALYSES

In this chapter we introduce several problem domains that we use to demonstrate the main ideas of the thesis. We present Lyapunov analyses and design controllers that, while not optimal, provably enjoy non-trivial performance guarantees (e.g., guarantees on bringing the environment to a goal state on every trial). In subsequent chapters, we apply optimal control algorithms to these domains. We use the analyses from this chapter to ensure well-definedness of the control problems we formulate, to ensure the completeness of the optimal control algorithms we apply, and to establish performance guarantees for algorithms that do not offer such guarantees otherwise.

6.1 Deterministic Pendulum Swing-Up and Balance

Pendulum control problems have been a mainstay of control research for many years, in part because pendulum dynamics are simply stated yet highly nonlinear. Many researchers have discussed the role of mechanical energy in controlling pendulum-type systems [84, 14, 15, 22, 68, 67]. The standard tasks are either to swing the pendulum's end point above some height (swing-up) or to swing the pendulum up and balance it in a vertical position (swing-up and balance). In either task, the goal states have greater total mechanical energy than the initial state, which is typically the hanging-down, rest position. Thus, any controller that solves one of these tasks must somehow impart a certain amount of energy to the system.

Recall that the state of the pendulum, depicted in Figure 6.1, is given by an angular position, $\theta \in [-\pi, \pi]$, and an angular velocity, $\dot{\theta} \in \mathbb{R}$. For the first control

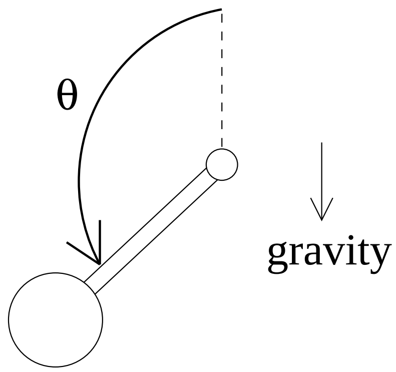

Figure 6.1. The single-link pendulum.

task we study, we assume a deterministic, frictionless pendulum controlled by torque applied to the fulcrum of the pendulum. This results in the acceleration equation:

$$\ddot{\theta} = \sin\theta + u \ ,$$

where the sine term is due to gravity and u is the control torque. We assume u is bounded in magnitude by some $u_{max} > 0$ that is relatively small. When $u_{max} < 1$, the influence of gravity is larger than u_{max} for some states, making the pendulum *underpowered*. In particular, it is not possible to drive the state of the pendulum directly to any desired value, nor is it possible to hold the pendulum at any given position. The first task we study is minimum-time swing-up to any near-upright, near-stationary state. Specifically, we choose the goal set $G^1_{pend} = \{(\theta, \dot{\theta}) : \|(\theta, \dot{\theta})\| \leq 0.01\}$.

The mechanical energy of the pendulum is

$$\mathrm{ME}(\theta, \dot{\theta}) = 1 + \cos\theta + \frac{1}{2}\dot{\theta}^2 \ . \tag{6.1}$$

When the pendulum is upright and stationary, it has a mechanical energy of 2. For any other state with ME $= 2$, if u is taken to be zero, the natural dynamics of the pendulum will cause it to swing upright and asymptotically approach the state $(\theta, \dot{\theta}) = (0, 0)$. The swing-up and balance problem for the single-link, frictionless pendulum can thus be reduced to the problem of achieving any state with ME $= 2$ (assuming the pendulum starts with less energy than 2). In other words, we can define a surrogate goal set of $G^2_{pend} = \{(\theta, \dot{\theta}) \mid \mathrm{ME}(\theta, \dot{\theta}) = 2\}$, and upon reaching that set, dictate the choice $u = 0$ in order to allow the pendulum to swing to near-upright.

6.1.1 First Attempt to Design a Controller

Let us design a controller based on the idea of increasing the pendulum's energy until it hits the set G^2_{pend}. The time derivative of the pendulum's energy is

$$\dot{\mathrm{ME}} = -\sin(\theta)\dot{\theta} + \dot{\theta}\ddot{\theta} = \dot{\theta}u \ .$$

An obvious strategy is to pick u to be of the same sign as $\dot{\theta}$ and of maximum magnitude, u_{max}. This maximizes, at each instant, the rate of increase of ME. When $\dot{\theta} = 0$, a natural choice is to match the sign of θ so that the pendulum is accelerated in the same direction as gravity. This choice also ensures that the control signal, u, is piecewise continuous from the right in time. We call this strategy the "energy ascent" (EA) control law:

$$\mathrm{EA}(\theta, \dot{\theta}) = \begin{cases} +u_{max} & \text{if } \dot{\theta} > 0 \text{ or } (\dot{\theta} = 0 \text{ and } \theta \geq 0) \\ -u_{max} & \text{if } \dot{\theta} < 0 \text{ or } (\dot{\theta} = 0 \text{ and } \theta < 0) \end{cases}$$

Is this a good control law? Under some circumstances it brings the state of the pendulum to G^2_{pend}, and then G^1_{pend}, quickly. Under other circumstances it works poorly or not at all. Figure 6.2 plots the time it takes EA to bring the state of the pendulum to G^1_{pend} and G^2_{pend} from initial state $(\pi, 0)$ for a range of u_{max}. Generally, as u_{max} increases, the time to reach the goal set decreases, as one would expect. However, the precise relationship is complex and non-monotonic. We argue below that for isolated values of u_{max}—roughly at the spikes in the time-to-goal curves—

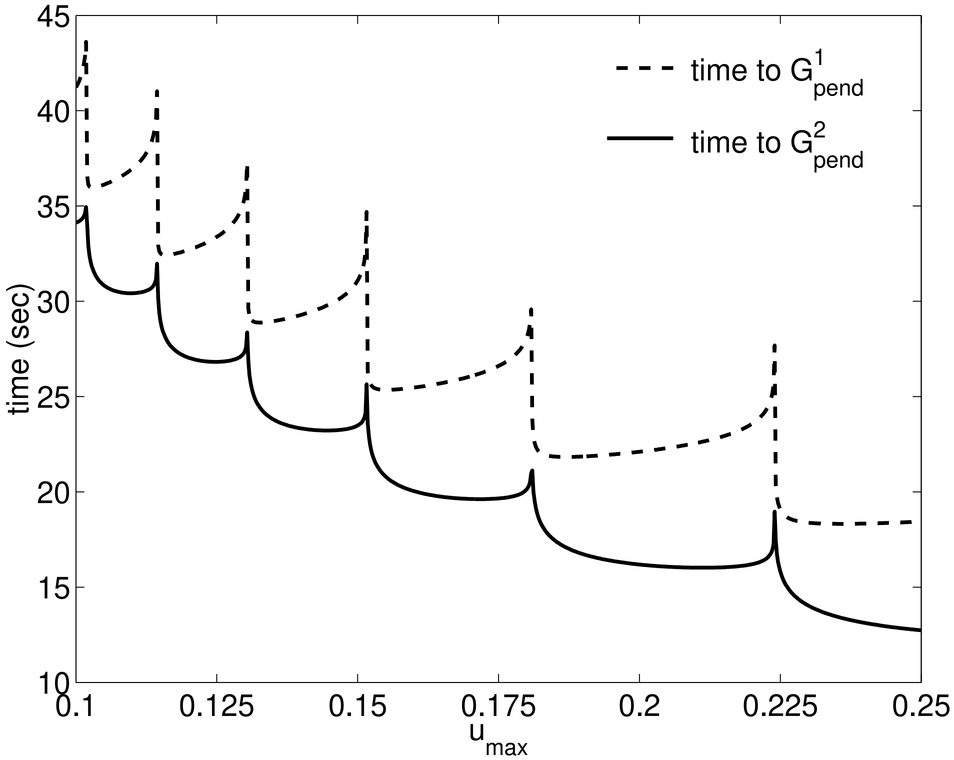

Figure 6.2. The time it takes EA to bring the state of the pendulum to the two goal sets.

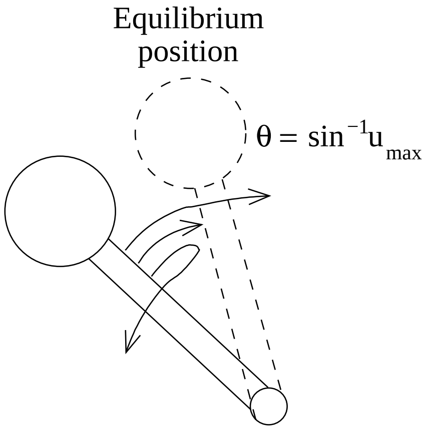

Figure 6.3. Pendulum controlled by EA approaching equilibrium with gravity.

the state of the pendulum does not reach the goal set at all. The time-to-goal goes to infinity at these points.

Why is this? When the pendulum is underpowered, any controller must swing the pendulum back and forth a number of times until it has enough mechanical energy to swing upright. The step-wise appearance of the curves reflects differences in the number of swings needed before enough energy is imparted. The spikes in the curves separate regions of u_{max} for which different numbers of swings are needed to get up, and this is where the EA controller runs into trouble.

Note that when the pendulum is underpowered ($u_{max} < 1$), there are states in which applying $\pm u_{max}$ torque exactly cancels the acceleration due to gravity. In particular, let us focus on the angular position $\theta_{u_{max}} = \sin^{-1}(u_{max})$. For the pendulum to swing upright, its position must pass either $\theta_{u_{max}}$ or $-\theta_{u_{max}}$. There are three qualitatively different behaviors that may occur, as depicted in Figure 6.3. For some choices of u_{max}, the trajectory generated by EA passes through a state with position $\theta_{u_{max}}$ and on to the upright position. For other choices of u_{max}, the pendulum may swing towards position $\theta_{u_{max}}$ but not reach it, instead swinging back the other way and passing $-\theta_{u_{max}}$ on the way to upright. For isolated values of u_{max}, the pendulum approaches $\theta_{u_{max}}$ asymptotically, neither passing by, nor swinging back the other way.

This is what happens at the spikes in the curves in Figure 6.2. EA causes the pendulum to approach one of $\pm\theta_{u_{max}}$, the equilibrium points where $\pm u_{max}$ torque exactly balances the acceleration due to gravity. The pendulum's velocity is nonzero as it approaches this equilibrium point, and so the mechanical energy of the pendulum keeps increasing. However, the rate of increase of the energy goes to zero and the pendulum never reaches mechanical energy ME = 2. That is, the state of the pendulum never reaches G^2_{pend}, and thus never reaches G^1_{pend}.

6.1.2 Second Controller and Lyapunov Analysis

The problem with the EA controller is that it can approach equilibrium points where $\pm u_{max}$ torque exactly balances the torque due to gravity. This problem can be avoided if we design a controller that follows the same basic pattern, but that does something different when the pendulum is near one of these equilibrium points. Our previous analysis showed that we should choose u to be of the same sign as $\dot{\theta}$ in order to ensure $\dot{\text{ME}} \geq 0$. To avoid the equilibrium problem, we propose a modification of the EA controller that switches to a torque of lower magnitude, but of the same sign, when the pendulum is near equilibrium. For reasons that become clear later, the modified energy ascent (MEA) control law depends on three parameters: μ, the magnitude of the torque applied by the motor "most of the time", and ϵ_θ and $\epsilon_{\dot{\theta}}$, which determine the "near to equilibrium" region of the state space. Four ranges of near-equilibrium positions are defined by μ and ϵ_θ, as depicted in Figure 6.4. Letting $\theta_\mu = \sin^{-1}(\mu)$, define:

$$\begin{aligned}
\theta_0 &= -\pi + \theta_\mu \ , & E_0 &= [\theta_0 - \epsilon_\theta, \theta_0 + \epsilon_\theta) \ , \\
\theta_1 &= -\theta_\mu \ , & E_1 &= [\theta_1 - \epsilon_\theta, \theta_1 + \epsilon_\theta) \ , \\
\theta_2 &= \theta_\mu \ , & E_2 &= (\theta_2 - \epsilon_\theta, \theta_2 + \epsilon_\theta] \ , \\
\theta_3 &= \pi - \theta_\mu \ , & E_3 &= (\theta_3 - \epsilon_\theta, \theta_3 + \epsilon_\theta] \ .
\end{aligned}$$

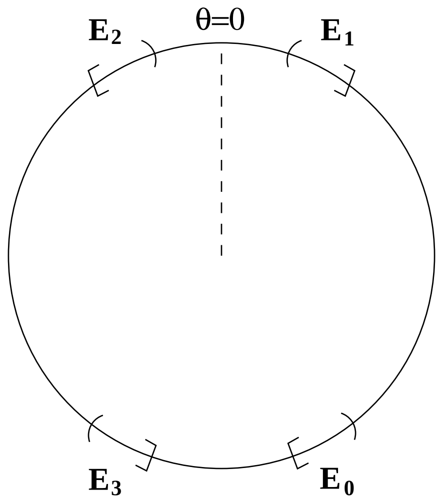

Figure 6.4. Near-equilibrium ranges for the MEA controller.

The MEA control law is:

$$\mathrm{MEA}_{\mu,\epsilon_\theta,\epsilon_{\dot\theta}}(\theta,\dot\theta) = \begin{cases} +\mu & \text{if } (\dot\theta = 0 \text{ and } \theta \geq 0) \text{ or } (\dot\theta > 0 \text{ and } (\dot\theta > \epsilon_{\dot\theta} \text{ or } \theta \notin E_0 \cup E_1)) \\ +\frac{1}{2}\mu & \text{if } \dot\theta \in (0, \epsilon_{\dot\theta}] \text{ and } \theta \in E_0 \cup E_1 \\ -\frac{1}{2}\mu & \text{if } \dot\theta \in [-\epsilon_{\dot\theta}, 0) \text{ and } \theta \in E_2 \cup E_3 \\ -\mu & \text{if } (\dot\theta = 0 \text{ and } \theta < 0) \text{ or } (\dot\theta < 0 \text{ and } (\dot\theta < \epsilon_{\dot\theta} \text{ or } \theta \notin E_2 \cup E_3)) \end{cases}$$

Theorem 6.1 *For any $\Delta > 0$, any $\mu > 0$, and appropriate ϵ_θ and $\epsilon_{\dot\theta}$, there exists $\delta > 0$, such that for any initial conditions $(\theta_0, \dot\theta_0)$, if the pendulum is controlled according to $MEA_{\mu,\epsilon_\theta,\epsilon_{\dot\theta}}$ for Δ time, then the mechanical energy of the pendulum is greater by at least δ at the end of that time.*

A proof of this theorem can be found in Appendix A, along with the definition of appropriate ϵ_θ and $\epsilon_{\dot\theta}$.

Suppose the pendulum is controlled by MEA, and consider the state of the pendulum just at the discrete points in time: $t = 0, \Delta, 2\Delta, \ldots$ Viewed this way, the system is a deterministic Markov process. Suppose we define $L_{pend}(\theta,\dot\theta) = 2 - \mathrm{ME}(\theta,\dot\theta)$ and $T = \{(\theta,\dot\theta) : \mathrm{ME}(\theta,\dot\theta) \geq 2\}$. By the previous theorem, the state of the pendulum

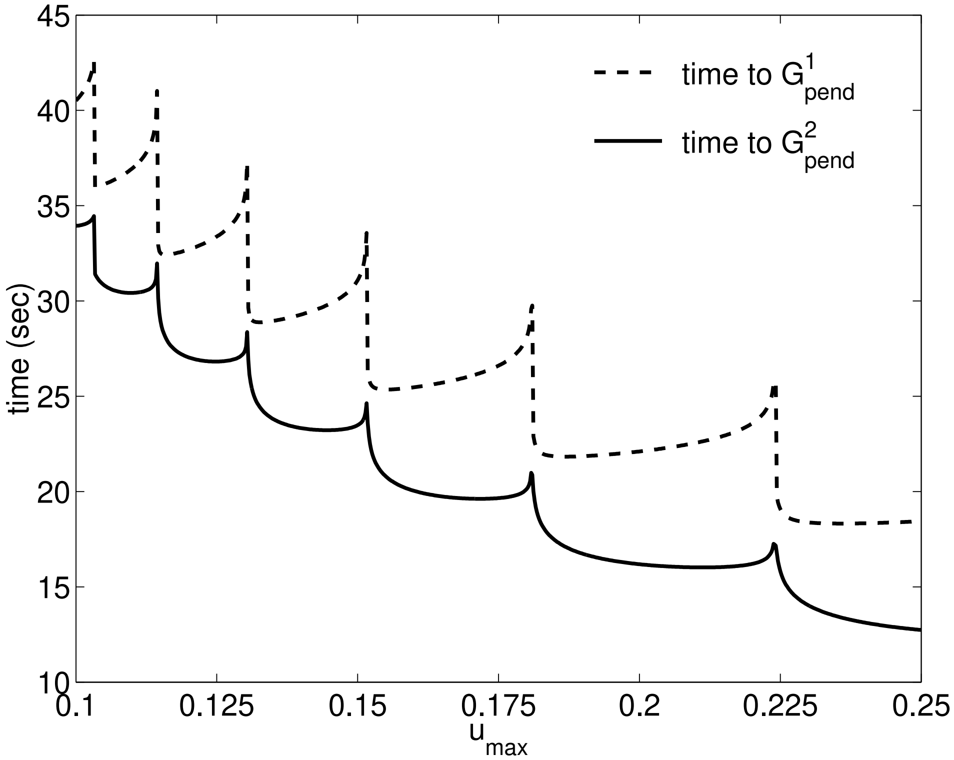

Figure 6.5. The time it takes MEA to bring the state of the penduulum to the two goal sets.

descends on L_{pend} outside of T. In other words, L_{pend} is a Lyapunov function. For any $(\theta_0, \dot{\theta}_0)$ such that $\mathrm{ME}(\theta, \dot{\theta}) < 2$, Theorem 5.3 (the guaranteed-descent theorem for Markov processes) ensures that the state of the pendulum reaches T within time $\lceil (2 - \mathrm{ME}(\theta_0, \dot{\theta}_0))\Delta/\delta \rceil$. Thus, the MEA controller provably brings the state of the pendulum to G^2_{pend}. From there, applying zero torque allows the state of the pendulum to reach G^1_{pend}, so the overall strategy provably succeeds in swinging the pendulum upright.

Figure 6.5 displays the time it takes for the state of the pendulum to reach G^1_{pend} and G^2_{pend} under the MEA control law. The curves are subtly different from those produced by the EA control law. In Figure 6.6 we plot the time to G^1_{pend} for EA and MEA for a small range of u_{max} surrounding one of the spikes in the curves. The solid curve shows how the performance of EA degenerates around a specific value of

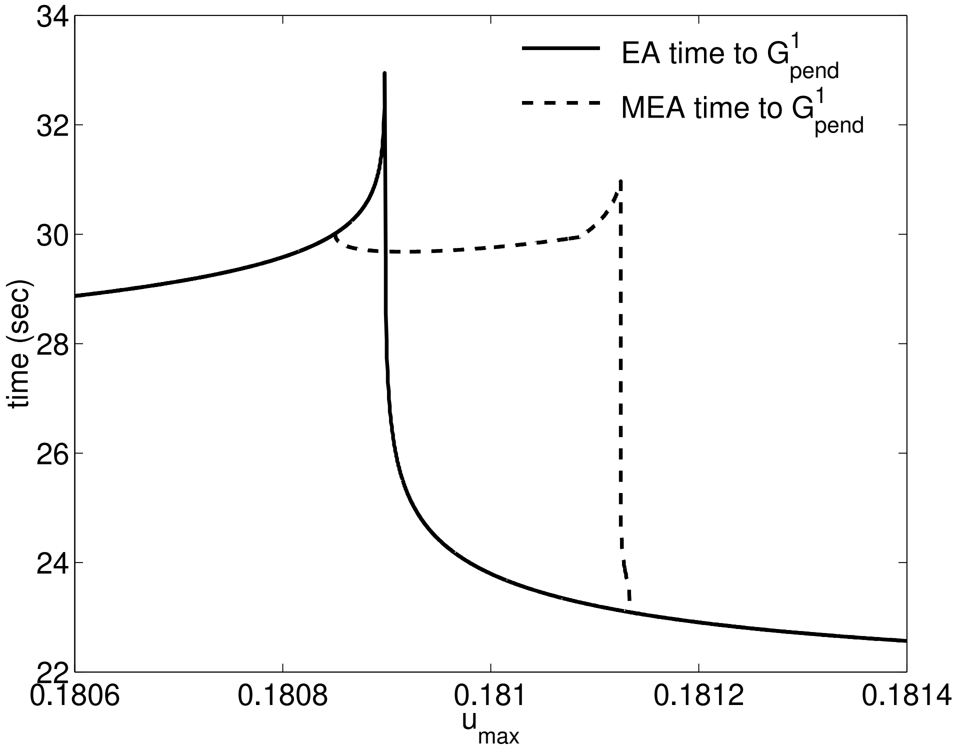

Figure 6.6. Close up comparison of EA and MEA.

u_{max}, resulting in a trajectory that does not reach G^1_{pend}. MEA avoids this problem, as is seen clearly from the dashed curve. The spike in time-to-goal is replaced by a shallow trough. (MEA appears to have a spike of its own in a different location, but this spike does not go to infinity, as we prove in Appendix A.) However, note that neither control law is better than the other for all values of u_{max}.

Note also that if one increases u_{max}, then one can apply a strictly greater set of control laws to the problem. So, the minimum achievable time-to-goal is a non-increasing function of u_{max}. The non-monotonicity of the time-to-goal curves indicates that for many values of u_{max}, neither EA nor MEA is optimal.

In summary, we can use MEA to increase the pendulum's mechanical energy to 2, and then let it free-swing upright. MEA provably achieves the basic task of swinging the pendulum up and balancing it. However, MEA is not optimal. We could continue trying to design a good control law by hand. Instead, we explore the possibility of using state-space search and reinforcement learning to determine better control strategies. In some cases, we incorporate our knowledge of L_{pend} and MEA into the MDP we formulate, ensuring that the problem can be solved and establishing performance guarantees for the solution algorithms.

The analysis and design presented above demonstrate several features typical of Lyapunov-based control design. With some effort, one can design a control strategy that provably achieves the basic goal of controlling the environment (e.g., swinging the pendulum upright). However, that solution is typically not optimal, and the relationships between various parameters (e.g., u_{max}) and performance can be complex and unpredictable.

6.2 Stochastic Pendulum Swing-Up and Balance

Next we consider a swing-up and balance task in which the pendulum's dynamics are stochastic. We assume the pendulum's angular position is continuously disturbed by Gaussian noise ("white noise") with zero mean and 0.1 standard deviation. These dynamics are modeled by the control stochastic differential equations:

$$\begin{aligned} d\theta &= \dot{\theta} dt + 0.1 dW \ , \\ d\dot{\theta} &= \sin(\theta) dt + u dt \ , \end{aligned}$$

where W denotes the standard Wiener process.[1] We also assume that the pendulum's velocity is bounded in magnitude by $\dot{\theta}_{max} = 4$. If the dynamics push velocity outside this range, it is simply held at $\pm\dot{\theta}_{max}$.

[1]Intuitively, a Wiener process is a continuous-time Gaussian-distributed random walk on the real line. More precisely, the change in the position of the "walker" during any period time Δt has a Gaussian distribution with mean zero and variance Δt. See Higham [31] for an introduction to Wiener processes.

The task we consider is to keep the state of the pendulum in a set of near-upright, low-velocity states, $T_{up} = \{(\theta, \dot{\theta}) : |\theta| < 0.5,\ |\dot{\theta}| < 0.3\}$, as much of the time as possible. The condition $|\theta| < 0.5$ corresponds to pendulum positions within approximately 30 degrees of upright. Because of the stochastic dynamics, no controller can keep the pendulum upright indefinitely. Indeed, because the Gaussian noise has infinite tails, for any $(\theta, \dot{\theta}) \in T_{up}$ there is some probability that the process "jumps" out of T_{up} by the next time step.

Regardless of the stochastic dynamics, the mechanical energy of the pendulum when it is upright and has zero velocity is precisely 2. Because the position disturbance has zero mean, if the pendulum starts with energy 2, it will tend to maintain that energy and swing upright. Thus, a sensible control strategy is to increase the pendulum's energy whenever it is below 2, and decrease it whenever it is above 2. If the energy is precisely 2, no control torque need be applied, and the pendulum is allowed to swing naturally, so that its state reaches T_{up} (if it is not there already).

To increase energy, we can rely on the MEA controller developed above. To decrease energy, recall that under the deterministic dynamics, $\dot{\mathrm{ME}} = u\dot{\theta}$. Choosing $u = -\mathrm{sgn}(\dot{\theta})u_{max}$ brakes the pendulum, decreasing ME as quickly as possible. Thus, we define the MEto2 control law:

$$\mathrm{MEto2}(\theta, \dot{\theta}) = \begin{cases} \mathrm{MEA}_{u_{max}, \epsilon_\theta, \epsilon_{\dot{\theta}}}(\theta, \dot{\theta}) & \text{if } \mathrm{ME}(\theta, \dot{\theta}) < 2 - \epsilon_{ME} \\ 0 & \text{if } 2 - \epsilon_{ME} \leq \mathrm{ME}(\theta, \dot{\theta}) \leq 2 + \epsilon_{ME} \\ -\mathrm{sgn}(\dot{\theta})u_{max} & \text{if } \mathrm{ME}(\theta, \dot{\theta}) > 2 + \epsilon_{ME}\ . \end{cases}$$

The control law is parameterized by ϵ_θ and $\epsilon_{\dot{\theta}}$, for the MEA case, and ϵ_{ME}, which allows us to specify a range of energies that are close enough to 2 so that no torque need be applied. Letting $L^2_{pend}(\theta, \dot{\theta}) = |\mathrm{ME}(\theta, \dot{\theta}) - 2|$ and $T_{\epsilon_{ME}} = \{(\theta, \dot{\theta}) : |\mathrm{ME}(\theta, \dot{\theta}) - 2| \leq \epsilon_{ME}\}$, we have:

Theorem 6.2 *For any $\Delta > 0$ there exists $\delta > 0$ and $p > 0$ such that for any initial state $(\theta_0, \dot{\theta}_0) \notin T_{\epsilon_{ME}}$, if MEto2 is used to control the stochastic pendulum for duration Δ, then with probability at least p, the state of the pendulum enters $T_{\epsilon_{ME}}$ or ends up at least δ lower on L^2_{pend}.*

Thus, if the pendulum is controlled according to MEto2, the Markov process that results from viewing the state of the pendulum at times $t = 0, \Delta, 2\Delta, 3\Delta, \ldots$ descends probabilistically on the Lyapunov function L^2_{pend} outside of $T_{\epsilon_{ME}}$. Since $\theta \in [-\pi, \pi]$ and since we assumed $\dot{\theta} \in [-4, 4]$, L^2_{pend} is bounded above outside of $T_{\epsilon_{ME}}$. Theorem 5.4 guarantees that MEto2 brings the pendulum to the desired energy with probability one. Afterwards, there is some probability that the pendulum swings up and its state enters T_{up}. If the state of the pendulum leaves $T_{\epsilon_{ME}}$ and fails to enter T_{up}, then Theorem 5.4 ensures that MEto2 brings the state back to $T_{\epsilon_{ME}}$, giving it another chance to enter T_{up}. Thus, MEto2 is guaranteed to bring the state of the pendulum to T_{up} eventually from any initial state.

Proof sketch (of Theorem 6.2): Under the deterministic dynamics, δ descent on L^2_{pend} is guaranteed. There is some probability of a δ descent under the stochastic dynamics because, roughly speaking, there is some probability that the stochastic dynamics result in a next state "near" to the one that would result under the deterministic dynamics. More formally, for any ϵ_1, there is some probability $p = p(\epsilon_1)$ that the random disturbance due to the Gaussian noise would stay within ϵ_1 over the course of Δ time. That means there is some probability that the state of the stochastic pendulum after Δ time is near (say within ϵ_2) the state that would result under the deterministic dynamics (which is at least δ_1 lower on L^2_{pend}). Since L^2_{pend} is Lipschitz continuous[2], for sufficiently small ϵ_1 this guarantees some probability of having a new state at least, say $\delta = \delta_1/2$ lower on L^2_{pend}. $\square$

---

[2]The fact that L^2_{pend} is Lipschitz continuous means that there exists some constant $c \in \mathbb{R}$ such that for any $(\theta_1, \dot{\theta}_1)$ and $(\theta_2, \dot{\theta}_2)$, $|L^2_{pend}(\theta_1, \dot{\theta}_1) - L^2_{pend}(\theta_2, \dot{\theta}_2)| \leq c\|(\theta_1, \dot{\theta}_1) - (\theta_2, \dot{\theta}_2)\|$.

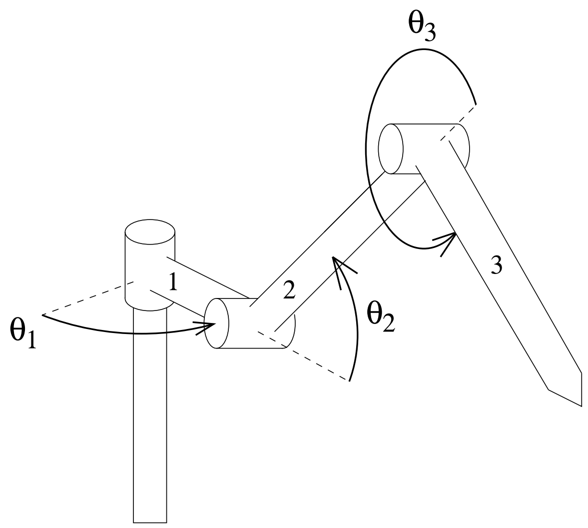

Figure 6.7. Three-link robot arm.

Suppose we let $\epsilon_\theta = \frac{1}{2}(\sin^{-1}(u_{max}) - \sin^{-1}(\frac{1}{2}u_{max}))$, $\epsilon_{\dot\theta} = 0.1$, and $\epsilon_{ME} = 0.01$. Simulating the stochastic dynamics under MEto2 reveals that the state of the pendulum spends approximately 49.2% of the time in T_{up}. In Chapter 8 we show that this is far from optimal. An agent that uses the MEto2 controller outside of T_{up}, but does a better job of keeping the state in T_{up} once it is there, can keep the pendulum upright roughly 78% of the time.

6.3 Robot Arm Control

Robot arm, or manipulator, control has numerous applications in manufacturing and aerospace settings. For example, assembly, loading and unloading, and painting are common tasks for robot arms. For many such tasks, good control designs have been developed based on qualitative, analytical methods, but cost-optimizing approaches are rare. Near-optimal control of even a three-joint arm is challenging even for state-of-the-art dynamic-programming and optimization approaches [47, 55, 98].

Figure 6.7 depicts the particular three-joint arm design on which we focus. In general, a robot arm comprises a series of links connected by n actuated joints. The state of an arm is described by a vector of joint angular positions $\Theta \in \mathbb{R}^n$ together with a vector of joint angular velocities $\dot{\Theta} \in \mathbb{R}^n$. We first consider a standard deterministic, frictionless model of the arm dynamics [19]:

$$\ddot{\Theta} = H^{-1}(\Theta, \dot{\Theta})(\tau - V(\Theta, \dot{\Theta}) - \mathcal{G}(\Theta)) ,$$

where H is the $n \times n$ inertia matrix, τ is the vector of actuator torques applied at the joints, V models Coriolis and other velocity-dependent forces, and $\mathcal{G}$ models gravitational forces. A friction term could easily be included with minimal change to the approach taken below, but we omit it for simplicity. We assume that the joint positions are limited to the range $[-\pi, \pi]$. If a joint position would cross one of the endpoints of this range, it is held there and the joint velocity is set to zero. Links are of length one and have mass one. Each link's mass is modeled as a point mass at the end of the link.

Note that the single-link pendulum described above can be viewed as a special case of a robot arm. However, the key assumption that makes controlling the pendulum interesting is that the control torque is bounded. In our arm control tasks, we assume that there are no torque limits or that the limits are sufficiently generous that the controllers we design do not run up against them. This assumption is important for the particular Lyapunov approach we take, although other Lyapunov approaches may be possible if actuator torques are limited to smaller ranges than the ones we use. Since it is fairly common in industrial practice, for example, for actuators to be sufficiently powerful to implement our control designs, we consider this assumption reasonable.

The task we consider is minimum-cost control to a region of states near the origin, $G_{arm} = \{(\Theta, \dot{\Theta}) : \|(\Theta, \dot{\Theta})\| \leq 0.01\}$. For the three-link arm depicted in Figure 6.7, this corresponds to a set of low velocity states near a fully extended, horizontal configuration. Let $(\Theta(t), \dot{\Theta}(t))$ be the state of the arm at time t and let $\tau(t))$ be the torque at time t. Suppose $(\Theta(t_G), \dot{\Theta}(t_G)) \in G_{arm}$ at some time t_G. The cost associated with the sequence of states and torques until time t_G is $\int_{t=0}^{t_G} \|\Theta(t)\|^2 + \|\tau(t) - \tau_0\|^2 dt$, where τ_0 is the amount of torque needed to hold the arm at the origin. Intuitively, the first term penalizes the extent to which the state of the arm is far from the origin, and thus far from G_{arm}. The second term penalizes the control torque used in getting the state of the arm to G_{arm}, except the torque τ_0, which is unavoidable.

6.3.1 A Control Design and Lyapunov Analysis

A standard approach to arm control combines feedback linearization with some form of linear system control [19]. In a simple approach to feedback linearization, we assume that the functions H, V, and $\mathcal{G}$ are known and computable. The control torque choice, τ, is reparameterized in terms of a vector u, where $\tau = H(\Theta, \dot{\Theta})u + V(\Theta, \dot{\Theta}) + \mathcal{G}(\Theta)$. This transformation is lossless, since H is always of full rank, and has the advantage of tremendously simplifying the equation for the dynamics of the arm, which becomes:

$$\ddot{\Theta} = u \ .$$

Once the dynamics are expressed in linear form, linear control design methods apply. We use a linear-quadratic regulation (LQR) approach. This results in a controller that is able cause the state of the arm to enter G_{arm}, and provides a control Lyapunov function. We briefly describe the general LQR approach.

The "linear" part of LQR comes from the assumption that the system dynamics are linear. In general, one assumes the form: $\dot{x} = Ax + Bu$, where x is the state vector of the system, u is a control vector, and A and B are constant matrices of appropriate dimension. There are no bounds on x or u.

Suppose we want to move this system to a target state x_T, from any initial state x_0, and let us assume that a control input of u_T holds the system at x_T. That is, $Ax_T + Bu_T = 0$. The "quadratic" part of LQR says that a good control law should produce sequences of states $x(t)$ and controls $u(t)$ that minimize the quadratic cost functional: $\int_{t=0}^{\infty}(x(t)-x_T)'Q(x(t)-x_T)+(u(t)-u_T)'R(u(t)-u_T)dt$, where Q and R are positive definite, symmetric matrices, which we can choose. The apostrophe ($'$) denotes taking the transpose. By varying our choices of Q and R, we can express different tradeoffs between the value of approaching x_T rapidly versus using little control effort. Note the similarity to the cost function defined above for the three-link arm control problem. An important difference is that the LQR design penalizes u, whereas the cost function we want to minimize penalizes $\tau - \tau_0$.

If we restrict ourselves to control laws of the form $u = -K(x - x_T) + u_T$, where K is a constant "gain" matrix, then the optimal gain matrix is $K = R^{-1}B'P$, where P solves the matrix algebraic Ricatti equation [96]:

$$PA + A'P - PBR^{-1}B'P = -Q \ .$$

Typically, the Ricatti equation is solved numerically, although in special cases an analytical solution is feasible. P is also important because it provides a Lyapunov function that can be used to prove that the gain matrix K causes the system to approach x_T asymptotically. It turns out that P is always a symmetric, positive definite matrix, so $L(x) = (x - x_T)'P(x - x_T)$ is positive for $x \neq x_T$. Further $\dot{L}(x) = -(x - x_T)'(Q + K'RK)(x - x_T)$, which is negative definite about x_T. We can easily conclude the following:

Theorem 6.3 *Let T_ϵ be an ϵ-ball of states centered on x_T. For any $\Delta > 0$, there exists $\delta > 0$ such that if the system $\dot{x} = Ax + Bu$ is controlled according to $u = -K(x - x_T) + u_T$ for Δ time, from any initial conditions, then the next state of the system either enters T_ϵ or is at least δ lower on L.*

6.3.2 Specialization to the Deterministic Three-Link Arm

We now apply these ideas to designing a control law for the three-link arm depicted in Figure 6.7. If we assume Q and R are diagonal, then P can be worked out by hand from the Ricatti equation. Letting $Q = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix}$, then $P = \begin{bmatrix} P_1 P_2 \\ P_2 P_3 \end{bmatrix}$ can be computed as:

$$
\begin{aligned}
P_2 &= \sqrt{RQ_1}\ , \\
P_3 &= \sqrt{R(Q_2 + 2P_2)}\ , \\
P_1 &= RP_2P_3\ ,
\end{aligned}
$$

For example, suppose we let $Q_1 = \text{diag}(1,1,1)$, $Q_2 = \text{diag}(.01,.01,.01)$ and $R = \text{diag}(1,1,1)$. This puts equal weight on position error and control effort, and little weight on velocity; we have no objection to letting the arm move to the target point quickly. Then we find $P_1 = P_3 = \text{diag}(\sqrt{2.01}, \sqrt{2.01}, \sqrt{2.01})$ and $P_2 = \text{diag}(1,1,1)$. The optimal gain matrix is $K = [K_1 K_2]$ where $K_1 = \text{diag}(1,1,1)$ and $K_2 = \text{diag}(\sqrt{2.01}, \sqrt{2.01}, \sqrt{2.01})$. Using this gain matrix, we define:

$$
\text{FL}_1(\Theta, \dot{\Theta}) = -H(\Theta, \dot{\Theta})K \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} + V(\Theta, \dot{\Theta}) + \mathcal{G}(\Theta)\ .
$$

This is a control law based on the feedback linearization transformation and LQR design, hence the designation, "FL." The "1" in the subscript anticipates other control laws based on feedback linearization and LQR design, that we introduce in Chapter 7. By Theorem 6.3, FL_1 causes the state of the arm to descend on the function $L_{arm} = [\Theta' \ \dot{\Theta}']P \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix}$ outside of G_{arm}. (Note that G_{arm} is an ϵ-ball around the origin with $\epsilon = 0.01$.) In other words, L_{arm} is a Lyapunov function for the Markov process that results when the arm is controlled by FL_1 and the state is viewed at

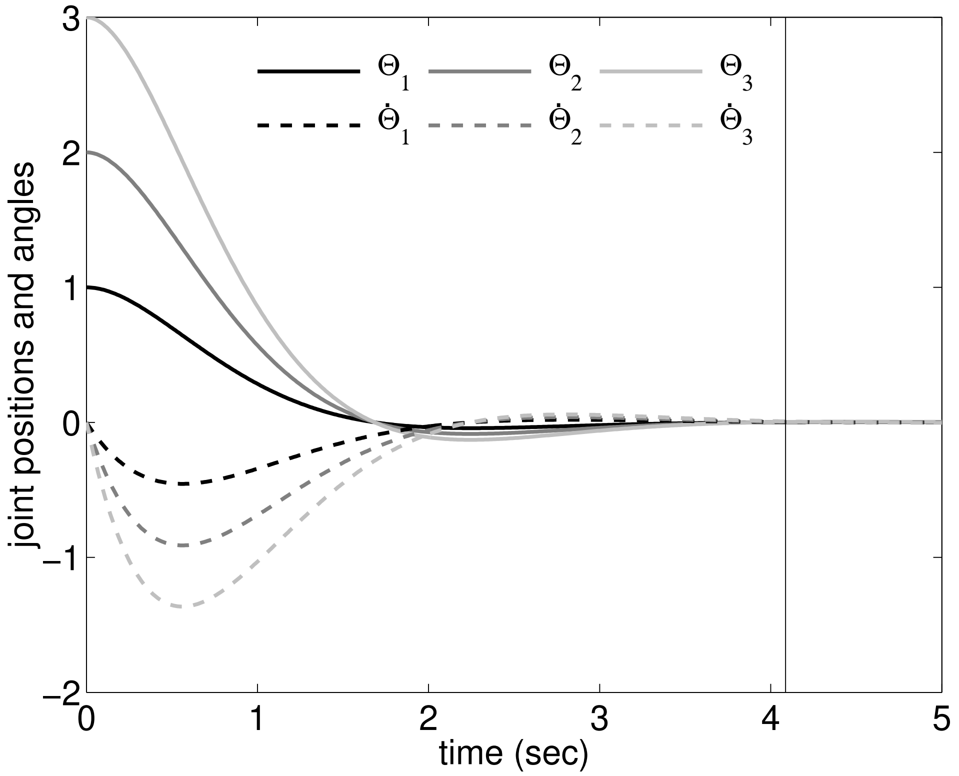

Figure 6.8. Arm state trajectory under FL_1.

times $t = 0, \Delta, 2\Delta, \ldots$ for any $\Delta > 0$. By Theorem 5.3, we know that the state of the arm is brought to G_{arm} from any initial state outside of G_{arm}. If FL_1 is applied to the arm indefinitely, without stopping when the state of the arm reaches G_{arm}, then the state approaches the origin asymptotically.

Figure 6.8 plots the state trajectory that results from initial configuration $\Theta'_0 = [1, 2, 3]$ and zero velocity. The vertical line shortly after the 4 second mark indicates the time at which the state of the arm enters G_{arm}. The controller is satisfactory, but it is not optimal for at least two reasons. First, the cost function minimized by the LQR design is different from the cost function of the control problem. Second, LQR design produces only linear feedback controllers. Nonlinear control laws allow better performance. In Chapters 7 and 8, we demonstrate that state-space search and reinforcement learning agents are able to find lower cost solutions by switching among this and other control laws. One weakness of the FL_1 controller is that it generates large control signals when the system is far from the target configuration. The size of this control effort is accentuated by the square in the cost function. In contrast, a more moderate, sustained acceleration incurs a lower control-effort cost and can still bring the state of the arm to G_{arm} relatively quickly. There are other issues as well, and these are discussed when we present alternative controls laws and compute switching policies in Chapters 7 and 8.

6.3.3 Stochastic Arm Control Problem

We also experiment with a stochastic version of the arm control problem, using dynamics:

$$d \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} = \begin{bmatrix} \dot{\Theta}(dt + 0.2dW) \\ H^{-1}(\Theta, \dot{\Theta})(\tau - V(\Theta, \dot{\Theta}) - \mathcal{G}(\Theta))dt \end{bmatrix},$$

where W is now to be interpreted as a vector of three independent Wiener processes. This models mean one, standard deviation 0.2, white noise disturbances to the joint positions that are multiplicative in the joint velocities. Under these dynamics, the FL_1 controller causes the state of the arm to descend probabilistically on L_{arm}, and has probability 1 of eventually bringing the state of the arm to G_{arm} from any initial state.

Theorem 6.4 *For any $\Delta > 0$ there exists $\delta > 0$ and $p > 0$ such that for any initial state $(\Theta_0, \dot{\Theta}_0) \in [-\pi, \pi]^3 \times [-\pi/\sqrt{2.01}, \pi/\sqrt{2.01}]^3$, if the arm is controlled by FL_1 for Δ time, then with probability at least p, the state of the arm enters G_{arm} or descends on L_{arm} by at least δ. If the arm is controlled by FL_1 from $(\Theta_0, \dot{\Theta}_0)$ indefinitely, then the state of the arm state stays bounded and with probability 1 eventually enters G_{arm}.*

Proof (sketch): The descent condition holds for the stochastic arm for the same reason it holds for the stochastic pendulum—the state of the arm descends

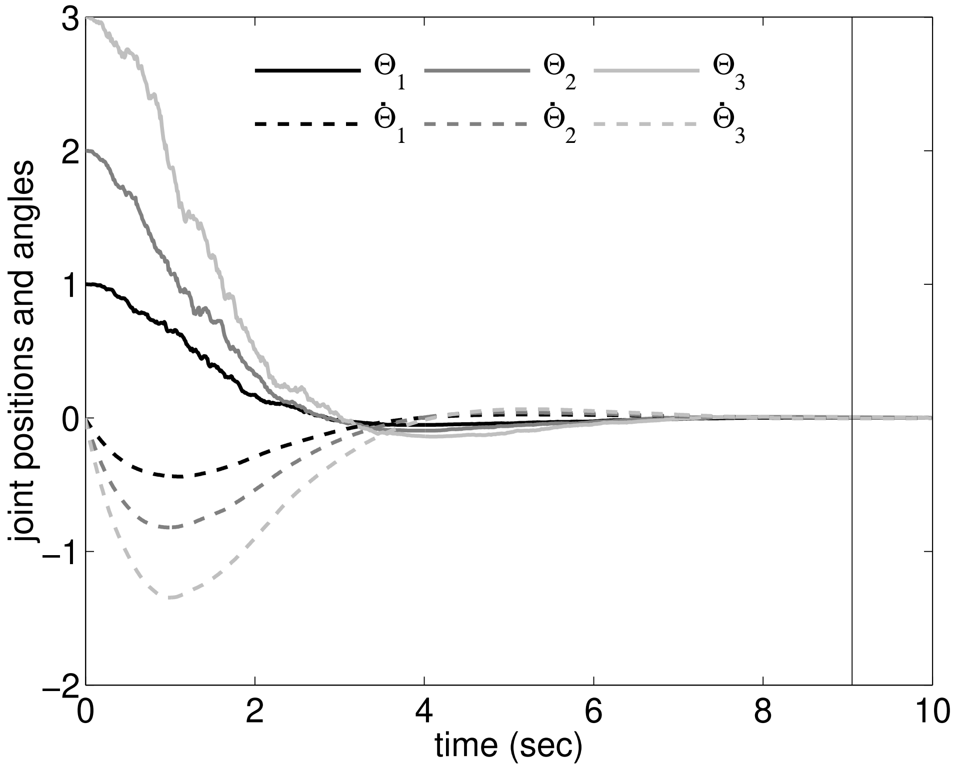

Figure 6.9. Arm state trajectory under FL_1 and stochastic dynamics.

under the deterministic dynamics, and the stochastic dynamics we have posited place some probability mass "near" the deterministic outcome. Why does the state stay bounded? The joint positions are, by definition, limited to the range $[-\pi, \pi]$. For joint i, the FL_1 control law makes the acceleration: $\ddot{\Theta}_i = -\Theta_i - \sqrt{2.01}\dot{\Theta}_i$. Since Θ_i is bounded, there is a bounded range of joint velocities under which the joint can experience acceleration. In particular, suppose $\ddot{\Theta}_i > 0$. Then $-\Theta_i - \sqrt{2.01}\dot{\Theta}_i > 0 \Rightarrow \dot{\Theta}_i < -\Theta_i/\sqrt{2.01} \leq \pi/\sqrt{2.01}$. So the joint's velocity can only increase if it is below $\pi/\sqrt{2.01}$. Similarly, velocity can only decrease above $-\pi/\sqrt{2.01}$, and thus velocity can never increase in magnitude beyond $\pi/\sqrt{2.01}$. Because the state stays bounded, L_{arm} can be bounded above outside of G_{arm}, so Theorem 5.4 implies that the state of the arm enters G_{arm} eventually with probability one. $\square$

Figure 6.9 displays one state trajectory from initial configuration $\Theta_0' = [1, 2, 3]$ and zero velocity. The position disturbance is evident in the curves, though they are qualitatively similar to the curves seen under the deterministic dynamics (Figure 6.8). The state of the arm takes more than twice as long to reach G_{arm} than in the deterministic problem. Indeed, under the stochastic dynamics there is no finite time bound by which we can be sure that the state of the arm enters G_{arm}. Understandably, it is difficult to bring the state of the arm to a small goal region when the dynamics are noisy.

We note that in the example above, the state of the arm is seen to approach the origin asymptotically. Theorem 6.4 implies only that the state of the arm reaches G_{arm}, not that it approaches the origin. Given the stochastic dynamics, it is not obvious that the state should even be able to approach the origin asymptotically, but it turns out that this is true [46]. We will not argue why this is the case. We simply note that since the Gaussian disturbances are multiplied by the arm's velocity, they approach zero if the arm's velocity approaches zero. Thus, as the state of the arm nears the origin, the noise disappears in such a way that an asymptotic approach is possible.

CHAPTER 7

LYAPUNOV FUNCTIONS FOR STATE-SPACE SEARCH

In this chapter, we show how Lyapunov domain knowledge can be used to address a number of the theoretical and practical difficulties that can arise in using state-space search methods, particularly for problems with infinite state sets. The two main questions we address are:

1. Under what conditions do solutions exist? That is, under what conditions is the goal set reachable from all non-goal states? In general, determining whether solutions exist is an undecidable problem.

2. Under what conditions are standard heuristic search algorithms complete? That is, under what conditions are they guaranteed to find a solution, if one exists?

We begin the chapter by establishing sufficient conditions, based on Lyapunov functions, for the existence of solutions and for the existence of optimal solutions. These conditions establish the well-definedness of the control problem. We then establish sufficient conditions for the completeness of the search procedures described in Section 4.1: best-first search, uniform-cost search, depth-first branch-and-bound, and limited look-ahead search. We demonstrate the basic theory on simulated pendulum and robot arm control problems. We show how Lyapunov domain knowledge can be used to design action formulations, goal sets, and heuristic functions for these problems, ensuring the existence of optimal solutions and ensuring that solutions are found by the search algorithms mentioned above.

7.1 Existence of Solutions and of Optimal Solutions

Recall that we are considering the problem of finding an action sequence solution to a deterministic, minimum cost-to-goal MDP. There is a start state $s_0 \in S$ and goal set $G \subset S$. We assume that in each non-goal state s, the set of allowed actions, $A(s)$, is finite and non-empty. Each $a \in A(s)$ results in an immediate cost $C(s, a)$ for the agent and causes the next state of the environment to be $F(s, a)$. An action sequence solution (henceforth, simply "solution") is a sequence of actions that brings the state of the environment from s_0 to some state in G. The total cost of a solution is the sum of the immediate action costs. An optimal solution has total cost less than or equal to that of any other solution.

In Chapter 5, we established the existence of solutions, although we did not use that terminology. Taking $T = G$, Theorem 5.9 implies that G is reachable from any initial state s_0 (i.e., a solution exists) if and only if there exists a CLF with target set G.

Theorem 7.1 *For a deterministic, minimum cost-to-goal MDP with goal set G, solutions exist from every initial state $s_0 \notin G$ iff there exists a CLF with target set G.*

Furthermore, knowing a CLF L makes it easy to construct a solution. At each state, one merely needs to choose any of the actions that cause the state of the environment to descend on L. One natural approach, called quickest descent, is to choose the action that decreases L the most. The solution produced in this way need not be optimal. Indeed, the existence of solutions does not imply the existence of optimal solutions, as discussed in Chapters 3 and 4. If there is an infinite number of solutions, it is possible that the infimal cost-to-goal is not attained by any solution, and that the problem is not well-defined. We propose two conditions under which the infimal cost-to-goal is guaranteed to be attained by some solution.

Theorem 7.2 *For a deterministic, minimum cost-to-goal MDP with goal set G, if L is a CLF with target set G and if all actions descend on L, then for any $s_0 \notin G$ there exists at least one optimal solution.*

Proof: The existence of L implies the existence of some solution. Since all actions descend on L (i.e., for all $s \notin G$ and $a \in A(s)$, $F(s,a) \in G$ or $L(s) - L(F(s,a)) \geq \delta$), all sequences of $\lceil L(s_0)/\delta \rceil$ actions bring the state of the environment to G on or before the $\lceil L(s_0)/\delta \rceil^{th}$ step. Thus, there is only a finite number of solutions. At least one of them must be optimal. $\square$

Having all actions descend on a Lyapunov function is a strong condition. If we know a CLF, we can ensure that all actions descend simply by discarding any actions that do not descend. However, discarding actions may result in a problem whose optimal solution is more expensive than that of the original problem. Whether discarding non-descending actions is a good idea depends on the particular problem and Lyapunov function.

If some actions do not descend, we can ensure the existence of optimal solutions by making another assumption that rules out the possibility of an infinite number of low cost solutions. We use the phrase "infinite action sequences have infinite cost" to mean that: for all $s_0 \notin G$, if $a_0, a_1, a_2, \ldots$ is an infinite sequence of actions that can be applied without causing the state of the environment to enter G, then $\sum_{i=0}^{\infty} c_i = \infty$, where c_i is the cost of action a_i.

Theorem 7.3 *For a deterministic, minimum cost-to-goal MDP with goal set G, if there exists a CLF with target set G and if infinite action sequences have infinite cost, then for any $s_0 \notin G$ there exists at least one optimal solution.*

Proof (by contradiction)**:** Consider any $s_0 \notin G$, and let L be a CLF with target set G. The existence of L implies the existence of some solution. Suppose this solution has cost C. Assume an optimal solution does not exist. Then there must be an infinite number of solutions with cost less than C. Since a finite number of actions are available from any state, any infinite set of solutions must contain arbitrarily long solutions. So, if there is an infinite number of solutions with cost less than C, there must be arbitrarily long solutions with cost less than C. That means there must be at least one infinite sequence of actions that does not bring the state of the environment to G, but which has a total cost less than or equal to C. This contradicts the assumption that infinite action sequences have infinite cost. $\square$

In summary, if a CLF exists, then a solution exists. If all actions descend on that CLF, or if infinite action sequences have infinite cost, then an optimal solution exists.

7.2 Completeness of Several State-Space Search Algorithms

Having provided conditions that establish the existence of solutions, the next question to address is how a solution can be found. A state-space search algorithm is *complete* if, given unbounded computational resources, the algorithm eventually finds and returns a solution for any problem that has one. As described in Section 4.1, many state-space search algorithms are not complete when S is infinite. In particular, this is the case for the three search algorithms we consider. Some search algorithms, including limited look-ahead search, are not complete even when S is finite. In this section, we consider what additional assumptions on the problem or on the search algorithms are sufficient to establish completeness. That is, we seek to identify limited classes of problems or ways of using the search algorithms such that finding solutions is guaranteed, if there are any to be found.

One important special class of problems are those MDPs for which all actions descend on a CLF, L, with target set G. From start state s_0, no action sequence longer than $\lceil L(s_0)/\delta \rceil$ is possible. So in this case, only a finite number of states are reachable from s_0. The fact that all actions descend on L also implies that no action sequence can cause a state to be visited more than once in a trajectory. Further, because $A(s)$ is non-empty for all $s \notin G$, any sequence of actions not bringing the state of the environment to G can always be extended to form a solution. Thus, we have:

Theorem 7.4 *For a deterministic, minimum cost-to-goal MDP with goal set G, if there exists a CLF, L, with target set G on which all actions descend, then for any $s_0 \notin G$ the search graph is finite and acyclic, and all terminal nodes (a.k.a. leaves) correspond to goal states.*

Under these conditions, depth-first search, breadth-first search, best-first search, uniform-cost search, greedy search, limited look-ahead search, and even random action selection produce solutions. Again, if we know a CLF for an MDP, we can ensure that all actions descend by forbidding any actions that do not. Although this may negatively impact the cost of optimal solutions, the fact that it allows one to use any search algorithm one wants is a strong incentive for making that restriction. Intermediate restrictions, such as allowing non-descending actions up to some depth k, but only allowing descending actions beyond depth k, may allow for lower-cost solutions than are possible under a strict descent constraint. Yet, finiteness of the search graph and completeness of the algorithms mentioned above is still guaranteed.

If L is a CLF for a problem but some actions do not descend, then with the exception of breadth-first search, none of the search algorithms mentioned above are guaranteed to find a solution. In the next two subsections, we describe additional assumptions that are sufficient to ensure that best-first search, uniform-cost search, depth-first branch and bound, and limited look-ahead search find solutions.

7.2.1 Best-First Search, Uniform-Cost Search and Depth-First Branch and Bound

If a CLF exists, then the additional assumption that infinite action sequences have infinite cost is sufficient to make best-first search complete for any choice of $\hat{h}$.

Theorem 7.5 *For a deterministic, minimum cost-to-goal MDP with goal set G, if there exists a CLF with target set G and if infinite action sequences have infinite cost, then for any $s_0 \notin G$, best-first search terminates and returns a solution. If the heuristic evaluation function, $\hat{h}$, is admissible, then the solution returned is optimal.*

Proof: Let L be the CLF. The existence of L implies the existence of some solution, call it P, with cost C. Every search node on the solution path has some finite evaluation. Let D be the maximum of all of these evaluations. Now, suppose the search does not terminate. That means it must expand nodes at arbitrarily large depths in the search graph. Since infinite action sequences have infinite cost, the cost-from-root $\hat{g}$, and thus total evaluation $\hat{f}$, of such nodes grows arbitrarily large. In particular, it grows larger than D. Best-first search would not expand such nodes before expanding every node in P, and thus discovering the solution. So we have a contradiction. Best-first search must terminate. If the heuristic is admissible, then best-first search is an instance of A* search. Thus, the solution returned must be optimal [64]. □

A similar proof works for uniform-cost search, thus:

Theorem 7.6 *For a deterministic, minimum cost-to-goal MDP with goal set G, if there exists a CLF with target set G and if infinite action sequences have infinite cost, then for any $s_0 \notin G$, uniform-cost search terminates and returns an optimal solution.*

DFBnB is usually applied to domains in which the length of any solution can be limited a priori. DFBnB is not complete for general finite-state environments because the left-most search path may never reach a goal state. This can be fixed by augmenting the algorithm with a closed list [74], but this fix does not work for infinite-state environments and, in any case, is not standard practice. If one ensures that the left-most search path does reach a goal state, and if one assumes that infinite action sequences have infinite cost, then the algorithm is complete.

Theorem 7.7 *For a deterministic, minimum cost-to-goal MDP with goal set G, if there exists a CLF, L, with target set G; and for all $s \notin G$ the first action branched upon, a, descends on L (i.e., $F(s,a) \in G$ or $L(s) - L(F(s,a)) \geq \delta$); and infinite action sequences have infinite cost, then for any $s_0 \notin G$, DFBnB terminates and returns a solution. If the heuristic evaluation function, $\hat{h}$, is admissible, then the solution returned is optimal.*

Proof: The assumption that the first action searched descends on L ensures that the left-most search path reaches G. Suppose the corresponding solution has cost C. The search never again expands a node with total evaluation greater than or equal to C. Thus, the search cannot run forever. That would imply expanding nodes at arbitrarily large depths. Because infinite action sequences have infinite cost, such nodes have unbounded cost-from-root $\hat{g}$, and thus unbounded total evaluation $\hat{f}$. The admissibility of $\hat{h}$ implies optimality of the solution returned for the same reason that it does for best-first search. □

7.2.2 Admissible Heuristics and Lyapunov Functions

Admissible heuristics play a key role in the theory of heuristic search. The admissibility of a heuristic ensures that best-first search and DFBnB return optimal solutions. Admissibility is also important in the proof of the optimal efficiency of A*. What is the relationship between Lyapunov functions and admissible heuristics? Lyapunov functions have a "first derivative" or difference property; one can always choose an action that causes the state of the environment to descend on the function and reach the goal set. The precise value of a Lyapunov function at any given state need not be meaningful. For example, if one scales a Lyapunov function by any positive constant, the result is still a Lyapunov function. This is not true of admissible heuristics. It is quite rare to be able to descend on a heuristic function at every step and reach a goal state. The defining feature of an admissible heuristic is that it does not overestimate the minimum cost to goal. A Lyapunov function would make a nice heuristic function if it were admissible, but this need not be the case.

However, if one makes additional assumptions about the MDP, then it is possible to construct an admissible heuristic function based on a Lyapunov function. A Lyapunov function naturally provides an upper bound on the number of actions required to bring the state of the system to the goal set. For a non-goal state s, this is just $\lceil L(s)/\delta \rceil$. What one needs is a lower bound on the number of actions required to bring the system to goal. Suppose that for all non-goal states, no action results in a state more than $\delta_{\max}$ lower on the Lyapunov function. Suppose also that no goal state has Lyapunov value more than M. Then for any non-goal state s the number of actions required to bring the system to goal is at least $\lceil (L(s) - M)/\delta_{\max} \rceil$. If we assume that costs are bounded below by c_{low}, then the minimum cost to goal from any non-goal state s is at least $c_{low} \lceil (L(s) - M)/\delta_{\max} \rceil$. Thus we have established the following theorem:

Theorem 7.8 *If L is a CLF with target set G for a deterministic, minimum cost-to-goal MDP with goal set G; and $\delta_{\max} = \sup_{s \notin G} \max_{a \in A(s)} (L(s) - L(F(s,a)))$ is finite; and $M = \sup_{s \in G} L(s)$ is finite; and $c_{low} = \inf_{s \notin G} \min_{a \in A(s)} C(s,a) > 0$, then*

$$h(s) = \begin{cases} \max(c_{low} \lceil (L(s) - M)/\delta_{\max} \rceil, 0) & \text{for } s \notin G \\ 0 & \text{for } s \in G \end{cases}$$

is an admissible heuristic evaluation function.

Even if a Lyapunov function is not admissible, it may be useful as a heuristic function. We return to this idea in the discussion section of this chapter.

7.2.3 Limited Look-Ahead Search

The search algorithms studied above find one or more complete solutions before returning an answer. Limited look-ahead search is targeted at on-line, real-time decision-making problems. In the usual scenario, the search algorithm must produce a "next" action to take before it has discovered a complete path to G. It should come as no surprise that Lyapunov domain knowledge is valuable for real-time decision-making. A Lyapunov function offers a measure of progress towards the goal. Given a Lyapunov function, a simple depth-one search is sufficient to determine which actions descend and allows one to construct a path to goal. In this section, we demonstrate that limited look-ahead search can be guaranteed to find solutions without restricting it to select actions that descend on a Lyapunov function.

Recall that limited look-ahead search constructs a solution by performing a sequence of limited-complexity searches. Each search generates a tree in which nodes are evaluated by the cost-from-root, $\hat{g}$, plus a heuristic cost-to-goal evaluation, $\hat{h}$. The i^{th} action of the solution is the first action on the path from the root of the i^{th} search tree (i.e., s_i) to the best leaf in that tree. Our main result is that if a CLF, L, is used *as the heuristic evaluation function* in a limited look-ahead search, and if L relates to the cost function of the MDP in a certain way, then the search is guaranteed to produce a path to goal.

Theorem 7.9 *For a deterministic, minimum cost-to-goal MDP with goal set G, if: (i) there exists a CLF, L, with target set G; (ii) L is used as the heuristic evaluation function for non-goal leaves; (iii) for all $s \notin G$ there exists $a \in A(s)$ such that $L(s) - L(F(s,a)) \geq C(s,a)$; (iv) for all i, the best leaf of the i^{th} search tree is a goal node or is expanded in the $(i+1)^{st}$ search tree; (v) infinite action sequences have infinite cost; then limited look-ahead search terminates, producing a solution.*

Condition (iii) is particularly interesting because it relates action costs to changes in the Lyapunov function. If descent on the Lyapunov function is costly, then a limited-lookahead search might never choose to take descending actions, and might never produce a path to G. Condition (iii) ensures that there is always some action that produces more descent on L than the cost incurred. Intuitively, this ensures that the Lyapunov function "outweighs" the cost function. The solution generated need not cause the state of the environment to descend on L at every step, but the state descends enough of the time to be sure of reaching G.

To prove this theorem we require some definitions and a lemma. Suppose that the first N iterations of a limited look-ahead search generate actions that produce the state sequence $s_0, s_1, \ldots, s_N$. The last state, s_N, may or may not be a goal state. For $i \in \{0, \ldots, N-1\}$, let $\hat{G}_i$ be the cost from s_0 to s_i; let l_i be the best leaf in search tree i; let C_i be the cost from from s_i to l_i in search tree i; let $c_{i,0}, \ldots, c_{i,n(i)}$ be the action costs constituting C_i; let $\hat{H}_i$ be the heuristic evaluation of l_i (i.e., $L(l_i)$ if $l_i \notin G$ and 0 otherwise); and let $\hat{F}_i = \hat{G}_i + C_i + \hat{H}_i$.

Lemma 7.10 *Under the assumptions of Theorem 7.9, for $i \in \{0, \ldots, N-2\}$, $\hat{F}_{i+1} \leq \hat{F}_i$.*

Proof: For all $i \in \{0, \ldots, N-2\}$,

$$\begin{array}{rrcl} & \hat{F}_{i+1} & \leq & \hat{F}_i \\ \iff & \hat{G}_{i+1} + C_{i+1} + \hat{H}_{i+1} & \leq & \hat{G}_i + C_i + \hat{H}_i \\ \iff & C_{i+1} + \hat{H}_{i+1} & \leq & c_{i,1} + \ldots + c_{i,n(i)} + \hat{H}_i \ . \end{array}$$

The last line follows because $\hat{G}_{i+1}$ is just $\hat{G}_i$ plus the cost of the first action from s_i to l_i, namely, $c_{i,0}$. The left-hand side is just the cost from s_{i+1} to l_{i+1} plus $\hat{h}(l_{i+1})$. The right hand side, $c_{i,1} + \ldots + c_{i,n(i)} + \hat{H}_i$, is the cost of a path from s_{i+1} to l_i plus $\hat{h}(l_i)$. If l_i is a goal node, then the same path and heuristic evaluation are part of search tree $i+1$. Since search tree $i+1$ optimizes over that and other paths, the cost of the best path in search tree $i+1$, $C_{i+1} + \hat{H}_{i+1}$, can be no worse. If l_i is not a goal node, then it is expanded in search tree $i+1$. That tree contains path(s) from s_{i+1} through l_i to one or more leaf nodes. On the path to at least one of those leaves, every action after l_i incurs less cost than the drop in L that results, by condition (iii). Thus, the evaluation of that path must be no higher than $c_{i,1} + \ldots + c_{i,n(i)} + \hat{H}_i$, and so the evaluation of the best path in tree $i+1$, $C_{i+1} + \hat{H}_{i+1}$, must also be no higher. Thus $\hat{F}_{i+1} \leq \hat{F}_i$. $\square$

Proof of Theorem 7.9: Suppose the search does not terminate. From the previous lemma, we have $\hat{F}_0 \geq \hat{F}_1 \geq \hat{F}_2 \geq \ldots$ On the other hand, $\hat{F}_i \geq \hat{G}_i$, and because we assume that infinite action sequences have infinite cost, the $\hat{G}_i$ grow without bound. This is a contradiction, so limited look-ahead search must terminate and return a solution. $\square$

A given CLF, L, may not meet condition (iii) of Theorem 7.9, or one may not know if it meets the condition. We propose two methods for generating a CLF which provably satisfies condition (iii).

The first method we propose is scaling an existing CLF. Let L be a CLF with target set G. For all $s \notin G$, let a_1 be an action that descends on L. The scaling method is applicable when

$$\inf_{s \notin G} \frac{L(s) - L(F(s,a_1))}{C(s,a_1)} = \frac{1}{\alpha} > 0 \ .$$

Consider $L' = \alpha L$. L' is obviously a CLF. Further, for any $s \notin G$,

$$\frac{L'(s) - L'(F(s,a_1))}{C(s,a_1)} = \alpha \frac{L(s) - L(F(s,a_1))}{C(s,a_1)} \geq \alpha \frac{1}{\alpha} = 1 \ .$$

Thus $L'(s) - L'(F(s,a_1)) \geq C(s,a_1)$, meaning L' satisfies condition (iii).

A potential problem with this approach is that the scaling factor, α, may not be known. In this case, we propose an on-line procedure that automatically determines an appropriate scaling factor. Note that an overestimate of α also produces a CLF satisfying condition (iii). Suppose we let $L' = \hat{\alpha}L$ for any initial guess $\hat{\alpha} > 0$. We then run a limited look-ahead search, checking during each expansion whether condition (iii) is violated. If a state, s, is encountered for which $L'(s) - L'(F(s, a_1)) < C(s, a_1)$, then the scaling factor can be updated as $\hat{\alpha} \leftarrow \dfrac{C(s, a_1)}{L(s) - L(F(s, a_1))} + \epsilon$ for some fixed $\epsilon > 0$. The first term makes the new $\hat{\alpha}$ consistent with all the states expanded so far, and the ϵ ensures that condition (iii) will be violated at most a finite number of times before $\hat{\alpha}$ becomes large enough to ensure that the search produces a solution. Note that at no time are we sure that $\hat{\alpha}$ is greater than α; but we are sure that a limited look-ahead search using this updating procedure terminates and produces a solution.

A second method for generating a CLF meeting condition (iii) is to perform roll-outs [90, 12]. Let $\mathcal{A}$ be an algorithm that assigns an action to every non-goal state and suppose that, for any s_0, $\mathcal{A}$ generates a path that reaches G. For example, if we know a CLF, L, then an obvious choice for $\mathcal{A}$ is quickest descent on L. Let $L'(s)$ be the cost of the solution generated by $\mathcal{A}$ starting at s. L' can be evaluated by actually constructing the solution. If the costs of the actions chosen by $\mathcal{A}$ always exceed some c_{low}, then the reader may verify that L' is an CLF (with descent constant $\delta = c_{low}$) and also satisfies condition (iii). Performing roll-outs can be an expensive method of heuristically evaluating leaves in a search tree, since an entire path to G must be generated for each evaluation. However, roll-outs have been found to be quite effective in both game playing and sequential control; it appears that the results often justify the effort involved [90, 12].

7.3 Pendulum Demonstration

Having established a theory of when solutions exist and when they can be found by standard search algorithms, we turn to demonstrating the theory on some example problems. Our first demonstration is in the deterministic pendulum swing-up and balance domain introduced in Section 6.1. Recall that the task is minimum-time

Action Formulation	Actions	Goal Set	Relevant Theorems
1	MEA(u_{max}), MEA($\frac{1}{2}u_{max}$)	G^2_{pend}	7.2, 7.4
2	MEA(u_{max}), MEA($\frac{1}{2}u_{max}$), $+u_{max}$, $-u_{max}$	G^2_{pend}	7.3, 7.5, 7.7, 7.9
3	$+u_{max}$, $-u_{max}$	G^2_{pend}	none
4	$+u_{max}$, $-u_{max}$, PLS	G^1_{pend}	none

Table 7.1. Four action formulations for the deterministic pendulum swing-up and balance problem.

control to the set $G^1_{pend} = \{(\theta, \dot{\theta}) : \|(\theta, \dot{\theta})\| \leq 0.01\}$. Recall also that if the pendulum has mechanical energy 2, fixing the control torque at zero allows it to swing upright. The state of the pendulum asymptotically approaches $(\theta, \dot{\theta}) = (0, 0)$, and thus enters G^1_{pend}. So, the problem of controlling the pendulum to near-upright can be reduced to the problem of ensuring that the state of the pendulum reaches $G^2_{pend} = \{(\theta, \dot{\theta}) : \mathrm{ME}(\theta, \dot{\theta}) = 2\}$. In Section 6.1 we described the MEA control law, which increases the pendulum's energy to 2. Using MEA to increase energy to 2 and then applying zero torque until the state of the pendulum reaches G^1_{pend} is one solution to the swing-up problem.

In order to apply search to a domain like the pendulum, in which a controller can in principle apply any of a continuum of control torques, it is necessary to limit the number of control choices to a finite, preferably small, set. We present four alternative action formulations for controlling the pendulum, which rely to differing degrees on Lyapunov domain knowledge. In each of these action formulations, each action corresponds to controlling the pendulum according to a fixed control law, a mapping from pendulum state to control torque, for a fixed period of time Δ or until a goal state is reached, whichever happens first.

We describe the four action formulations in order of decreasing dependence on Lyapunov domain knowledge. They are summarized in Table 7.1. The first action formulation uses the G^2_{pend} goal set and has two actions corresponding to the control laws: MEA(u_{max}) and MEA($\frac{1}{2}u_{max}$). By MEA(w), we mean the MEA control law with parameters $\mu = w$, $\epsilon_\theta = \frac{1}{2}(\sin^{-1}(w) - \sin^{-1}(\frac{1}{2}w))$, and $\epsilon_{\dot\theta} = 0.1$. By Theorem 6.1, both of these actions cause the state of the pendulum to descend on $L_{pend}(\theta, \dot\theta) = 2 - \mathrm{ME}(\theta, \dot\theta)$ outside of G^2_{pend}. Thus L_{pend} is a CLF for action formulation 1, and all (both) actions descend on L_{pend} outside of G^2_{pend}. Theorem 7.2 guarantees the existence of optimal solutions under this formulation.

Action formulation 2 also uses the G^2_{pend} goal set, but includes two more actions, corresponding to constant-torque control laws $u = +u_{max}$ and $u = -u_{max}$. These two actions do not always cause the state of the pendulum to descend on L_{pend}, but L_{pend} remains a CLF because of the MEA-based actions. Because infinite action sequences have infinite cost, Theorem 7.3 guarantees the existence of optimal solutions.

Action formulation 3 drops the MEA-based actions, leaving just $\pm u_{max}$. Our Lyapunov-based theory offers no guarantee that the problem is solvable with these simple actions, though the experiments we present below indicate that it is. Given this empirical observation and the fact that infinite action sequences have infinite cost, the existence of an optimal solution follows.

Action formulation 4 uses the G^1_{pend} goal set. In pilot experiments, we found that it was extremely difficult or impossible to control the state of the pendulum until it reaches this goal set using just the $\pm u_{max}$ constant-torque actions. We developed a control law which we call PLS for "pendulum linear saturating" controller. It is a standard LQR controller based on a linear approximation to the pendulum dynamics around the upright, zero-velocity state (see, e.g., Vincent & Grantham [96]). The LQR controller's signal is constrained to stay in the range $[-u_{max}, u_{max}]$, resulting in the control law:

$$\mathrm{PLS}(\theta, \dot\theta) = \max(\min(-2.41421\theta - 2.1974\dot\theta, u_{max}) - u_{max})$$

For some region of states surrounding G^1_{pend}, PLS is sufficient to "pull" the state of the pendulum in to G^1_{pend}. Action formulation 4 contains three actions, corresponding to the constant-torque control laws $+u_{max}$ and $-u_{max}$, and to PLS. We suspect that under this formulation G^1_{pend} is reachable from any $s_0 \notin G^1_{pend}$. If that is so, the existence of optimal solutions follows from the fact that infinite action sequences have infinite cost.

Under all action formulations, the cost of a non-terminal action of duration Δ is just Δ. The first three action formulations use the G^2_{pend} goal set. For those, the cost of an action leading to a state $s \in G^2_{pend}$ is equal to the time it takes for the state of the pendulum to reach s, plus the time it takes for the pendulum to swing up afterwards (i.e. until its state enters G^1_{pend}). Under action formulation 4 the cost of an action leading to a state $s \in G^1_{pend}$ is just the time it takes for the state of the pendulum to reach s. For all action formulations, then, the costs of actions are designed so that an optimal solution corresponds to a minimal-time trajectory to G^1_{pend}.

7.3.1 Experiments

For each action formulation, we experimented with a number of different search procedures: A* or uniform-cost search, DFBnB, and RFDS with three different heuristic functions. For formulations 1 through 3, which use the G^2_{pend} goal set, we ran A* and DFBnB searches using a heuristic function which we call "admissible energy deficit" (AED):

$$\mathrm{AED}(\theta, \dot\theta) = \begin{cases} (2 - \mathrm{ME}(\theta, \dot\theta))/(2u_{max}) & \text{if } \mathrm{ME}(\theta, \dot\theta) < 2 \\ 0 & \text{otherwise} \end{cases}$$

The term $2 - \mathrm{ME}(\theta, \dot\theta)$ represents the amount of energy that must be added to the pendulum in order for its state to reach G^2_{pend}. For any state with $\mathrm{ME}(\theta, \dot\theta) < 2$, $\dot\theta < 2$. So the maximum rate at which energy can be added to the pendulum is

$2u_{max}$. AED is a lower bound on the time to reach G^2_{pend}, and so is an admissible heuristic. We note that AED is constructed using essentially the approach described in Theorem 7.8, but we have analyzed the pendulum domain at the continuous-time level, rather than at the discrete-time, action-based level. For formulation 4, which uses the G^1_{pend} goal set, we ran uniform-cost search and DFBnB with $\hat{h} = 0$.

Theorems 7.4, 7.5, and 7.7 guarantee that A* and DFBnB searches find solutions and terminate under action formulations 1 and 2. A priori, we had no such guarantee for the other action formulations. We ran A*, uniform-cost search and DFBnB for the choices of action duration: $\Delta \in \{1, 1.5, 2, 3, 4\}$.

Recall that in Section 4.1.3 we described a particular limited look-ahead search procedure call repeated fixed-depth search (RFDS) in which the search tree built at each iteration has fixed depth. We ran RFDS at $\Delta = 1$ and search depths $d \in \{1, 2, 3, 4, 5\}$ with three different heuristic functions. RFDS-Z uses the zero heuristic. Theorem 7.4 guarantees that RFDS-Z finds a solution under the action formulation 1, but we knew of no guarantees for the other action formulations beforehand.

RFDS-S uses a scaled version of the AED heuristic to evaluate leaves. AED itself does not meet condition (iii) of Theorem 7.9. We did not know an appropriate scaling factor, so we used the on-line updating method described near the end of Section 7.2.3. We began each run of RFDS with a scaling factor of $\hat{\alpha} = 1$. Whenever a state was expanded for which action one (MEA(u_{max}) for formulations 1 and 2, and $+u_{max}$ for formulations 3 and 4) produced a descent δ which was smaller than the cost incurred c, the scaling factor was updated as $\hat{\alpha} \leftarrow c/\delta + 0.001$. When δ was non-positive, which is possible only under formulations 3 and 4, no update was performed. The scaling approach does not make much sense under these two formulations since a_1 is not guaranteed to descend on AED, but we ran the experiments for information purposes. Under formulations 1 and 2, updating the scaling factor ensures that the search eventually returns a solution.

The last heuristic with which we experimented is based on the roll-out idea described at the end of Section 7.2.3. From any pendulum state with mechanical energy less than 2, control according to MEA(u_{max}) brings the state of the pendulum to G^2_{pend}. For the first three action formulations, which use G^2_{pend}, RFDS-R evaluated non-goal states with the time it took MEA to bring the state of the pendulum to G^2_{pend} plus the time it took for the state of the pendulum to reach G^1_{pend}. For action formulation 4, MEA alone does not suffice since the pendulum can have mechanical energy greater than 2. MEA, which only increases the pendulum's energy, does not bring the state of the pendulum to the goal set in such situations.

In Section 6.2 we introduced the MEto2 control law, which reduces the pendulum's energy whenever it is above 2 and increases it whenever it is below 2. We use this control law, with constant $\epsilon_{ME} = 0$, as the roll-out controller for the RFDS-R runs under action formulation 4.

For action formulations 1 and 2, Theorem 7.9 guarantees that RFDS-R finds solutions. This guarantee does not extend to action formulations 3 or 4. The roll-out is not necessarily a CLF for those action formulations because the roll-outs are based on control laws that are not part of those action formulations.

The initial state for all searches was the hanging-down rest position, $(\theta_0, \dot{\theta}_0) = (\pi, 0)$. The RFDS searches were run for a maximum of 1000 iterations, and terminated if no solution was found by that time. We believe our results are not sensitive to this particular threshold. In pilot studies, all the runs we observed either terminated with far fewer major iterations, or appeared to enter a cyclic behavior that would never produce a solution. A*, uniform-cost search, and DFBnB searches were run for as long as was feasible. One A* search had to be terminated after it filled up main memory and started thrashing badly. Several other A* searches crashed, we believe because of memory problems. We eventually ran them successfully on more powerful machines. We also chose to terminate one of the DFBnB searches under action formulation 2 when it had not completed after running for over two weeks.

7.3.2 Results

Table 7.2 presents the results of the experiments. For each search algorithm, set of parameter settings, and action formulation, the table either reports the cost of the solution found and the search effort required, or "dnc," which indicates that the search did not complete. Search effort was measured by the number of seconds of pendulum dynamics that were simulated. For most searches, this correlated closely with more traditional measures of complexity such as the number of nodes expanded or number of actions searched. The complexity of RFDS-R, which simulates entire trajectories to evaluate leaves, appears misleadingly low if measured by node expansions or actions searched. Hence, we settled on simulation time as the most fair measure of search effort. A triangle (◤) in the upper-left-hand corner of a group of results acts to remind the reader which searches should have completed eventually, given unbounded computational resources.

Despite no Lyapunov-based assurances, we found that A* terminated under action formulation 3 and uniform-cost search terminated under action formulation 4 for all choices of Δ. This was not surprising, really, since we had determined by prior experiments that the problem could be solved under these action formulations, at least for some choices of Δ. Except for the case $\Delta = 4$, the solutions found by A* under formulation 3 were slightly better than under formulation 1. The solutions found under formulation 2 were better than either of those. Since the actions in formulation 2 strictly include the actions in formulations 1 and 3, the solutions certainly could not be any worse. However, the best solutions of all were found under formulation 4. At $\Delta = 1$, both uniform cost-search and RFDS-R discovered trajectories that take 18.07 seconds to reach G^1_{pend}. Because formulation 4 uses that goal set instead of

Search	Action Formulation 1		2		3		4	
A* for 1-3, uniform-cost search for 4								
$\Delta = 4$	21.70	497	21.50	1.30×10^4	23.64	368	29.33	5367
$\Delta = 3$	21.79	1209	20.92	3.68×10^4	21.22	534	19.38	1017
$\Delta = 2$	21.17	1.04×10^4	20.02	4.54×10^5	20.13	2467	19.03	4572
$\Delta = 1.5$	20.94	8.22×10^4	19.74	6.33×10^6	19.88	1.15×10^4	18.08	1.90×10^4
$\Delta = 1$	20.80	8.08×10^6	dnc		19.33	2.05×10^5	18.07	2.74×10^5
DFBnB								
$\Delta = 4$	21.70	497	21.50	1.35×10^4	dnc		dnc	
$\Delta = 3$	21.79	1226	20.92	5.68×10^4	dnc		dnc	
$\Delta = 2$	21.17	1.04×10^4	20.02	2.63×10^6	dnc		dnc	
$\Delta = 1.5$	20.94	8.41×10^4	19.74	1.49×10^8	dnc		dnc	
$\Delta = 1$	20.80	8.17×10^6	dnc		dnc		dnc	
RFDS-Z								
d=1	24.19	73	66.91	742	dnc		dnc	
d=2	24.18	165	102.87	3835	dnc		dnc	
d=3	23.67	337	100.77	1.18×10^4	dnc		dnc	
d=4	23.11	663	66.75	2.36×10^4	dnc		dnc	
d=5	22.50	1218	45.25	5.17×10^4	dnc		dnc	
RFDS-S								
d=1	24.19	73	22.97	123	22.91	62	dnc	
d=2	22.46	153	20.81	405	20.45	119	dnc	
d=3	22.68	320	21.70	1613	24.63	349	dnc	
d=4	23.11	663	21.96	6075	22.13	541	dnc	
d=5	21.92	1188	20.33	2.45×10^4	20.77	966	dnc	
RFDS-R								
d=1	21.50	559	19.51	934	19.33	468	18.07	758
d=2	20.80	948	19.51	3667	19.33	944	18.07	2326
d=3	20.80	1817	19.51	1.41×10^4	19.33	1862	18.07	6943
d=4	20.80	3460	19.51	5.37×10^4	19.33	3660	18.07	2.05×10^4
d=5	20.80	6520	19.33	1.96×10^5	19.33	7124	18.07	5.96×10^4

Table 7.2. Results of search experiments in the pendulum domain: solution cost and search effort (measured in seconds of pendulum dynamics simulated).

G^2_{pend}, faster swing-up can be obtained by continuing to accelerate the pendulum after ME > 2, and decelerating near the end, to approach the upright, balanced position.

From the table of results there is no apparent benefit of DFBnB compared to A*. However, as noted above, several of our A* runs crashed the first time we ran them, and we had to use more powerful machines so that they could complete. Though DFBnB searches took longer than A* searches, their modest memory requirements allowed them to complete the first time with no problem. Run-time is generally a softer constraint than memory usage, and so DFBnB is more useful in some situations.

DFBnB found no solutions under the second two action formulations because the left-most search paths correspond to continuously applying a constant torque of u_{max} to the pendulum. This does not bring the state of the pendulum to the goal set. Thus, a first solution was never discovered and DFBnB did not terminate.

All the RFDS runs under the first two action sets completed, in many cases after a fairly modest search effort. The results for RFDS-R are particularly impressive. Optimal or near optimal solutions were generated for all action formulations, even with depth one search. Roll-outs appear to provide powerful heuristic information. The results of RFDS-R under formulation 2 are the best generated for that action formulation. We do not know the optimal solution under formulation 2 since A* and DFBnB did not complete at $\Delta = 1$. These results confirm other reports in the literature about the empirical value of roll-outs [90, 12].

7.4 Arm Demonstration

In this section, we apply heuristic search to the deterministic robot arm problem introduced in Section 6.3. Recall that the task is to move the state of the arm to the goal set $G_{arm} = \{(\Theta, \dot{\Theta}) : \|(\Theta, \dot{\Theta})\| \leq 0.01\}$, which is a set of states near to a stationary, horizontally-extended configuration (see Figure 7.1). Below, we describe a number of control laws for controlling the arm. The cost of applying a control law

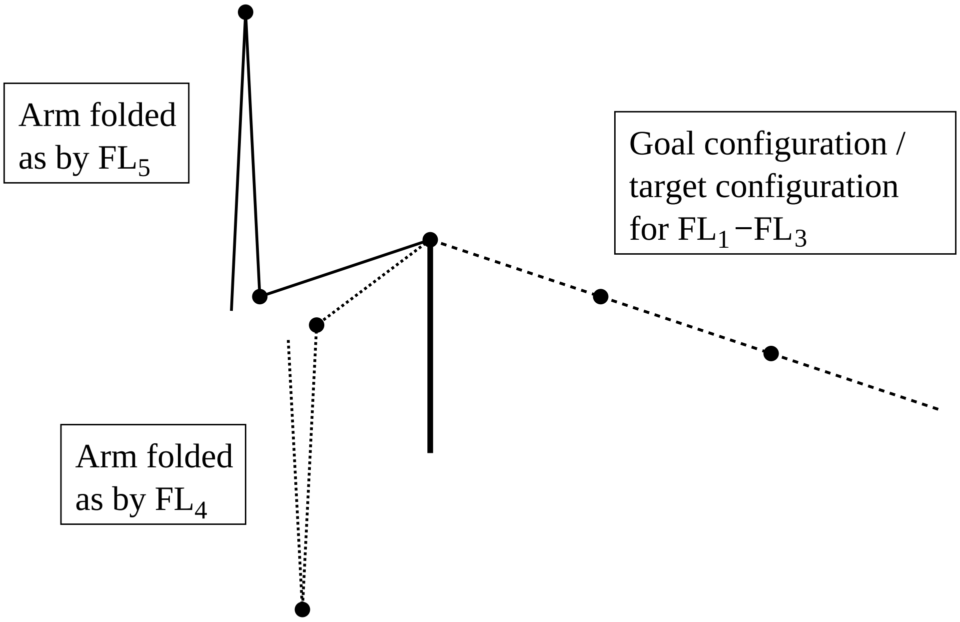

Figure 7.1. Goal configuration and target configurations for several controllers.

for a period of Δ time is the continuous-time integral: $\int_{t=0}^{\Delta} \|\Theta(t)\|^2 + \|\tau(t) - \tau_0\|^2 dt$, where $\Theta(t)$ is the position of the arm as a function of time, $\tau(t)$ is the torque applied as a function of time, and τ_0 is the torque required to hold the arm at the origin. A good controller moves the state of the arm to the origin, thus avoiding accumulating a large penalty from the first term, but must not use excessive torque in doing so.

In Section 6.3.2, we proposed a control law, FL_1, based on feedback linearization and LQR methods. We showed that this control law brings the state of the arm to G_{arm} from any initial state. Below, we present several other control laws based on the feedback linearization-LQR approach, with varying choices of gain matrices and target points. Let $\mathrm{LQR}(Q, R)$ stand for the gain matrix resulting from LQR design with penalty matrices Q and R; let $Q_0 = \mathrm{diag}(1, 1, 1, 0.01, 0.01, 0.01)$; let R_0 be the 3×3 identity matrix; let $K{=}\mathrm{LQR}(Q_0, R_0)$; let $\Theta_4 = [0, -\frac{1}{2}\pi, \pi]'$ and $\Theta_5 = [0, \frac{1}{2}\pi, -\pi]'$; and let $Z([x, y, z]') = [0, y, z]'$. Our first five control laws for the arm are:

$$\mathrm{FL}_1(\Theta, \dot\Theta) = -H(\Theta, \dot\Theta) K \begin{bmatrix} \Theta \\ \dot\Theta \end{bmatrix} + V(\Theta, \dot\Theta) + \mathcal{G}(\Theta) \ ,$$

$$\mathrm{FL}_2(\Theta,\dot{\Theta}) = -H(\Theta,\dot{\Theta})K_2 \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} + V(\Theta,\dot{\Theta}) + \mathcal{G}(\Theta) \text{ where } K_2{=}\mathrm{LQR}(4Q_0, R_0) \ ,$$

$$\mathrm{FL}_3(\Theta,\dot{\Theta}) = -H(\Theta,\dot{\Theta})K_3 \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} + V(\Theta,\dot{\Theta}) + \mathcal{G}(\Theta) \text{ where } K_3{=}\mathrm{LQR}(Q_0, 4R_0) \ ,$$

$$\mathrm{FL}_4(\Theta,\dot{\Theta}) = Z \left(-H(\Theta,\dot{\Theta})K \begin{bmatrix} \Theta - \Theta_4 \\ \dot{\Theta} \end{bmatrix} + V(\Theta,\dot{\Theta}) + \mathcal{G}(\Theta) \right) \ ,$$

$$\mathrm{FL}_5(\Theta,\dot{\Theta}) = Z \left(-H(\Theta,\dot{\Theta})K \begin{bmatrix} \Theta - \Theta_5 \\ \dot{\Theta} \end{bmatrix} + V(\Theta,\dot{\Theta}) + \mathcal{G}(\Theta) \right) \ .$$

The first three control laws move the state of the arm to the origin, but at different rates. Compared to FL_1, the gain matrix for FL_2 is based on an LQR design that places more penalty on the distance of the state of the arm from the origin. Thus, FL_2 tends to exert a greater control torque than FL_1, moving the state of the arm to the goal more quickly. The LQR design for FL_3 is based on a smaller penalty on the distance of the state of the arm from the origin, thus FL_3 tends to exert smaller controls torques than FL_1, bringing the state of the arm to the origin more slowly. FL_4 and FL_5 do not apply torque to the first joint of the arm, but pull the outer two links into a folded position, as depicted in Figure 7.1. Although these control laws do not, by themselves, bring the state of the arm to G_{arm}, we expected that they would be useful as part of a solution involving other control laws. By pulling the arm inward, its moment of inertia is reduced, allowing it to swing more easily around its first joint.

As discussed in Section 6.3.2, FL_1 causes the state of the arm to descend on the quadratic Lyapunov function

$$L_{arm} = [\Theta' \dot{\Theta}'] P \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix} \ ,$$

Action Formulation	Actions	Relevant Theorems
1	FL_1, FL_2, FL_3, FL_4, FL_5	7.3, 7.6, 7.7, 7.9
2	FL_1, LD[FL_2], LD[FL_3], LD[FL_4], LD[FL_5]	7.2, 7.4

Table 7.3. Two action formulations for robot arm control.

where P is the symmetric, positive definite matrix defined in that section. The other control laws above do not necessarily cause the state of the arm to descend on L_{arm} from any non-goal state. However, by a simple transformation we can produce a related control law that does cause descent. Let C be any control law. We define the L_{arm}-descending version of C to be:

$$\mathrm{LD[C]}(\Theta,\dot{\Theta}) = \begin{cases} \mathrm{C}(\Theta,\dot{\Theta}) & \text{if applying C causes } \frac{d}{dt}L_{arm}(\Theta,\dot{\Theta}) < -0.1 \\ & \text{and if } L_{arm}(\Theta,\dot{\Theta}) > 0.1 \\ \mathrm{FL}_1(\Theta,\dot{\Theta}) & \text{otherwise} \end{cases}$$

In other words, the control law LD[C] checks at each instant of time whether C causes sufficiently fast descent on L_{arm}. If so, and if the state of the arm is not near the origin already, then LD[C] controls the arm just as C would. Otherwise, LD[C] reverts to controlling the arm as FL_1 would. By construction, LD[C] causes the state of the arm to descend on L_{arm}.

We propose two action formulations for controlling the arm, as summarized in Table 7.3. Both formulations include FL_1 and so L_{arm} is a CLF for both. In formulation 2, all actions cause the state of the arm to descend on L_{arm}.

7.4.1 Experiments

For both action formulations we ran uniform-cost search and DFBnB with $\hat{h} = 0$ for $\Delta \in \{0.375, 0.5, 0.75, 1.0, 1.5, 2.0\}$. Recall that choosing an action means the arm is controlled by the corresponding control law for Δ time. We used $\hat{h} = 0$ only for lack of a better choice. For this domain, a heuristic designed according to Theorem 7.8 is not very helpful, as the rate of decrease in L_{arm} for some states and actions is very large, and the minimal cost incurred from some states and actions is very small. The heuristic is very flat, yielding little or no useful information.

We ran RFDS with $\Delta = 0.5$ using three different heuristics: (1) the zero heuristic, (2) a scaled version of L_{arm}, and (3) roll-outs based on FL_1. We designate these cases RFDS-Z, RFDS-S, and RFDS-R respectively. For RFDS-S, we did not know an appropriate scaling factor. Since the state of the arm stays bounded under FL_1, $\frac{d}{dt}L_{arm}$ is bounded below zero and cost accumulates at a rate that is bounded above.[1] So, for sufficiently large $\hat{\alpha}$, using $\hat{\alpha}L_{arm}$ as a heuristic ensures that RFDS finds a solution. We used the method described in Section 7.2.3 to find an appropriate scaling factor on-line, with an initial scaling factor of $\hat{\alpha} = 1$ and an increment of $\epsilon = 0.001$.

Instead of having a single initial state, we measured performance across a set of nine initial states with zero velocity and with joint positions of the form $[x, y, -y]$, where $x \in \{-\frac{1}{3}\pi, -\frac{2}{3}\pi, \pi\}$ and $y \in \{-\frac{1}{2}\pi, 0, \frac{1}{2}\pi\}$. The RFDS runs were allowed a maximum of 100 iterations to produce a solution, and were terminated if no solution was produced by that time.

RFDS-Z with action formulation 1 is the only case for which finding solutions is not guaranteed under the assumption of unbounded computational resources. Uniform-cost search and DFBnB are guaranteed to produce solutions under formulation 1 by Theorems 7.6 and 7.7. RFDS-S and RFDS-R are guaranteed to produce solutions under formulation 1 by Theorem 7.9. All searches find solutions under formulation 2 by Theorem 7.4.

---

[1]In the proof of Theorem 6.4, we showed that FL_1 keeps the state of the arm bounded in the case of stochastic dynamics. The same holds for the deterministic dynamics and for any of the other controllers discussed here.

7.4.2 Results

Table 7.4 presents the results of the experiments. For each algorithm and each action formulation, we report the average solution cost across the nine initial states and the average search effort, measured in seconds of arm dynamics simulated. For smaller choices of Δ, some of the uniform-cost and DFBnB searches were terminated when they took too long. (We allowed at least two weeks for each set of nine searches, but we did not have a rigorous criterion for terminating runs early.) Simulating the arm dynamics is a fairly complex process, and our code was not optimized for speed. Run-time, and not memory usage, was the limiting factor in all of the searches that had to be terminated.

Excepting RFDS-Z, even the simplest searches produced better solutions than FL_1 alone, for which the average solution cost is 434.9. For every parameterization of every search algorithm, the solutions found under formulation 1, if any, were better than the solutions found under formulation 2. As was the case in the pendulum domain, it appears that restricting the action choices to descend on the Lyapunov function limits performance.

Between uniform-cost search and DFBnB, the best solutions were found under formulation 1 at $\Delta = 0.75$, with an average cost of 266.4. Even at twice the decision frequency ($\Delta = 0.375$), the solutions found by uniform-cost search under formulation 2 were not as good. Because time, and not space, was the limiting factor in these searches, and also because we relied on the zero function as a heuristic, DFBnB offered no significant advantages over uniform-cost search. The case of uniform-cost search with action formulation 2 is notable because it succeeds in finding solutions for the smallest choice of Δ, and it does so with relatively little search effort. We attribute this good scaling to the fact that control laws of the form LD[C] act as FL_1, and not as C, whenever C does not descend on L_{arm}. Because our implementation of uniform-cost search checks for duplicate siblings and prunes them away, the branching factor

Search		Action Formulation			
		1		2	
Uniform-cost search	$\Delta = 2.0$	333.8	96	333.5	40
	$\Delta = 1.5$	304.3	216	325.3	50
	$\Delta = 1.0$	283.1	1152	303.7	86
	$\Delta = 0.75$	266.4	1.14×10^4	290.9	162
	$\Delta = 0.5$	dnc		280.2	994
	$\Delta = 0.375$	dnc		277.1	1.04×10^4
DFBnB	$\Delta = 2.0$	333.8	358	333.5	444
	$\Delta = 1.5$	304.3	975	325.3	1917
	$\Delta = 1.0$	283.1	6219	303.7	3.81×10^4
	$\Delta = 0.75$	266.4	4.70×10^4	290.9	8.78×10^5
	$\Delta = 0.5$	dnc		dnc	
	$\Delta = 0.375$	dnc		dnc	
RFDS-Z $\Delta = 0.5$	d=1	dnc		348.5	33
	d=2	dnc		291.5	89
	d=3	dnc		282.3	232
	d=4	dnc		280.6	576
RFDS-S $\Delta = 0.5$	d=1	313.5	49	313.7	29
	d=2	310.1	215	313.2	89
	d=3	301.2	848	301.6	264
	d=4	291.7	2839	300.0	758
RFDS-R $\Delta = 0.5$	d=1	280.9	453	287.4	245
	d=2	287.5	1612	298.9	720
	d=3	278.6	5188	291.4	2011
	d=4	253.7	1.68×10^4	280.6	4751

Table 7.4. Results of search experiments in the robot arm domain: solution cost and search effort (measure in seconds of arm dynamics simulated) averaged across the nine initial states.

of the search is effectively reduced in states where some control laws do not descend on L_{arm}. Importantly, this happens for FL_4 and FL_5 when the state of the arm is near G_{arm}, which is deep in the search tree, at the time when a smaller branching factor is the most helpful. Our implementation of DFBnB does not check for duplicate siblings, and one can see that search effort increases much more rapidly under formulation 2. The search effort under formulation 1 is more moderate for both uniform-cost search and DFBnB.

None of the RFDS-Z searches under action formulation 1 produced solutions in the first 100 major iterations. In all cases, we observed that the search procedure got stuck in a cycle alternating between the FL_4 and FL_5 actions, with the arm mostly outstretched. This "wiggles" the arm up and down, without incurring too much torque penalty and only a modest position deviation penalty. The other control laws incur large control torque penalties due to the torque they apply at the first joint. This makes them appear bad to a shallow search. We verified that with sufficiently deep search, depth 6 in particular, RFDS-Z does construct a path to G_{arm} from all nine initial states.

The best solutions overall were found by RFDS-R under formulation 1, with an average cost of 253.7. Considerable search effort was involved, approaching the effort of the largest uniform-cost and DFBnB searches that completed. The shorter RFDS searches might reasonably be called "real-time". The longest RFDS searches took approximately 10 minutes apiece, which is not an appropriate time scale for reactive control of a robot arm.

In the pendulum domain, we found that a depth one search with roll-outs to evaluate leaves was very effective, producing optimal or near optimal solutions under every action set. In the robot arm domain, the story is more complex. Depth one search with roll-outs was as good, or nearly as good, as depth 2 or depth 3 searches with roll-outs. Depth 4 search with roll-outs was significantly better, under both action formulations. The RFDS-Z and RFDS-S searches seem best at producing solutions of moderate cost with very little search effort. They outperformed uniform-cost search, DFBnB, and RFDS-R in that respect.

7.5 Discussion

In this chapter we have demonstrated that state-space search is feasible and potentially useful for infinite-state control problems, if Lyapunov domain knowledge is available. Lyapunov domain knowledge ensures the existence of optimal solutions and provides conditions for the completeness of a variety of standard heuristic search algorithms, including best-first search, uniform-cost search, depth-first branch-and-bound, and limited look-ahead search. Similar results can be obtained for related algorithms, such as iterative-deepening A*. We also discussed connections between Lyapunov functions, admissible heuristics, and roll-outs.

In Sections 7.3 and 7.4, we demonstrated the process of designing finite action sets for continuous-state, continuous-time control problems so that solution-existence and completeness criteria are satisfied. In experiments, we made several important observations. One is that restricting attention to solutions that strictly descend on a Lyapunov function can limit performance. We return to this issue in subsequent chapters. We also noted that using roll-outs for heuristic leaf evaluations in limited look-ahead search resulted in optimal or near-optimal solutions, and sometimes required significantly less search effort than A*, uniform-cost search, and DFBnB. This was particularly the case for the pendulum swing-up task.

In Section 7.2.2 we claimed that a Lyapunov function may be useful as a heuristic functions even if it is not admissible. One reason for this is that using a Lyapunov function as a heuristic provides one approach to anytime heuristic search, in which one desires to find a solution of some quality quickly and find solutions of increasing quality as long as search is allowed to continue. One approach would be to perform a sequence of best-first searches with different scalar multiples of a Lyapunov function as the heuristic. Theorem 7.5 would ensure that each search produces a solution. If the Lyapunov function is scaled to be "large" with respect to the MDP's cost function, then a best-first search mostly descends on the Lyapunov function. This produces a solution quickly, though its cost may be high. If the Lyapunov function is smaller compared to the MDP's cost function, then a best-first search is more sensitive to cost, and one expects a lower-cost solution in return for greater search effort. So, if one performs a series of best-first searches in which the Lyapunov function used for the heuristic is scaled down in successive searches, one would expect to see a sequence of solutions of increasing quality generated in anytime fashion.

We close by noting a complementarity between the admissibility property and the properties of a heuristic function that make limited look-ahead search complete. Admissible heuristics are usually monotonic, meaning that the total evaluations, $\hat{f}$, of states along any path from s_0 are non-decreasing. Admissible heuristics without this property can always be modified on-line to create this property [64, 74]. In contrast, Lemma 7.10 states that the opposite is true under the conditions we used to establish the completeness of limited look-ahead search. The estimates of the value of the solution path as it is constructed by a limited look-ahead time search, $\hat{F}_i$, is non-increasing.

CHAPTER 8

LYAPUNOV FUNCTIONS FOR REINFORCEMENT LEARNING AGENTS

In the previous chapter, we took a detailed look at the implications of Lyapunov domain knowledge for heuristic search approaches to deterministic optimal control. In this chapter we take a much broader view. We consider stochastic control problems as well as deterministic ones, and we focus on guarantees that can be established making little or no assumptions about how the agent chooses actions.

Such guarantees are particularly useful for agents that learn solutions on-line. A learning agent's behavior is typically a complex function of the state of the environment and the history of the agent. It is difficult to predict how a learning agent will behave in a given situation, even if one knows the exact learning algorithm and internal representations used by the agent. Making predictions for a class of learning agents is even harder. Instead, we look at methods for establishing provable safety and performance guarantees based on Lyapunov domain knowledge and characteristics of the MDP—guarantees which hold for any agent behavior.

In Chapter 7 the primary theoretical issues we sought to address are the existence of solutions and the completeness of various state space search algorithms. Properties such as the reachability of the goal set are important in the present setting too. However, when learning agents are applied to on-line control tasks, other concerns arise: Does the agent keep the state of the environment in a safe, acceptable region of state space? Does the agent bring the state of the environment to the goal quickly and reliably on every attempt? Does the agent avoid incurring excessive costs during learning? Does the agent learn quickly? In Section 8.1 we provide Lyapunov-based tools for formulating MDPs so that some of these problems can be addressed. In Sections 8.2 through 8.4, we present Lyapunov-based designs and experiments in the four problem domains introduced in Chapter 6.

8.1 General Guarantees Based on Lyapunov Functions

Let the MDP under consideration be fixed. Let $L : S \to \mathbb{R}$; let s_t be the state of the environment at time $t \geq 0$; let $T \subset S$ be a set of "desirable" states; and let $\tau(t)$ be the first time the state of the environment is in T on or after time t, that is:

$$\tau(t) = \begin{cases} 0 & \text{if } s_t \in T, \\ t' & \text{if } s_{t'} \in T \text{ and } s_i \notin T \text{ for } t \leq i < t' \\ \infty & \text{otherwise} \end{cases}$$

Suppose that an agent chooses actions by any means whatsoever (e.g., according to a fixed policy, according to some learning rule, or simply randomly). One way to guarantee that the state of the environment enters T is if all actions descend on L outside of T.

Theorem 8.1 *If L is a CLF with target set T and if all actions descend on L, then for all $t \geq 0$, if $s_t \notin T$ then $\tau(t) - t \leq \lceil L(s_t)/\delta \rceil$ and for all $i \in \{t, \ldots, \tau(t) - 1\}$, $L(s_i) \leq L(s_t) - \delta(i - t)$.*

That is, if all actions descend on a CLF, L, then from any state s_t, the state of the environment reaches T within $\lceil L(s_t)/\delta \rceil$ time steps, and in doing so it descends on L by at least δ every time step. We omit proof, but note that an argument similar to the proof of Theorem 5.3 suffices.

If T is a goal set, then this theorem provides a guarantee of goal achievement from any initial state. If the states in T are not terminal, then the state of the environment may reach T but not stay in T forever. However, if the state of the environment leaves T, the theorem implies that it will surely return eventually. The fact that the state of the environment descends monotonically on L also yields desirable safety properties. For example, in many continuous-state control problems level sets of a Lyapunov function are bounded and lower level sets are "smaller" than higher level sets. So if the state of the environment descends on L, we know that the state stays bounded and is contained in "shrinking" subsets of state space en route to T.

If L is a CLF but not all actions descend, then there is no guarantee that the state of the environment reaches T from any initial state. However, with one or two additional assumptions, some guarantees are possible.

Theorem 8.2 *If L is a CLF with target set T; and $0 < L(s) < U \in \mathbb{R}$ for all $s \notin T$; and there exists $p_1 > 0$, $p_2 > 0$, and $\delta > 0$ such that with probability at least p_1 on every time step the agent chooses an action that with probability at least p_2 causes the state of the environment to descend on L by an amount δ, then with probability one, for all $t \geq 0$, $\tau(t) < \infty$. The probability that $\tau(t) - t \geq n$ for any $n \geq 0$ is bounded above by a function that decays to zero exponentially in n.*

This theorem is true for essentially the same reason that Theorem 5.4 is true.

Proof: Fix $t \geq 0$ for which $s_t \notin T$. Under the assumptions of the theorem, there is probability at least $p = p_1 p_2 > 0$ on every time step that if the state of the environment is outside of T, the next state of the environment is in T or lower on L by at least δ. Let $k = \lceil U/\delta \rceil$. If the state of the environment does not enter T within k time steps of t, it must be that the state of the environment did not descend on L during at least one of those k time steps. (Otherwise, $L(s_{t+k}) \leq L(s_t) - \delta k \leq U - \delta \lceil U/\delta \rceil \leq 0$, which implies $s_{t+k} \in T$.) The probability that at least one of those k steps does not descend on L is no more than $1 - p^k$. The probability that the state of the environment does not enter T in the first $2k$ time steps after t is no more than $(1-p^k)^2$. More generally, the probability that the state of the environment does not enter T in the first jk time steps is no more than $(1-p^k)^j$. The probability that the state of the environment never enters T is no more than $\lim_{j\to\infty}(1-p^k)^j = 0$. $\square$

These guarantees are not nearly as strong as those of the previous theorem, yet they are reassuring. It is a non-trivial assumption that L is bounded above and that δ descent is possible from any state with at least some probability. In essence, it means that the state of the environment cannot get arbitrarily "far away" from T—it is always within $\lceil U/\delta \rceil$ steps. In the examples below, we use various means to ensure that a Lyapunov function is bounded above on the set of reachable states. Being able to show this kind of boundedness can be an important factor in deciding what actions the agent is allowed to take.

In summary, if all actions guarantee descent on a Lyapunov function with target set T, then the state of the environment is guaranteed to reach T in bounded time. If there is some non-zero probability of descent on a Lyapunov function at every step and if the Lyapunov function is bounded above, then the state of the environment reaches T eventually, and the probability of waiting n steps for the state of the environment to enter T decays exponentially in n.

8.2 Deterministic Pendulum Demonstration

In Section 7.3 we solved the deterministic pendulum swing-up and balance task using state-space search methods. Here, we take a reinforcement learning approach. We begin by briefly reviewing the problem definition and the four action formulations we proposed for controlling the pendulum (see Sections 6.1 and 7.3 for more details.) We then discuss what sorts of guarantees can be established for reinforcement learning agents using those problem formulations. In Sections 8.2.2 and 8.2.3 we describe and present the results of experiments in which reinforcement learning agents learned to control the pendulum.

Action Formulation	Actions	Goal Set
1	MEA(u_{max}), MEA($\frac{1}{2}u_{max}$)	G^2_{pend}
2	MEA(u_{max}), MEA($\frac{1}{2}u_{max}$), $+u_{max}$, $-u_{max}$	G^2_{pend}
3	$+u_{max}$, $-u_{max}$	G^2_{pend}
4	$+u_{max}$, $-u_{max}$, PLS	G^1_{pend}

Table 8.1. Four action formulations for the deterministic pendulum swing-up and balance problem.

8.2.1 Problem Review and Safety and Performance Guarantees

The basic task is minimum-time control to the set $G^1_{pend} = \{(\theta, \dot{\theta}) : \|(\theta, \dot{\theta})\| \leq 0.01\}$. In Section 7.3 we proposed four action formulations, which are recapitulated in Table 8.1. The first three formulations rely on an alternate goal set, $G^2_{pend} = \{(\theta, \dot{\theta}) : \mathrm{ME}(\theta, \dot{\theta}) = 2\}$, with the understanding that when that goal set is reached, the pendulum is allowed to swing until its state enters G^1_{pend}. The time of the swing from G^2_{pend} to G^1_{pend} is incurred as a terminal cost. Formulation 1 has two actions based on the MEA control law. Both of these descend on CLF $L_{pend}(\theta, \dot{\theta}) = 2 - \mathrm{ME}(\theta, \dot{\theta})$. Formulation 2 has the two MEA-based actions as well as two constant-torque ($\pm u_{max}$) actions. These latter two do not always descend on L_{pend}. Formulation 3 has just the $\pm u_{max}$ actions. Formulation 4, which is the only action formulation to use the G^1_{pend} goal set, has the two constant-torque actions plus a third action based on an LQR design for a linear approximation to the pendulum dynamics. The third action makes hitting the small goal set, G^1_{pend}, feasible. We know of no CLF for formulations 3 and 4.

What can one say about reinforcement learning agents using these different action formulations—agents which may behave in a complex, stochastic, non-stationary manner that depends, potentially, on every time step of the previous experience controlling the pendulum?

Under the first three formulations, the state of the pendulum is guaranteed to stay in the set $\{ME \leq 2\}$ if it starts there, a simple safety guarantee. No agent behavior can impart a dangerous amount of energy to the pendulum, and the state stays bounded at all times. If the state of the pendulum ever reaches G^2_{pend}, then applying zero torque allows it to continue to G^1_{pend} and to asymptotically approach the upright, stationary state (the origin).

All actions in formulation 1 descend on L_{pend}, thus Theorem 8.1 applies. An agent using these actions is guaranteed to bring the state of the pendulum to G^2_{pend} within bounded time on every trial.

Because L_{pend} is a CLF for formulation 2, we know that G^2_{pend} is reachable from any state with ME < 2. However, there is no guarantee that an agent using those actions causes the state of the pendulum to reach G^2_{pend}. For example, the agent might simply choose the $+u_{max}$ action indefinitely, which puts the state of the pendulum on a periodic trajectory that does not intersect G^2_{pend}. As described more fully below, the learning approach we take distinguishes between learning trials and testing trials. During every step of a testing trial, the agent chooses whichever action it estimates to be best. It does not take any other actions nor does it change its estimates of the value of the actions. However, during learning trials the agent updates its action-value estimates and chooses an action at random with some non-zero probability on every time step. Both of these tend to break the agent out of unproductive, cyclic behaviors. Further, for an agent operating under formulation 2, the occasional random action selections mean that there is some positive probability of choosing a descending action on every time step. This means that the conditions of Theorem 8.2 are satisfied, ensuring that the state of the pendulum is brought to G^2_{pend} on every learning trial.

Under formulation 3, similar behavior should be expected. Although we have not proved it, we suspect that from any state with ME < 2, G^2_{pend} is reachable in a bounded number of steps using the $\pm u_{max}$ actions. Thus, we expect that during learning trials, the occasional random action selections are sufficient to guarantee eventual arrival to the goal set. During testing trials, one cannot guarantee that the state of the environment reaches the goal set.

The situation is different for agents operating under formulation 4. Because this action formulation relies on the G^1_{pend} goal set, the set of non-goal states is unbounded and a badly-behaving agent could easily drive the state of the pendulum arbitrarily far from the goal set. For example, if the agent causes the state of the pendulum to have ME > 2 and $\dot{\theta} > 0$, and chooses the $+u_{max}$ action from then on, then the pendulum's velocity and energy diverge to $+\infty$. There is no reason to think that occasional random action selections would save the pendulum from such a fate. So, although we expect G^1_{pend} to be reachable from any pendulum state, one cannot guarantee any kind of safety or goal-achievement.

8.2.2 Experiments

We performed experiments with reinforcement learning agents using the Sarsa(λ) algorithm ($\lambda = 0.9$) to update action value estimates which were stored in CMAC function approximators (see Appendices B and C). This combination has proven successful in a number of dynamical system control tasks, learning good policies quickly and stably (see, e.g., Sutton [87] and Santamaria et al. [75]). Each agent kept separate CMACs for each action, estimating the action's value as a function of the two state variables of the pendulum. Each CMAC covered the range of states $-\pi \leq \theta \leq \pi$ and $-4.01 \leq \dot{\theta} \leq 4.01$. Each CMAC had 10 layers, and each layer divided each dimension into 24 bins, for a total of 576 tiles per layer. Offsets were random. The step size for the k^{th} update of a tile's weight was $1/\sqrt{k}$. Weights were initialized to zero.

Under each action formulation we performed 30 independent learning runs, consisting of 20,000 trials each. Odd numbered trials were learning trials. During learning trials, the agents chose actions ϵ-greedily with $\epsilon = 0.1$. That is, on each time step, with probability ϵ, an action was chosen uniformly randomly, and otherwise the action with lowest estimated cost was chosen, ties being broken randomly. Even numbered trials were testing trials. In testing trials, agents always chose the action with lowest estimated cost, ties being broken randomly, and did not update their action value estimates. The purpose of testing trials was simply to assess the quality of the policies the agents learned over time. In all trials, the initial state of the pendulum was $(\theta, \dot{\theta}) = (\pi, 0)$, which is the downward-hanging, zero-velocity state. Choosing an action meant that the pendulum was controlled according to the corresponding control law for $\Delta = 1$ second. Trials were terminated after 999 time steps (seconds) if the state of the pendulum did not enter the goal set (G^1_{pend} or G^2_{pend}, depending on the action formulation) by that time. This gives plenty of time to reach the goal. The minimum time to goal is approximately 20 time steps, as we saw in Chapter 7. Under formulation 4, there is no bound on how large the pendulum's velocity can grow, which can be awkward for learning purposes. In these experiments we artificially limited the pendulum's velocity to the range $[-4, 4]$, holding $\dot{\theta}$ at the boundary if the natural dynamics would take $\dot{\theta}$ out of that range. We simulated the dynamics using 4^{th}-order Runge-Kutta integration with an integration step size of 0.01 seconds.

8.2.3 Results

Figure 8.1 and Table 8.2 summarize the results of the experiments. Figure 8.1 plots learning curves for the agents—the observed trial costs as a function of trial number—which are good for observing gross, qualitative differences resulting from the different experimental conditions. Remember that the trial cost is just the time taken to reach G^1_{pend} or 999 if the goal is not reached during the trial. Each plot

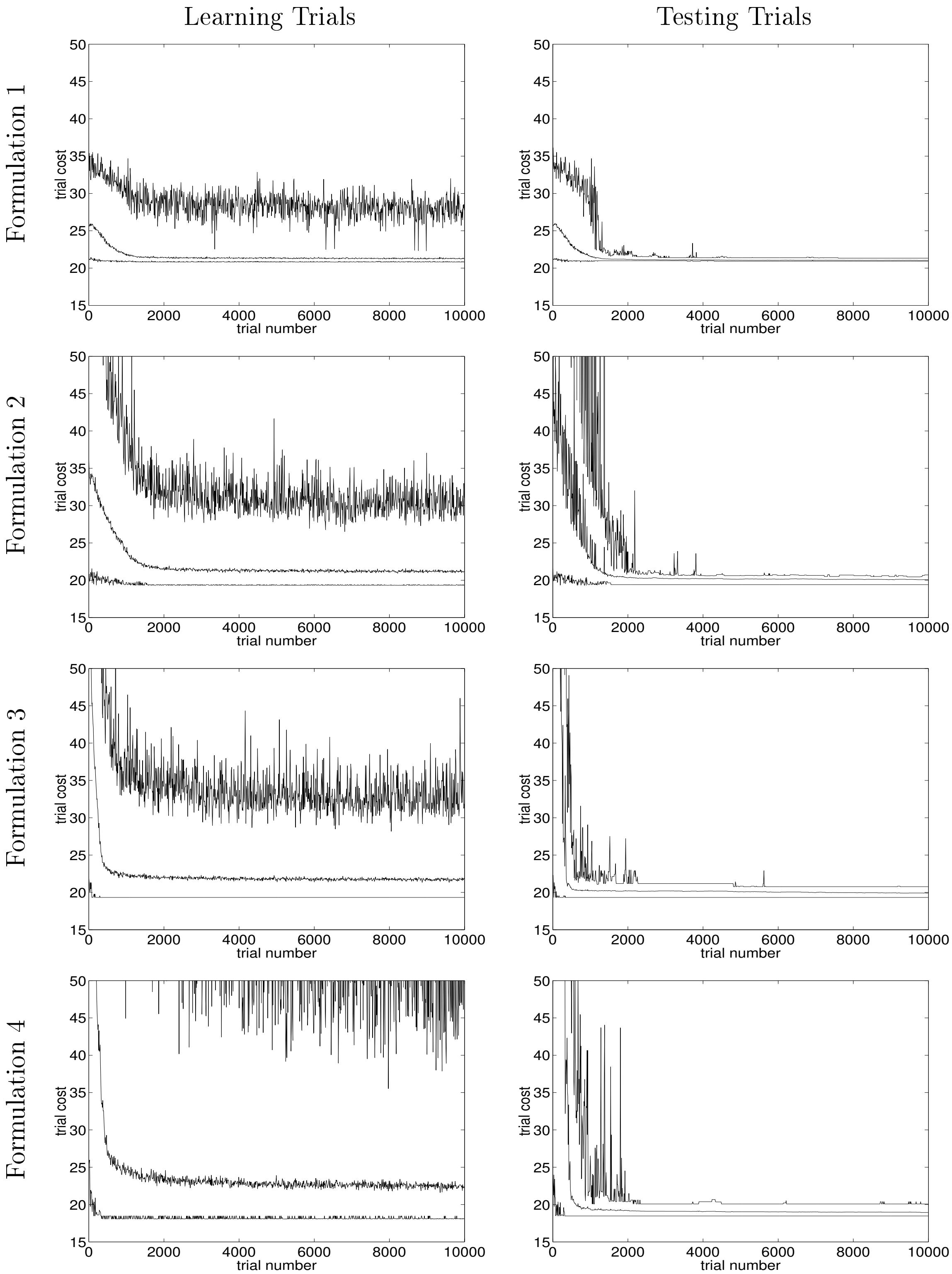

Figure 8.1. Trial costs as a function of trial number for learning experiments in the deterministic pendulum domain.

shows three curves. The middle curve is the mean trial cost, taken by averaging first across blocks of 10 successive trials and then averaging across runs. The upper curve represents the maximum trial costs observed across blocks of 10 trials and across runs. The lower curve represents the minimum trial costs.

All agents learn to control the pendulum well, as shown by the improvements in the learning and testing trial costs over time. The testing trial graphs indicate that by about trial 2000, all of the agents had settled (or nearly settled) on their final solutions, and there is very little variation in performance. The difference between the minimum and maximum trial cost curves during testing is due not to trial-by-trial variations in performance, but to the fact that different runs converged to slightly different policies. There is more variability in the learning trial costs, indicated by the spread between the minimum and maximum trial cost curves. This is mostly due to the ϵ-greedy behavior.

Not surprisingly, the agents using action formulation 1 showed the best initial performance, with initial performance decreasing as the reliance on Lyapunov domain knowledge decreases in action formulations 2, 3, and 4. The Lyapunov domain knowledge allows the agents using action formulation 1 to drive the state of the system to the goal set, even before there has been any learning and when actions are chosen essentially at random. Lyapunov domain knowledge also improves performance during learning trials, especially as measured by the maximum trial costs across runs. Under formulation 1, for example, both actions increase the energy. Even if an agent chose actions at random, the pendulum would swing upright. By contrast, in formulation 4, one or two bad actions can derail the state of the pendulum from the desired trajectory to the small, hard-to-hit goal set G^1_{pend}. The agent then has to recover from those bad actions, which can take considerable time.

Table 8.2 presents summary statistics. For different subsets of trials, minimum, mean, and maximum trial costs are reported. The mean trial cost is computed by

	Formulation 1		Formulation 2		Formulation 3		Formulation 4	
	learn	test	learn	test	learn	test	learn	test
First Trial								
min	**21.38**	**21.15**	21.83	23.41	22.05	35.74	76.85	58.53
mean	**25.50**	**26.98**	34.91	35.88	70.23	420.8	428.7	775.6
stderr	±0.79	±1.02	±3.25	±3.53	±13.0	±152	±104	±128
max	**33.00**	**32.54**	53.86	63.27	181.5	999	999	999
First 10 Trials								
min	21.12	**21.15**	**20.71**	21.54	21.96	22.19	25.37	24.05
mean	**25.81**	**25.75**	36.24	81.55	62.91	268.9	264.1	508.7
stderr	±0.23	±0.28	±0.65	±25.5	±1.69	±43.3	±31.7	±44.3
max	**34.51**	**33.37**	72.82	999	181.5	999	999	999
Last 100 trials								
min	20.80	20.92	19.33	19.41	19.33	19.33	**18.07**	**18.47**
mean	21.28	21.03	**21.16**	20.08	21.78	19.92	22.35	**18.94**
stderr	±0.04	±0.04	±0.07	±0.13	±0.12	±0.18	±0.20	±0.14
max	**28.53**	21.33	32.98	20.72	36.19	20.77	92.56	**20.08**
All trials								
min	20.80	20.80	19.33	19.33	19.33	19.33	**18.07**	**18.47**
mean	**21.57**	**21.33**	22.11	21.65	22.48	22.19	24.61	23.11
stderr	±0.03	±0.04	±0.03	±0.20	±0.06	±0.34	±0.12	±0.46
max	**35.51**	**36.13**	80.46	999	181.46	999	999	999
total time-outs	**0**	**0**	**0**	175	**0**	432	8	757

Table 8.2. Summary statistics for learning experiments in the deterministic pendulum domain.

averaging first across trials and then across runs. The "stderr" row gives the $\pm 2/\sqrt{30}$ standard deviations of the mean trial cost across runs. In the min, mean, and max rows, the best (lowest) figure for learning trials and for testing trials is highlighted in bold.

As mentioned above, the performance near the start of learning was generally best with action formulation 1. The difference between formulations 1 and 4 (and to a lesser extent, formulation 3) is striking. The mean cost of the first learning trial, for example, was 20 times better under formulation 1 than under formulation 4. For testing trials, there was a factor of 30 difference. These comparisons would favor formulation 1 even more if there were not an artificial limit of 999 time steps put on the trials. A slight exception to the rule is that the best learning trial out of the first ten came under formulation 2 rather than formulation 1.

Final performance was good for all action formulations. The MEA(u_{max}) and MEA($\frac{1}{2}u_{max}$) control laws alone can swing the pendulum upright, resulting in trajectories taking 24.19 and 37.27 seconds respectively to reach G^1_{pend}. Agents using any of the action formulations learned to do significantly better than that. For reasons explained in Chapter 7, best final performance is achieved under formulation 4. Intermediate performance results are obtained under formulations 2 and 3. Formulation 1 leads to the worst final performance. Ultimately, the least-constrained agents are able to swing the pendulum upright most quickly.

During the last 100 trials, the best trajectories under each action formulation matched the best trajectories found by state-space search. Except in the case of formulation 2, for which the optimal A* and DFBnB searches did not terminate, we know those trajectories to be optimal. The average trial cost was slightly higher. Even after 9,900 learning trials, a few runs continued to show small changes in what policy was estimated to be best. However, most of the variation in mean test trial time is due to the fact that different runs settled on slightly different policies. If learning continued long enough, all the runs might have converged to the same policy. However, there is no theoretical reason to think this would happen. The use of CMAC function approximators to estimate action values and the ϵ-greedy action selection with constant ϵ violate the assumptions of any known convergence proof.

The last row of the table reports the total number of learning and testing trials that timed out (i.e., the number of trials during which the state of the pendulum did not reach the goal before 999 time steps elapsed). The results here perfectly match the qualitative projections made in Section 8.2.1 based on Theorems 8.1 and 8.2. The state of the pendulum reached the goal in all formulation 1 trials as well as all learning trials under formulations 2 and 3. Even the longest formulation 1 trial took only 36.13 seconds—just twice the cost of the best trial under any action formulation. During a significant number of formulation 4 trials and formulation 2 and 3 testing trials, the state of the pendulum was not brought to the goal. As suggested by the theory at the beginning of this chapter, ensuring descent on a Lyapunov function on every time step, or at least some probability of descent, ensures that the state of the pendulum is brought to the goal set. The presence of Lyapunov-based actions or goal sets in the problem formulation is helpful, but does not guarantee good performance.

8.3 Stochastic Pendulum Demonstration

Next, we turn to the stochastic pendulum swing-up and balance problem that was introduced in Section 6.2. After reviewing the problem, we introduce three action formulations for controlling the pendulum, which differ from the action formulations we used for the deterministic swing-up and balance problem. We discuss safety and performance guarantees for the different action formulations and present reinforcement learning experiments.

Action Formulation	Actions
1	$+u_{max},\ 0,\ -u_{max}$
2	$+u_{max},\ +\frac{1}{2}u_{max},\ 0,\ -\frac{1}{2}u_{max},\ -u_{max}$
3	MEto2[$+u_{max}$], MEto2[0], MEto2[$-u_{max}$]

Table 8.3. Three action formulations for the stochastic pendulum swing-up and balance.

8.3.1 Problem Review, Action Formulations, and Safety and Performance Guarantees

Recall that the stochastic pendulum dynamics introduced in Section 6.2 include Gaussian noise driving the position variable. As such, asymptotically stabilizing the pendulum upright, or even maintaining the state of the pendulum in some set around the upright position, is impossible. The task is to keep the state of the pendulum near upright and stationary as much of the time as possible, where "near upright and stationary" is defined by the set $T_{up} = \{(\theta, \dot{\theta}) : |\theta| < 0.5,\ |\dot{\theta}| < 0.3\}$. To formalize the task as an optimal control problem, we suppose that unit cost is incurred per unit time when the state of the pendulum is not in T_{up} and zero cost is incurred when the state of the pendulum is in T_{up}. There are no goal states. We discount future costs with a discount rate of $\gamma = 0.95$, so that expected discounted returns are finite.

In all, we propose three action formulations for the problem, which are summarized in Table 8.3. The first two action formulations are based on simple, constant-torque control laws. Formulation 1 has three actions, corresponding to torques of $\pm u_{max}$ and 0. Formulation 2 has five actions, corresponding to torques of $\pm u_{max}$, $\pm\frac{1}{2}u_{max}$, and 0.

Formulation 3 is a Lyapunov-based design. Recall that in Section 6.2 we described a control law called MEto2, which brings the pendulum's energy toward 2. When energy is low, it uses the MEA strategy to increase it. When energy is high, MEto2 applies braking torque to reduce it. When the pendulum's energy is near 2, no torque is applied and the pendulum tends to swing upright, its state entering T_{up}. The intuition behind formulation 3 is that MEto2 is always used when the state of the pendulum is outside of T_{up}, in order to bring it back into T_{up}. The agent must learn how to keep the state of the pendulum in T_{up}. Formulation 3 has three actions, corresponding to the control laws:

$$\mathrm{MEto2}[+u_{max}](\theta,\dot{\theta}) = \begin{cases} +u_{max} & \text{if } (\theta,\dot{\theta}) \in T_{up} \\ \mathrm{MEto2}(\theta,\dot{\theta}) & \text{otherwise} \end{cases},$$

$$\mathrm{MEto2}[0](\theta,\dot{\theta}) = \begin{cases} 0 & \text{if } (\theta,\dot{\theta}) \in T_{up} \\ \mathrm{MEto2}(\theta,\dot{\theta}) & \text{otherwise} \end{cases},$$

$$\mathrm{MEto2}[-u_{max}](\theta,\dot{\theta}) = \begin{cases} -u_{max} & \text{if } (\theta,\dot{\theta}) \in T_{up} \\ \mathrm{MEto2}(\theta,\dot{\theta}) & \text{otherwise} \end{cases},$$

What can we predict or guarantee about agents using these action formulations? We have already assumed that the state says bounded ($(\theta,\dot{\theta}) \in [-\pi,\pi] \times [-4,4]$). No stronger safety guarantees can be provided. For any agent, it is possible to devise sequences of random disturbances to the position variable that cause the pendulum state to reach any value in $[-\pi,\pi] \times [-4,4]$. On the other hand, the same randomness ensures that for any agent the state of the pendulum enters T_{up} eventually. However, if an agent using action formulation 1 or 2 learns a bad policy, one may have to wait a long time for the lucky position disturbances that bring the state of the pendulum into T_{up}.

Agents operating under formulation 3 always control the pendulum according to MEto2 when its state is outside of T_{up}. Intuitively, then, the state of the pendulum is drawn towards T_{up} much more strongly. In Theorem 6.2, we showed that if the state of the pendulum starts outside of $T_{\epsilon_{ME}} = \{(\theta,\dot{\theta}) : |\mathrm{ME}(\theta,\dot{\theta}) - 2| \leq \epsilon_{ME}\}$, then there is some probability of descent on $L^2_{pend}(\theta,\dot{\theta}) = |\mathrm{ME}(\theta,\dot{\theta}) - 2|$. That is, L^2_{pend} is a CLF for formulation 3 agents with target set $T_{\epsilon_{ME}}$. Further, because the state of the pendulum is bounded, L^2_{pend} is bounded above and Theorem 8.2 implies that formulation 3 agents always cause the state of the pendulum to enter $T_{up} \cup T_{\epsilon_{ME}}$ eventually. If the state of the pendulum enters $T_{\epsilon_{ME}}$ before T_{up}, then there is some chance it will stay in $T_{\epsilon_{ME}}$ until it reaches T_{up}. Otherwise, the agent is assured of returning the state of the pendulum to $T_{up} \cup T_{\epsilon_{ME}}$ eventually, giving another chance at entering T_{up}. So, although under all action formulations the state of the pendulum is guaranteed to reach T_{up} eventually, the Lyapunov design provides a much stronger heuristic bias towards reaching T_{up} than formulations 1 or 2 do.

8.3.2 Experiments

We performed learning experiments in largely the same manner as we did for the deterministic pendulum domain. Agents used the Sarsa(λ) algorithm with $\lambda = 0.9$ to update action value estimates, which were stored using separate CMAC function approximators for each action. The CMACs were of the same design described in Section 8.2.2. For each action formulation we performed 30 independent runs of 2000 trials each, alternating learning trials and testing trials. All trials started with the initial state $(\theta, \dot{\theta}) = (\pi, 0)$, the downward-hanging, zero-velocity state. Since there are no goal states to signal the end of a trial, we simply ran all trials for 999 seconds (simulated time). In pilot experiments we found that allowing agents to make control decisions only once per second made keeping the state of the pendulum in T_{up} very difficult. On the other hand, if agents were allowed to make 10 control decisions per second, the learning problem was very hard. We settled on an intermediate action duration of $\Delta = 0.25$ seconds. This means that the 999 second trials consisted of 3996 action choices (time steps). Action selection was ϵ-greedy with $\epsilon = 0.1$ during learning trials and greedy during testing trials. We simulated the stochastic differential dynamics of the pendulum using an integration time step of 0.01 seconds by using 4^{th} order Runge Kutta integration to calculate the deterministic dynamics and then adding Gaussian noise to the pendulum's position with mean zero and standard deviation $0.1 \times \sqrt{0.01} = 0.01$. (This is a slightly non-standard approach, though perfectly adequate for our purposes. See, for example, Higham [31] for other methods.)

8.3.3 Results

Figure 8.2 and Table 8.4 summarize the results of the experiments. Figure 8.2 shows the minimum, mean, and maximum trial costs across runs for each action formulation and for learning and testing trials separately. These curves are not smoothed across trials. The trial cost is just the amount of time during which the state of the pendulum is not in T_{up}, which should be minimized.

Recall from Section 6.2 that using the MEto2 control law alone keeps the state of the pendulum in T_{up} approximately 49.2% of the time. This would correspond to an average trial cost of 507. Under all action formulations, the agents learn to do better than this, with mean trials costs between 200 and 300 during the final testing trials. Unlike what we saw in the deterministic pendulum domain, the costs of testing trials continue to vary significantly, even after the learning has mostly settled on a final policy. This is to be expected in a stochastic problem. Indeed, if one takes the distance between the min-cost and max-cost curves to indicate the amount of variability, it appears relatively independent of the action formulation and independent of whether the trial is a learning trial or a testing trial. This suggests that the stochasticity of the dynamics is the cause of the variation.

The most obvious difference between the action formulations is the comparatively good initial performance and rapid learning exhibited by the formulation 3 agents. Good initial performance was not unexpected, for the reasons discussed in Section 8.3.1. Rapid learning also makes sense, since these agents only needed to learn what actions to take when the state of the pendulum was near upright. The formulation

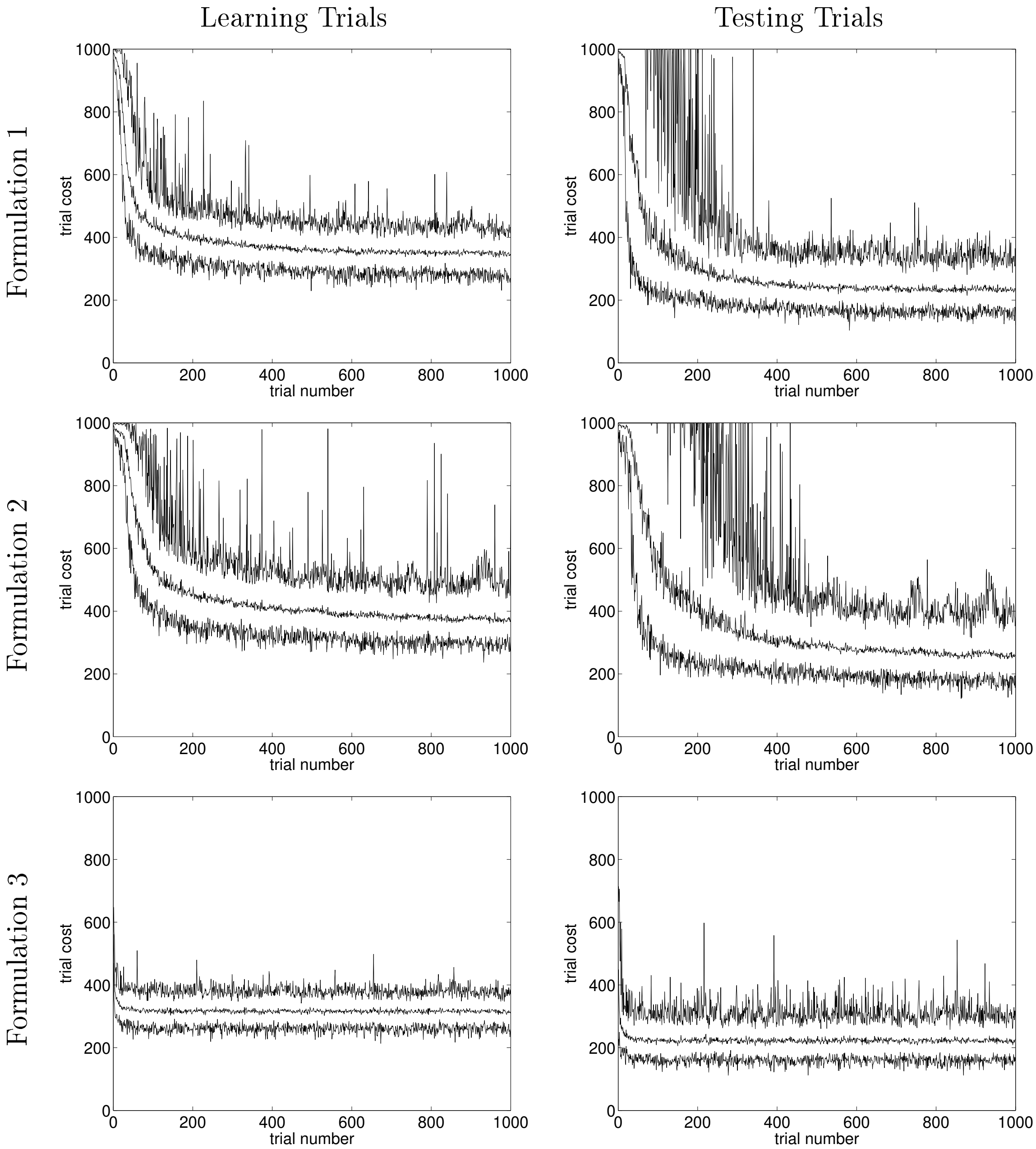

Figure 8.2. Trial costs as a function of trial number for learning experiments in the stochastic pendulum domain.

	Formulation 1		Formulation 2		Formulation 3	
	learn	test	learn	test	learn	test
First Trial						
min	966.6	916.1	979.8	949.4	**469.7**	**227.4**
mean	988.3	991.1	991.5	989.1	**561.8**	**448.9**
stderr	±2.7	±5.8	±1.9	±5.2	±16.8	±44.5
max	997	999	999	999	**647.9**	**714.1**
First 10 Trials						
min	861.7	885.7	925.2	891.6	**287.5**	**164.2**
mean	973.2	987.9	979.9	990.8	**397.6**	**305.7**
stderr	±2.1	±2.6	±1.1	±1.5	±7.0	±14.0
max	999	999	999	999	**647.9**	**714.1**
Last 100 trials						
min	231.15	**120.78**	237.17	125.26	**227.59**	129.12
mean	349.2	234.5	375.1	262.2	**315.4**	**222.1**
stderr	±8.4	±10.2	±13.6	±15.6	±5.6	±7.6
max	499.8	**421.38**	814.09	533.85	**416.84**	468.32
All trials						
min	229.6	**103.8**	237.2	121.6	**213.6**	111.9
mean	392.4	293.3	442.7	360.5	**317.6**	**223.9**
stderr	±8.4	±10.0	±10.5	±12.1	±5.6	±7.4
max	999	999	999	999	**647.9**	**714.1**
trials w/ no time in T_{up}	3	592	9	1042	**0**	**0**

Table 8.4. Summary statistics for learning experiments in the stochastic pendulum domain.

1 and formulation 2 agents needed to learn what actions to take in all parts of the state space.

Table 8.4 presents summary statistics. For different subsets of trials, minimum, mean, and maximum trials costs are reported. The mean cost is computed by averaging first across trials and then across runs. The "stderr" row indicates $\pm 2/\sqrt{30}$ standard deviations of the mean trial cost across runs.

Initially, agents using formulation 3 performed best by all measures. By the last 100 trials performance was comparable under all three action formulations. Though formulation 3 agents still generated most of the best learning and testing performances, a formulation 1 agent was responsible for the lowest-cost learning trial of the last 100. The final performance was quite close for the agents using formulation 1 and 3. The agents using formulation 2 did not produce results quite as good. The learning curves of Figure 8.2 indicate that these agents were still improving their policies at the end of the 1000 learning trials.

The last row of the table reports the number of trials in which the state of the pendulum was never in T_{up}. A surprisingly large number of formulation 1 and 2 testing trials included no time in T_{up}. We observed several of these trials, but there was no clear, intuitive explanation for their poor performance except that the agents had not yet learned what actions to take in a large enough part of the state space and hence could not get the state of the pendulum into T_{up}. In some trials, the state of the pendulum stayed near the bottom of its range most of the time. In others, the pendulum swung back and forth with high amplitude, its state repeatedly coming near T_{up} but not making it in.

Even in the worst formulation 3 trial, the state of the pendulum was in T_{up} more than one quarter of the time. Once again, we observe that strong, Lyapunov-based constraints on a learning controller can result in good initial performance and reasonable worst-case performance. The asymptotic performance of agents using the Lyapunov-based action formulation was no worse than that of the other agents. In this case, it appears that constraining the agents in this way did not restrict their ability to optimize for cost. Although a Lyapunov design is, in general, going to be suboptimal with respect to an arbitrary cost function, it does not have to be so. It appears that in this problem, controlling the pendulum according to MEto2 when its state is outside of T_{up} is an optimal or near optimal thing to do.

8.4 Robot Arm Demonstration

In this section, we apply reinforcement learning to the deterministic and stochastic robot arm control problems introduced in Section 6.3. Recall that the two problems differ only in their dynamics, with the stochastic version of the dynamics including Gaussian disturbances to the joint positions that do not appear in the deterministic version. In Section 7.4, we described two action formulations, and we used state-space search to find optimal trajectories for the deterministic problem. We begin by reviewing the problem and action formulations introduced above. We then discuss safety and performance guarantees for agents using these action formulations, and present the results of reinforcement learning experiments.

8.4.1 Problem Review and Safety and Performance Guarantees

The task is to bring the state of the arm to the goal set $G_{arm} = \{(\Theta, \dot{\Theta}) : \|(\Theta, \dot{\Theta})\| \leq 0.01\}$, while minimizing the trajectory cost: $\int_{t=0}^{t_G} \|\Theta(t)\|^2 + \|\tau(t) - \tau_0\|^2 dt$, where $\Theta(t)$ is the arm's position as a function of time, $\tau(t)$ is the torque applied at the joints as a function of time, t_G is the time at which the state of the arm enters G_{arm}, and τ_0 is the torque needed to hold the state of the arm at the origin. In Section 7.4 we introduced two sets of control laws for the arm, which we designed using feedback linearization and linear-quadratic regulation methods (see Table 8.5). Action formulation 1 includes 5 control laws. The first three of these bring the state of arm to the origin, in the center of G_{arm}, but at different rates. The last two control laws do not accelerate the first joint of the arm, but pull the other two links into a folded position, allowing the arm to swing more easily around its first joint. The first control law, FL_1, causes the state of the arm to descend on the function $L_{arm}(\Theta, \dot{\Theta}) = [\Theta' \dot{\Theta}'] P \begin{bmatrix} \Theta \\ \dot{\Theta} \end{bmatrix}$, where P is a symmetric positive definite matrix. Thus, L_{arm} is a CLF for action formulation 1. Action formulation 2 also has five actions, which are identical to the formulation 1 actions as long as they cause the state of

Action Formulation	Actions
1	FL_1, FL_2, FL_3, FL_4, FL_5
2	FL_1, $LD[FL_2]$, $LD[FL_3]$, $LD[FL_4]$, $LD[FL_5]$

Table 8.5. Two action formulations for robot arm control.

the arm to descend on L_{arm}. When a formulation 1 action does not cause the state of the arm to descend on L_{arm}, then the corresponding formulation 2 action controls the arm like the FL_1 control law instead. In formulation 2, all actions cause the state of the arm to descend on L_{arm}.

In Section 6.3.3 we showed that, under the stochastic dynamics, the state of the arm stays bounded under control by FL_1. The same argument applies to the case of deterministic dynamics, and to control by any form of switching among the controllers defined above. Thus, any agent behavior under either action formulation is safe in the sense of keeping the state of the arm bounded.

Because the state of the arm stays bounded, L_{arm} is bounded above for all reachable states. During learning trials under formulation 1, in which a random action is selected at each time step with probability $\epsilon > 0$, Theorem 8.2 guarantees that the state of the arm is eventually brought to G_{arm} (under deterministic or stochastic dynamics). This guarantee does not hold for testing trials.

For formulation 2 learning and testing trials, Theorem 8.1 ensures that the state of the arm is brought to G_{arm} in bounded time in the case of deterministic dynamics. Under the stochastic dynamics, Theorem 8.2 ensures that the state of the arm is brought to G_{arm} eventually.

8.4.2 Experiments

Recall that in Section 7.4 we reported heuristic search results that were averaged over a set of nine initial configurations of the form $\Theta = [x, y, -y]$ for $x \in \{-\pi, -\frac{2}{3}\pi, -\frac{1}{3}\pi\}$ and $y \in \{-\frac{1}{2}\pi, 0, \frac{1}{2}\pi\}$. We do the same here. Learning runs were organized into learning suites and testing suites. Each suite included one trial from each initial state. As in our other learning experiments, action selection was ϵ-greedy with $\epsilon = 0.1$ during learning trials. Action value estimates were updated using the Sarsa(λ) algorithm with $\lambda = 0.9$. Action value functions were represented using separate CMACs for each action. Each covered the range of states: $\Theta \in [-3.2, 3.2]^3$ and $\dot{\Theta} \in [-5, 5]^3$, with 20 layers and 5 divisions per dimension in each layer. Each CMAC thus had a total of 155,520 tiles. The step size for the k^{th} update of a tile was $1/\sqrt{k}$. During testing trials, action selection was greedy and action values were not updated.

We performed 30 independent learning runs for both action formulations and for the deterministic and stochastic versions of the dynamics. Runs consisted of 1000 learning suites and 1000 testing suites of nine trials each. Choosing an action meant that the arm was controlled by the corresponding control law for $\Delta = 1$ second. Trials were terminated after 250 time steps (seconds) if the arm had not reached G_{arm} by that time. The deterministic version of the dynamics was simulated using 4^{th}-order Runge Kutta integration with an integration time step of 0.1 seconds. The stochastic dynamics were simulated by computing the deterministic dynamics using 4^{th}-order Runge Kutta integration with an integration time step of 0.1 seconds and then adding Gaussian noise to the joint positions with zero mean and standard deviation $0.2 \times \sqrt{0.1}$.

8.4.3 Results

Figures 8.3 and 8.4 and Tables 8.6 and 8.7 present the results of the experiments. Figure 8.3 shows learning curves for the experiments with the deterministic version of the arm dynamics. The minimum, mean, and maximum suite costs across runs are plotted, where the cost of a suite is just the average cost of the nine trials in the suite. Learning was rapid, with performance improving to near asymptotic levels after 50 or 100 learning suites. On average, agents using action formulation 2 per-

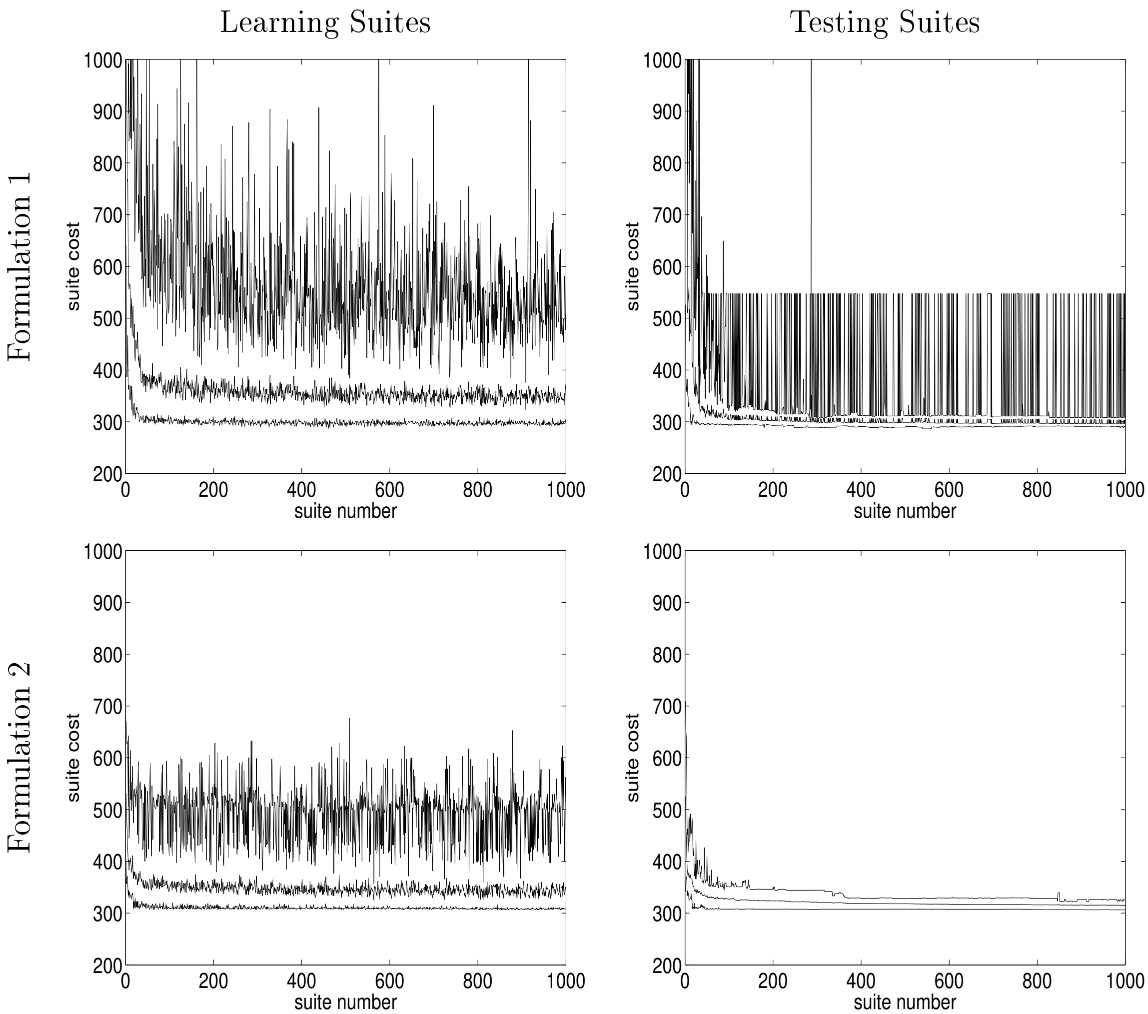

Figure 8.3. Suite costs as a function of suite number for learning experiments in the deterministic robot arm domain.

formed better initially than agents using formulation 1. Formulation 1 agents also had better worst-case performance, as indicated by the maximum-suite-cost curve. The test performance of formulation 2 agents qualitatively matched expectations, rapidly improving to near its final level, and exhibiting little variance. Something unexpected showed up in the testing suites for formulation 1 agents. One of the runs, run 26, failed to converge to a good policy. Instead, the run flip-flopped between two policies, one with testing suite cost 308.3 and one testing suite cost 547.6. The agent switched between these two policies every fifth suite or so. The most notable feature of the testing suite cost graph is the jagged maximum-suite-cost curve, which oscillates between 308 and 547 as a result of run 26. That run also affected the mean performance curve, although the affect was smaller since run 26 accounts for only one thirtieth part of the mean.

Table 8.6 presents summary statistics for the experiments with the deterministic version of the dynamics. For different sets of suites, the minimum, mean, and maximum suite costs are reported. The mean suite cost is computed by averaging first across suites and then across runs. The "stderr" row gives $\pm 2/\sqrt{30}$ standard deviations of the mean suite cost. As we observed from the graphs, initial performance was better under formulation 2 than under formulation 1. In the last 100 suites, and over all suites, the best learning and testing performances came under formulation 1. We know from Section 7.4 that optimal solutions are better under formulation 1 than under formulation 2, so this is not surprising. Somewhat surprisingly, no learning or testing suites achieved the optimal suite cost, although the best suites were not far above optimal (283.1 and 303.7 for formulations 1 and 2 respectively). Once there has been sufficient learning, mean test suite costs are better under formulation 1 than under formulation 2. Mean learning trial performance is comparable, perhaps slightly favoring formulation 2. The maximum suite costs later in learning are significantly better under formulation 2. The formulation 1 statistics are hurt in part by

	Formulation 1		Formulation 2	
	Learning	Testing	Learning	Testing
First Suite				
min	641.2	526.322	**403.3**	**341.7**
mean	1,090.5	1,334.0	**531.7**	**525.5**
stderr	±69.9	±488.1	±29.7	±39.2
max	1,395.7	7,958.6	**671.8**	**708.9**
First 10 Suites				
min	342.7	329.0	**329.4**	**324.2**
mean	737.5	1,094.1	**437.0**	**410.6**
stderr	±17.9	±346.4	±4.9	±7.5
max	1,583.1	29,097.0	**671.8**	**708.9**
Last 100 Suites				
min	**291.5**	**288.7**	306.2	306.4
mean	349.3	**298.3**	**343.1**	315.3
stderr	±3.8	±5.3	±2.1	±2.0
max	1,007.2	547.6	**623.2**	**326.3**
All Suites				
min	**287.3**	**286.3**	306.2	306.4
mean	361.6	**314.4**	**348.1**	321.0
stderr	±3.3	±6.7	±1.6	±2.0
max	1,583.1	29,097.0	**677.2**	**708.9**
Trial Statistics				
Longest (sec)	250	250	**11.66**	**10.98**
Costliest	7,866.6	245,254.2	**1,966.7**	**1,965.7**
Time-outs	3	2287	**0**	**0**

Table 8.6. Summary statistics for learning experiments in the deterministic robot arm domain.

the anomalous run 26. Learning performance is also hurt by the exploratory actions taken, which have more capacity to bring the state of the arm away from goal and to incur high costs than the exploratory actions under formulation 2.

The final section of the table reports statistics concerning individual trials. Some learning and testing trials under formulation 1 timed out at 250 seconds, whereas no trials under formulation 2 were even as long as 12 seconds. The lion's share of the time-outs under formulation 1, 2234 of the 2287 testing trial time-outs, were due to run 26. However, none of run 26's learning trials timed out nor did run 26 produce the costliest trials. The most expensive trial in run 26 cost 4,213.9. Of the 29 other formulation 1 runs, 28 included at least one trial more expensive than that, including six runs with testing trials costing over 245,000. The worst trial under formulation 2 was far better, costing only 1,966.7.

Figure 8.4 displays learning curves under the stochastic arm dynamics. The curves are qualitatively similar to the curves for the deterministic case. Learning is rapid under both action formulations; initial and worst-case performance is better under formulation 2 than under formulation 1. There are two main differences in the results for the stochastic arm. One is that there was no anomalous "run 26". The other is that, because of the stochasticity in the dynamics, a greater amount of noise persists in the performance curves.

Table 8.7 presents summary statistics for the experiments using the stochastic dynamics. The comparisons between formulations 1 and 2, indicated by the bold text, largely match what was seen for the deterministic dynamics. One of the most significant differences was the much smaller number of time-outs observed under the stochastic dynamics. This can be attributed to the fact that there was no run that failed to converge to a good policy.

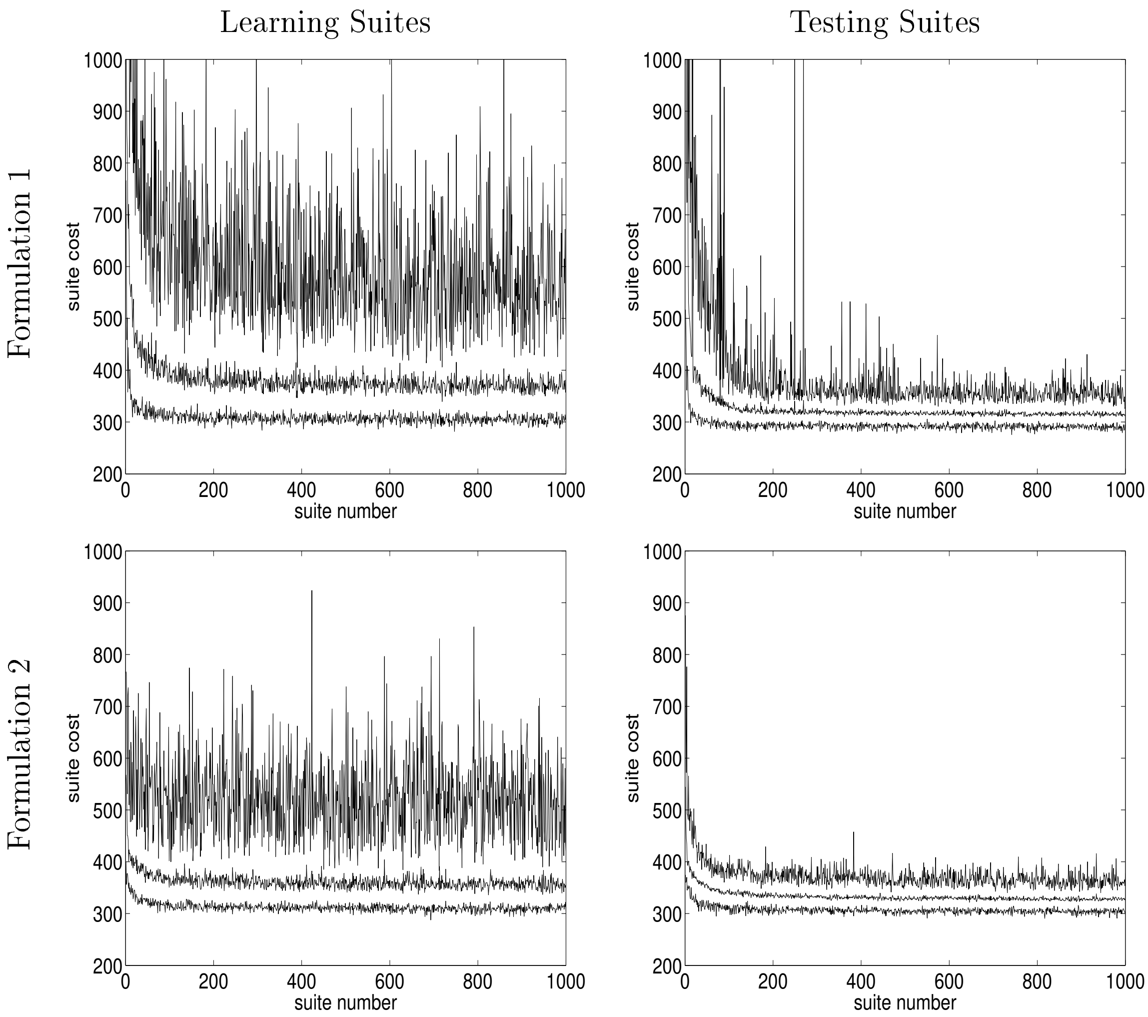

Figure 8.4. Suite costs as a function of suite number for learning experiments in the stochastic robot arm domain.

	Formulation 1		Formulation 2	
	Learning	Testing	Learning	Testing
First Suite				
min	766.3	516.7	**400.2**	**374.6**
mean	1,165.0	1,976.2	**567.9**	**544.5**
stderr	±78.8	±1,776.8	±38.1	±43.7
max	1,564.4	27,661.9	**767.1**	**875.6**
First 10 Suites				
min	357.8	**324.0**	**342.1**	345.1
mean	773.4	1,540.9	**461.6**	**428.3**
stderr	±21.1	±642.4	±8.9	±6.3
max	1,677.7	54,797.3	**767.1**	**875.6**
Last 100 Suites				
min	**287.9**	**279.1**	294.8	290.9
mean	370.0	**315.3**	**356.2**	328.0
stderr	±2.7	±1.6	±2.3	±2.0
max	833.2	430.6	**715.6**	**415.9**
All Suites				
min	**281.6**	**275.6**	287.6	290.5
mean	383.6	335.7	**362.0**	**333.7**
stderr	±1.2	±7.3	±1.8	±2.0
max	1,677.7	54,797.3	**923.9**	**875.6**
Trial Statistics				
Longest (sec)	250	250	**12.47**	**12.47**
Costliest	8,610.4	245,312.7	**5,690.2**	**3,649.4**
Time-outs	1	16	**0**	**0**

Table 8.7. Summary statistics for learning experiments in the stochastic robot arm domain.

8.5 Discussion

In this chapter we demonstrated that Lyapunov domain knowledge and other control-theoretic techniques can be used to design action formulations for optimal control problems, providing basic safety and performance guarantees. By formulating an agent's actions so that descent on a Lyapunov function is guaranteed, or probabilistically guaranteed, one can: (1) ensure that the state of the environment reaches a desirable region of state space, such as the goal set; in some cases one can bound on the time this takes; (2) ensure that the state of the environment stays bounded at all times, one of the most basic forms of safety; and (3) ensure desirable asymptotic system behavior.

Importantly, these guarantees are derived primarily from properties of the action formulations, making few or no assumptions about how an agent chooses among those actions. This allows us to provide guarantees for approaches to approximate optimal control that otherwise offer no such guarantees. In particular, we have demonstrated that reinforcement learning agents using established, state-of-the-art algorithms and representations for learning to control dynamical systems benefit greatly from Lyapunov-based designs. In practice, the empirical benefits include: better initial performance during learning, better worst-case performance, shorter trial times, and more rapid learning.

Strict Lyapunov-descent constraints provide good theoretical guarantees, but they are subject to at least one potential drawback. Restricting an agent to choose actions that cause the state of the environment to descend on a given Lyapunov function may limit its ability to minimize cost. In our example problems, the least-constrained agents performed up to 10 or 15% better than the most-constrained agents. In the deterministic pendulum domain, agents using the relatively naive and unconstrained formulation 4 found trajectories taking 18.08 seconds to reach the goal, whereas agents using the Lyapunov-based formulation 1 did no better than 20.80 seconds. In the stochastic pendulum domain, agents were able to do equally well under all action formulations, ranging from a heavily-constrained Lyapunov-based design to a simple design with five constant-torque actions. In this problem, the Lyapunov design did not impede the agent's ability to minimize costs.

In the robot arm domain, both of the action formulations we considered relied heavily on control-theoretic techniques. However, one ensured descent on a Lyapunov function and the other did not. Again, we observed that (most) of the less-constrained agents eventually learned to perform better than their more-constrained counterparts. In Chapter 10 we discuss whether or not it is really necessary to accept the possibility of inferior asymptotic performance in order to achieve theoretical guarantees on safety and performance, and we discuss how trade-offs between these sometimes-competing goals might be achieved.

CHAPTER 9

LEARNING LYAPUNOV FUNCTIONS

In the previous chapters we assumed that Lyapunov domain knowledge was available, and we demonstrated how such knowledge can be integrated into standard AI approaches to sequential control to achieve both safety and reliability as well as optimal or approximately optimal control. Lyapunov functions are a useful form of domain knowledge, but such knowledge is not always available. Even assuming that the environment's dynamics are known can be unrealistic.

Could an agent itself identify a Lyapunov function based on experience with an environment? Consider the problem of determining whether a deterministic Markov decision process descends on a given function L from all $s \notin T$ for some $T \subset S$. Suppose that the agent can observe trajectories of the process. Supposed there are an infinite number of states outside of T. No matter how many times the agent observes the state of the process descending on L outside of T, the agent can never be sure that L is a Lyapunov function. There is always the possibility that from some as-yet-unobserved $s \notin T$, the next state of the environment is neither in T nor lower on L. However, a single observation of the process not descending on L is sufficient to rule out L as a potential Lyapunov function.

In this chapter, we present an algorithm that takes as input a set of candidate Lyapunov functions and uses them to dynamically restrict the action choices of an agent such that, with bounded total "loss", either: (1) the agent causes the state of its environment to reach the goal set on an infinite number of trials, or (2) all candidate Lyapunov functions are ruled out by observing non-descending transitions.

9.1 The Algorithm

The algorithm, which we call "LL" for Learning Lyapunov functions, is applicable to deterministic minimum-cost-to-goal MDPs. LL is presented in Figure 9.1. The algorithm takes as input $\mathcal{L}$, a finite set of candidate Lyapunov functions, and Π, a finite set of deterministic policies. A candidate Lyapunov function is any mapping from S to $\mathbb{R}$ that is positive outside of the goal set, G.

Candidate Lyapunov functions may be hypothesized based on a partial understanding of the environment. For example, in an experiment below we assume that the agent will face a pendulum swing-up task, but that we do not know the length of the pendulum a priori. We provide the agent with a set of candidate Lyapunov functions that covers a range of possible pendulum lengths that we anticipate. Candidate Lyapunov functions might also be generated based on optimistic assumptions. For example, an agent navigating in an unknown spatial environment might assume that decreasing the Euclidean distance between its current location and its desired location will eventually bring it to its desired location.

Intuitively, the idea of the LL algorithm is that it identifies which policies in Π cause the state of the environment to descend on which candidate Lyapunov functions in $\mathcal{L}$, if any. In other words, the algorithm learns which policies are safe, and restricts action choice to that set of policies. If there are multiple safe policies, the algorithm does not identify which is best from a cost point of view. That can be done by combining LL with other (e.g., reinforcement learning) approaches. If there are no safe policies, then the algorithm proves this is the case by eliminating all candidate Lyapunov function-policy pairs.

In order to do this, actions must be chosen carefully. An agent that chooses actions in an arbitrary fashion, say randomly, need not ever bring the environment to G nor rule out all candidate Lyapunov functions. At the same time, one must recognize the cost-minimizing goals of the agent, and not focus exclusively on the question of which

Inputs:

$\mathcal{L}$, a finite set of candidate Lyapunov functions,

Π, be a finite set of deterministic policies,

δ, the desired descent constant,

and access to an deterministic environment in some initial state.

Cand $\leftarrow \mathcal{L} \times \Pi$
ActiveL $\leftarrow \mathcal{L}$.
repeat
  Let s be the current state of the environment.
  Choose $a \in \text{Acts}(\text{Cand}, \text{ActiveL}, s)$.
  Apply a to the environment and observe the next state s'.
  if $s' \in G$ **then**
    Start a new trial.
    Set the environment to a new initial state.
    ActiveL $\leftarrow$ LiveL(Cand)
  else
    Update Cand.
    for all $(L, \pi) \in$ Cand such that $\pi(s) = a$ and $L(s) - L(s') < \delta$ **do**
      Remove (L, π) from Cand.
    end for
    Update ActiveL.
    for all $L \in$ ActiveL such that $L(s) - L(s') < \delta$ **do**
      Remove L from ActiveL.
    end for
    if ActiveL $= \emptyset$ **then**
      ActiveL $\leftarrow$ LiveL(Cand)
    end if
  end if
until Cand $= \emptyset$
return "None are Lyapunov functions"

Where:
$\text{Acts}(\text{Cand}, \text{ActiveL}, s) = \{a : \text{for some } (L, \pi) \in \text{Cand}, L \in \text{ActiveL and } \pi(s) = a\}$,
and $\text{LiveL}(\text{Cand}) = \{L : \text{for some } \pi,\ (L, \pi) \in \text{Cand}\}$.

Figure 9.1. The LL Algorithm.

candidate Lyapunov functions are valid and which policies are safe. The LL algorithm does not specify precisely which actions the agent should take. It circumscribes the agent's choice just enough to achieve the loss bound mentioned above, while allowing space for cost minimization.

In addition to the $\mathcal{L}$ and Π inputs described above, LL takes a real number, δ, which is the minimum amount the state of the environment should descend on a candidate Lyapunov function. The algorithm keeps track of the set Cand, which is the set of all Lyapunov function-policy pairs that have not been ruled out by any transitions observed so far. A pair (L, π) is ruled out when the state of the environment is s, the agent takes action $a = \pi(s)$, the next state of the environment is $s' \notin G$ and $L(s) - L(s') < \delta$. A Lyapunov function, L, is ruled out when the pairs (L, π) have been ruled out for all $\pi \in \Pi$.

LL also keeps a set of Lyapunov functions, called ActiveL, which contains the set of Lyapunov functions on which the state of the environment has been descending since the start of the trial or since the last time ActiveL became empty, whichever is more recent. The restriction that the agent choose actions in $\text{Acts}(\text{Cand}, \text{ActiveL}, s)$ is the crux of the algorithm. As long as the agent chooses any action in this set, the performance guarantees below hold. In Section 9.2 we demonstrate this approach on a pendulum swing-up problem, using Sarsa(λ) to learn which actions are best to take within the restrictions imposed by LL. The following lemma establishes a key property of the LL algorithm.

Lemma 9.1 *Let* $\max_{L \in \mathcal{L}} \sup_{s \notin G} L(s) = U \in \mathbb{R}$*, and consider any time step on which ActiveL is reset from Cand (i.e., the start of a trial or when ActiveL becomes empty). Within* $\lceil U/\delta \rceil$ *time steps either the state of the environment enters* G *or at least one* $(L, \pi) \in$ *Cand is ruled out by observing a non-descending transition.*

Proof: Suppose the state of the environment does not enter G in $K = \lceil U/\delta \rceil$ time steps. Then ActiveL must become empty again within K time steps. A Lyapunov function stays in ActiveL only if the state of the environment descends on it at every step, and K descents of at least δ are not possible on any of the candidate Lyapunov functions without entering G. At the start of the time step on which ActiveL becomes empty, it contains at least one L such that $(L, \pi) \in$ Cand and $\pi(s)$ is the action chosen. Since ActiveL becomes empty, it must be that $L(s) - L(s') < \delta$, thus the pair (L, π) is removed from Cand. $\square$

For trajectory τ, let $t_G(\tau)$ be the time at which the state of the environment enters G, if any, and $+\infty$ if the state does not enter G. Let the loss of the trajectory be $l(\tau) = \max(t_G(\tau) - \lceil U/\delta \rceil, 0)$. The loss is the time taken in excess of $\lceil U/\delta \rceil$, which is the time-to-goal we could guarantee if we knew a Lyapunov function with upper bound U and descent constant δ. Let $\{\tau_i\}$ be a potentially-infinite set of trajectories (trials) generated by the LL algorithm.

Theorem 9.2 *Let* $\max_{L \in \mathcal{L}} \sup_{s \notin G} L(s) = U \in \mathbb{R}$. *Either LL generates an infinite number of trials that reach G or the algorithm returns "None are Lyapunov functions", and in either case* $\sum_{\tau \in \{\tau_i\}} l(\tau) \leq \lceil U/\delta \rceil (|\mathcal{L} \times \Pi| - 1) + 1$. *If LL generates an infinite number of trials that reach G, then at most* $|\mathcal{L} \times \Pi| - 1$ *of them last longer than* $\lceil U/\delta \rceil$ *time steps.* $\square$

Proof: For the first claim of the theorem, there are three possibilities—the two mentioned, and the possibility that the algorithm generates a finite number of trials, the last one of which has infinite length. However, Lemma 9.1 implies that a trajectory of infinite length is impossible. Each $\lceil U/\delta \rceil$ time steps, the agent would rule out at least one element of Cand and, in finite time, exit the main loop and return "None are Lyapunov functions". The total loss bound follows by the same reasoning. The first Lyapunov function-policy pair is ruled out while incurring no more than 1 loss, because a loss of 1 comes from a trajectory of length $\lceil U/\delta \rceil + 1$, which is assured of ruling out one pair. Subsequent pairs are ruled out while incurring between 0 and $\lceil U/\delta \rceil$ loss, depending on whether they are ruled out in the same trial or on different trials. The final claim holds, again, because any trial lasting longer than $\lceil U/\delta \rceil$ time steps rules out at least one Lyapunov function-policy pair, and if LL generates an infinite sequence of trials in which the state of the environment reaches G, at most $|\mathcal{L} \times \Pi| - 1$ pairs can be ruled out. $\square$

So, an agent using the LL algorithm to constrain action choices can either use the set of hypothesized Lyapunov functions to ensure reliable performance on all but a finite number of trials, or can inform its designer that none of the candidate Lyapunov functions are valid while incurring bounded loss. The loss bounds we establish may be quite loose, but it is significant to show that any loss bound at all is achievable. In practice, one might expect that Lyapunov function-policy pairs are ruled out far more frequently than the algorithm ensures, leading to a rapid focusing of attention on good candidate Lyapunov functions and policies, and hence ensuring reasonable agent behavior.

If $\mathcal{L}$ or Π are large, keeping track of the set Cand and computing the Acts and LiveL functions can be onerous. We close this section by presenting a specialization of the LL algorithm which is useful when the policies in Π are constructed in a particular way (described below). Suppose that A is a set of actions and B maps each $s \notin G$ to an element of $\{1, 2, \ldots, M\}$. In other words, B partitions the non-goal states into a finite number of bins. If Π is the set of all policies that can be described as associating an action in A to each state-space bin, then LL can be implemented efficiently, keeping track of Cand implicitly rather than explicitly. We call this version LL-E, for "efficient". The algorithm is presented in Figure 9.2. It maintains an array of truth values, called Okay, indexed by candidate Lyapunov function L, bin b, and action a, indicating whether or not *all* observed environment transitions from any state in bin b, under action a, have descended on L. Using the array, it is easy to compute which actions are available from any state s. LL-E keeps track of the LiveL function explicitly, since recovering that from the Okay array would be time-

Inputs: $\mathcal{L}$, a finite set of candidate Lyapunov functions, B, mapping S to $\{1, 2, \ldots, M\}$, A, a set of actions, δ, the desired descent constant, and access to an deterministic environment in some initial state.

---

for all $L \in \mathcal{L}$, $b \in \{1, \ldots, M\}$, and $a \in A$ **do**
  Okay(L,b,a)=**true**
end for
LiveL $\leftarrow \mathcal{L}$
ActiveL $\leftarrow$ LiveL
repeat
  Let s be the current state of the environment.
  $b \leftarrow B(s)$
  Choose $a \in A$ such that Okay$(L, b, a) =$ **true** for some $L \in$ ActiveL.
  Apply a to the environment and observe the next state s'.
  if $s' \in G$ **then**
    Start a new trial.
    Set the state of the environment to a new initial value.
    ActiveL $\leftarrow$ LiveL
  else
    Update LiveL and Okay.
    for all $L \in$ LiveL **do**
      if $L(s) - L(s') < \delta$ **then**
        Okay$(L, b, a) \leftarrow$ **false**
      end if
      if Okay$(L, b, a') =$ **false** for all $a' \in A$ **then**
        Remove L from LiveL.
      end if
    end for
    Update ActiveL.
    for all $L \in$ ActiveL such that $L(s) - L(s') < \delta$ **do**
      Remove L from ActiveL.
    end for
    if ActiveL $= \emptyset$ **then**
      ActiveL $\leftarrow$ LiveL.
    end if
  end if
until LiveL $= \emptyset$
return "None are Lyapunov functions"

Figure 9.2. The LL-E Algorithm.

consuming. Since LL-E is just a special case of LL, it retains all of LL's theoretical benefits.

9.2 Pendulum Demonstration

We demonstrate the LL approach on a pendulum swing-up problem, in which we assume the length of the pendulum is initially unknown. For a pendulum of length l, the acceleration equation is:

$$\ddot{\theta} = \frac{sin\theta}{l} + \frac{u}{l^2}$$

The task we study is minimum-time control to the set $G^3_{pend} = \{(\theta, \dot{\theta}) : |\theta| < 0.1\}$. This corresponds to a set of angular positions within approximately 6 degrees of upright. Note that the goal set does not restrict the velocity of the pendulum. This is a swing-up task, not a swing-up and balance task.

9.2.1 Lyapunov Functions and Controllers

The mechanical energy of a pendulum with length l is:

$$\mathrm{ME}_l(\theta, \dot{\theta}) = l(1 + \cos\theta) + \frac{1}{2} l^2 \theta^2 \ .$$

One strategy for swinging up the pendulum is to increase the pendulum's energy continuously until its state enters G^3_{pend}. One can define a generalization of the MEA control law that works for pendula of any length. However, we found in pilot experiments that the MEA controller, designed for the case of $l = 1$, is adequate for the task over a wider range of pendulum lengths. MEA is not optimal, as is shown in the experiments below. Reinforcement learning agents are able to discover better policies that switch between different control laws. Indeed, for any agent, some form of on-line learning is important because the optimal strategy for controlling the pendulum depends on its length, and that is unknown until the agent begins

Action Formulation	Actions	Restrictions	Relevant Theorems
1	$+u_{max}$, $-u_{max}$	none	none
2	$+u_{max}$, $-u_{max}$, MEA	none	none
3	$+u_{max}$, $-u_{max}$, MEA	LL-E	9.2

Figure 9.3. Action formulations for controlling a pendulum of unknown length.

observing the pendulum's response to control inputs. The problem, then, is how to ensure robust, safe learning when the system dynamics are not completely known, and thus a Lyapunov function is unavailable.

We propose three action formulations for learning to control the pendulum. They are summarized in Figure 9.3. Formulation 1 has two actions, corresponding to constant $\pm u_{max}$ torque control laws. Formulation 2 adds a third action, corresponding to MEA. For formulations 1 and 2, agents are allowed to choose unrestrictedly from the actions listed. Formulation 3 has the same actions as formulation 2, but the agent's action choices are restricted using the LL-E algorithm. The details of how we use LL-E are specified in the next section.

9.2.2 Experiments

For each action formulation and each pendulum length, $l \in \{1.0, 1.2, 1.4, \ldots, 3.0\}$, we ran 30 independent reinforcement learning runs. Agents used the Sarsa(λ) algorithm with $\lambda = 0.9$ to update action value estimates stored in CMAC function approximators. The CMACs were of the same design as described in the pendulum learning experiments in the previous chapter (10 layers, 24 bins per dimension). Each run alternated between learning and testing trials, with a total 2000 trials of each type in each run. In each trial, the initial state of the pendulum was $(\theta_0, \dot{\theta}_0) = (\pi, 0)$. During learning trials, action selection was ϵ-greedy with ϵ=0.1. For formulation 3 agents, that meant ϵ-greedy over the set of actions allowed by LL-E, not over all actions. During testing trials, action selection was greedy. For agents using formulation

3, none of the Okay, LiveL, and ActiveL were updated during a testing trial, as this would constitute a form of learning. The allowed actions where those for which Okay was true for some $L \in$ LiveL. Trials were terminated if the state of the pendulum did not reach G^3_{pend} within 999 time steps (seconds).

For LL-E we took $\mathcal{L}$ to be the set of functions: $-\mathrm{ME}_l(\cdot,\cdot)$, for $l \in \{1.0, 1.25, 1.5, \ldots, 3.0\}$. In other words, we hypothesized that the agent should make the state of the pendulum ascend on the mechanical energy function of a pendulum with length $l \in \{1.0, 1.25, \ldots 3.0\}$. We intentionally choose a set of candidate Lyapunov functions based on pendulum lengths different from the set of lengths used in the experiments. Note that, strictly speaking, these candidate Lyapunov functions do not meet the criteria of Theorem 9.2, which requires that candidates be positive outside of G. However, since these functions can be bounded above (by zero) and below for all reachable states, the same theoretical guarantees hold. Viewed another way, we could simply add a large positive constant to each of the functions, and the conditions of Theorem 9.2 would be satisfied. Since LL-E only looks at relative values of states, adding a constant does not change the behavior of the algorithm.

For B we divided the state space region $[-4, 4]^2$ into 6,400 bins by dividing each dimension into 80 equal-sized intervals. We took δ to be 0.00001.

9.2.3 Results

Because there are so many different pendulum lengths and action formulations to consider, we present most of the results graphically. We begin by describing the final performance learned by the agents. Figure 9.4 plots performance during the last 100 testing trials. For each pendulum length, a cluster of three error bars is shown, representing the formulation 1, formulation 2, and formulation 3 runs in left-to-right order. The dark, middle mark on each error bar represents the mean trial cost (duration), computed by averaging first across trials and then across runs. The

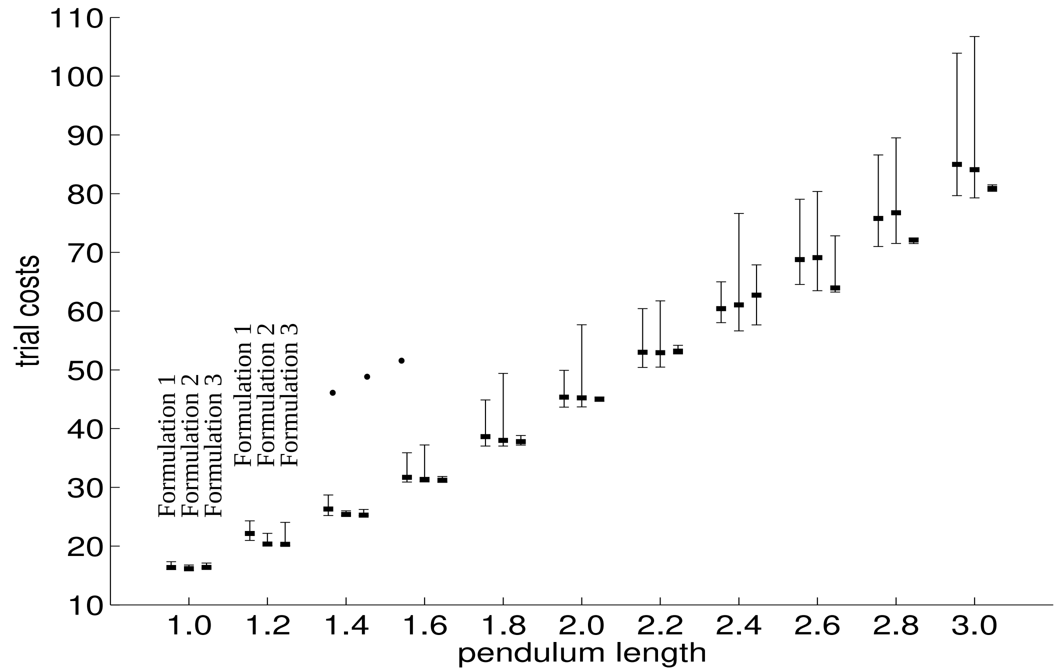

Figure 9.4. Performance during the last 100 testing trials.

upper and lower extents of the error bars indicate the minimum and maximum trial costs across the last 100 test trials and across runs.

It is clear that the dominant factor in solution quality is the length of the pendulum, and not the solution method used. The longer the pendulum gets, the more slowly it accelerates for a given torque input. Because $u_{max} = 0.2222$ is the same for all runs, swing-up times are considerably longer for the longer pendula. At most lengths, the final mean and minimum solution costs are comparable across the three action formulations. On average, the formulation 3 agents performed marginally better than the other agents at the longest lengths ($l \geq 2.6$). The final performance of the formulation 3 agents also tended to be less variable than that of the other agents.

Figure 9.5 shows the performance during the last 100 learning trials. There was more variability in performance at all pendulum lengths, due mostly to the random exploratory actions taken by the agents. There is also a clear pattern at each length in which mean and maximum trial costs were highest for formulation 1 agents, better

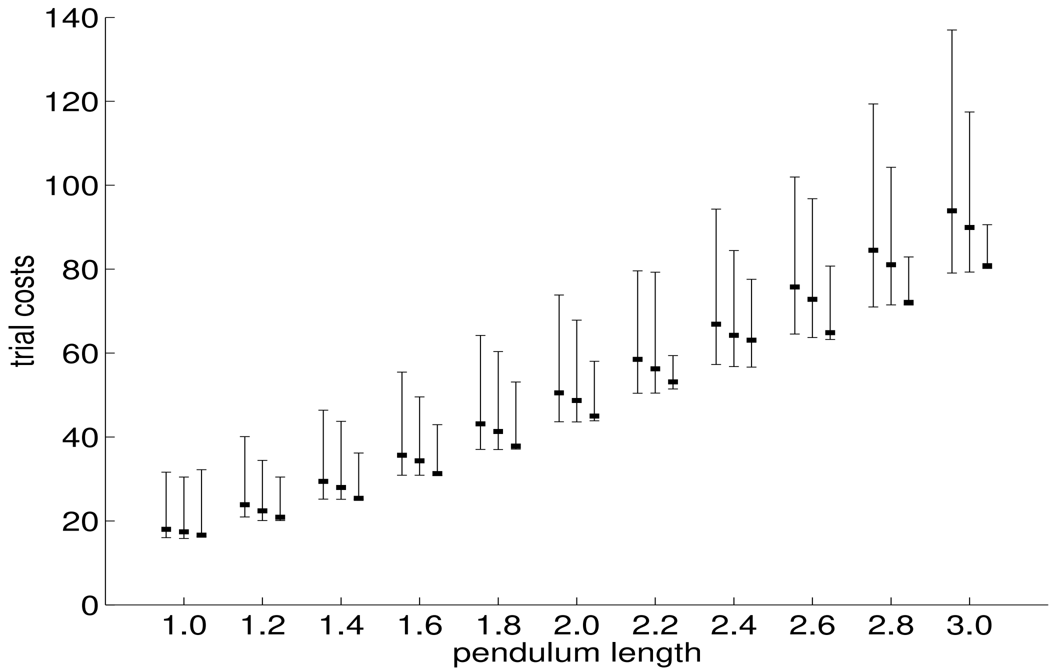

Figure 9.5. Performance during the last 100 learning trials.

for formulation 2 agents, and best for formulation 3 agents. Again, this is due in significant part to the random action selection. Because formulation 3 agents had ruled out many non-descending actions by this point in learning, random actions from the allowed set were likely to be descending. In formulation 2, the presence of the MEA action biases agents towards increasing energy, so, again, randomly selected actions were less likely to be harmful.

Table 9.1 presents the cost of the trajectory produced by MEA for each pendulum length, along with the cost of the best test trials that occurred under each action formulation. Although MEA does bring the state of the pendulum to G^3_{pend} for all the lengths we tested, it does not do so in minimum time. Learning agents were able to find superior solutions at all lengths. In some cases, the improvements were quite small. In a few, particularly the cases $l = 1.0$ and $l = 2.4$, the learned solutions were significantly better (18.5% and 10.7% shorter, respectively). Most of the best solutions came from formulation 2 agents. This is not surprising, since these agents

l	MEA	Minimum Test Trial Costs Formulation 1	Formulation 2	Formulation 3
1.0	19.39	16.03	**15.80**	16.26
1.2	20.22	20.96	**20.08**	**20.08**
1.4	25.37	25.19	**25.16**	**25.16**
1.6	31.40	30.89	30.85	**30.84**
1.8	37.96	37.04	**36.96**	37.00
2.0	45.43	43.65	**43.53**	44.31
2.2	53.78	50.42	**50.38**	51.98
2.4	63.42	57.27	**56.65**	56.69
2.6	63.61	64.43	63.47	**63.15**
2.8	71.63	**71.01**	71.12	71.46
3.0	80.58	79.66	**79.28**	80.12

Table 9.1. Best test trials under the three formulations, compared to MEA's performance.

were able to choose from every action that other agents could, and could do so unrestrictedly. The set of policies these agents could learn strictly includes the sets of policies that agents using formulations 1 and 3 could learn.

Figure 9.6 shows the performance during the first ten learning trials. Formulation 2 and 3 agents performed comparably, with formulation 3 agents doing slightly better for higher l. It is not surprising that the effect of using LL to restrict action choices was muted early in learning. There were many state space bins, and it took time to rule out actions. Formulation 1 agents performed significantly worse, with some trials timing out for $l \geq 2.6$. Figure 9.7 shows performance during the first ten testing trials. Without learning or exploration to break an agent out of a cycling policy, and without LL to prune actions, there were time-outs under all action formulations and at all pendulum lengths. In terms of mean performance, formulation 2 and 3 agents did better than formulation 1 agents. The advantage that formulation 3 agents had over formulation 2 agents is more clear than it is in learning trials.

Figure 9.8 shows the total number of time-outs across all learning and testing trials. For each pendulum length, the three bars represent action formulations 1, 2,

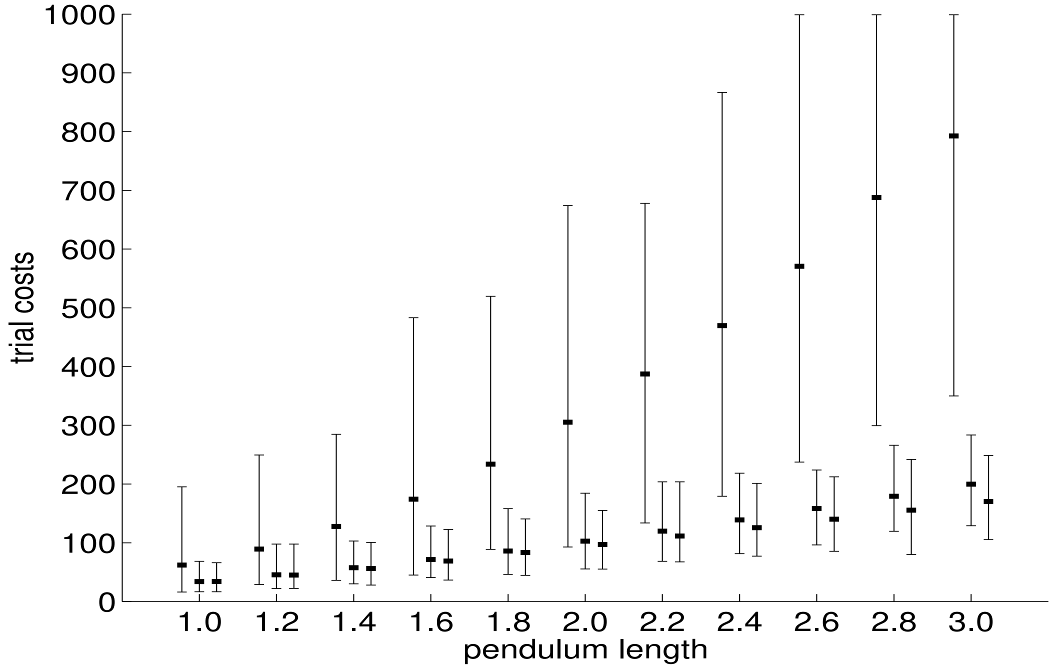

Figure 9.6. Performance during the first ten learning trials.

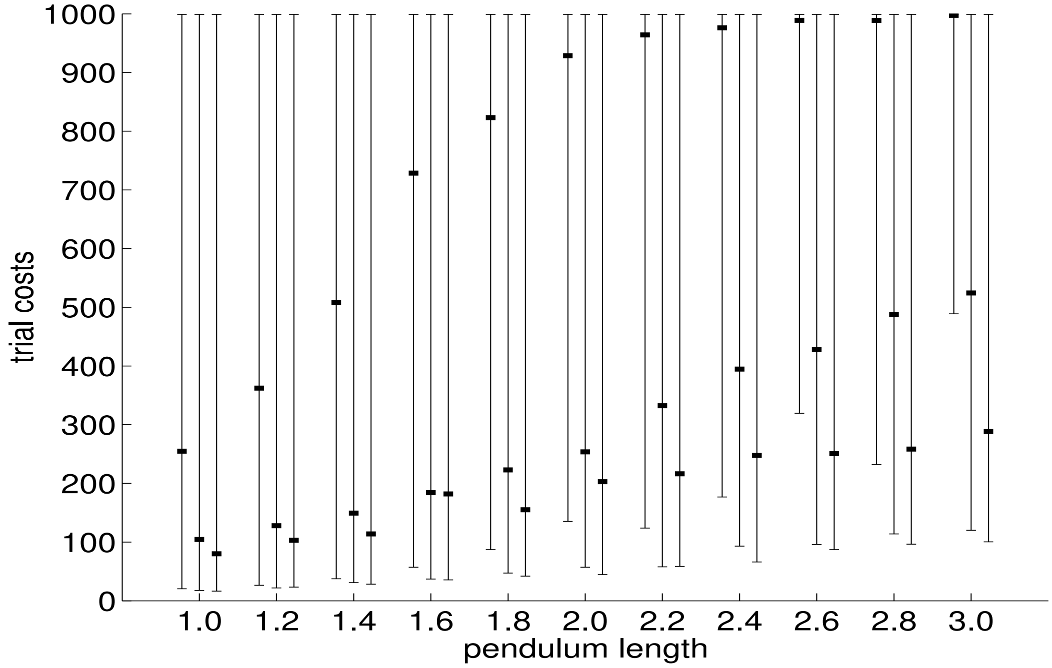

Figure 9.7. Performance during the first ten testing trials.

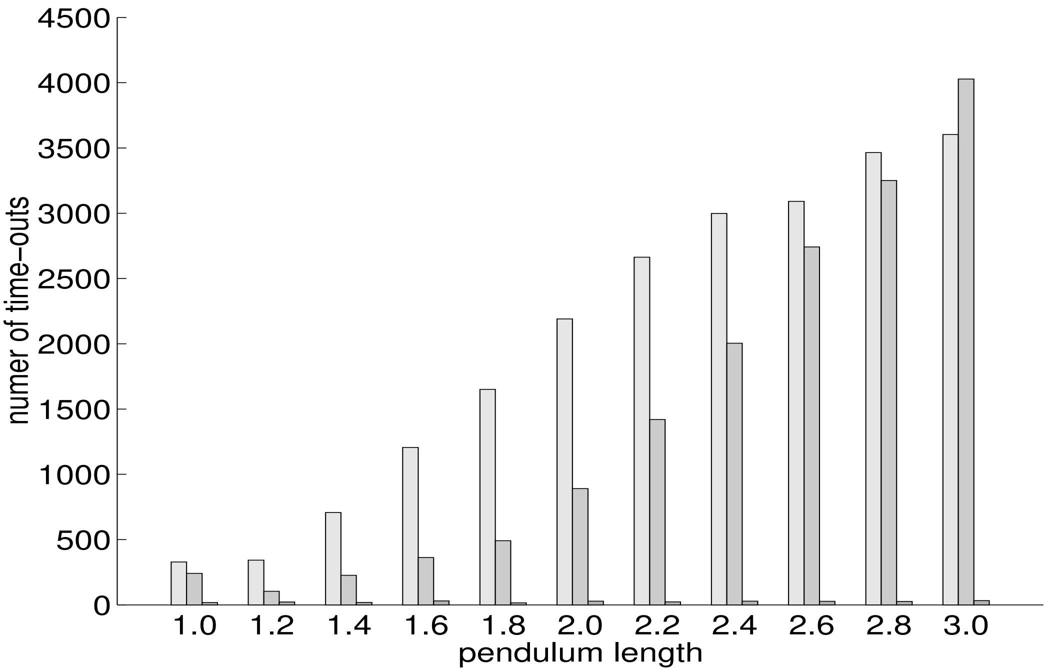

Figure 9.8. Total time-outs by pendulum length and action formulation.

	Formulation 1	Formulation 2	Formulation 3
Total time-outs	42,451	29,135	500
Time time spent	1.86×10^8	1.83×10^8	1.21×10^8
Costliest learning trial	999	309.28	275.17

Table 9.2. Some summary statistics.

and 3 in left-to-right order. From the plot, the benefit of the LL algorithm is strikingly clear. Over all, formulation 3 trials timed out far less often than the trials produced by formulations 1 and 2. The relative benefit grows as pendulum length increases and the problem becomes harder. For most lengths, formulation 2 trials timed out less than formulation 1 trials did, though this trends seems to be reversing for the longest pendula. Table 9.2 contains several statistics which further illustrate the benefit of the LL algorithm and of the presence of the energy-based action, MEA.

9.3 Discussion

In this chapter we demonstrated that one need not know a Lyapunov function in advance to be able to use Lyapunov-based methods to constrain and improve agent behavior. If one can provide a reasonable set of *candidate* Lyapunov functions, an agent using the LL algorithm can learn which are valid and use them to guide its behavior. Loss bounds can be established, ensuring that the learning process is not overly costly. In simulated pendulum swing-up experiments, in which the length of the pendulum was unknown to the agent a priori, we demonstrated the value of the LL algorithm in maintaining reasonable agent behavior during learning.

Another possible approach to ensuring safety when a Lyapunov function or the system dynamics are unknown in advance is to use control theoretic methods designed for that purpose. In the pendulum example, for instance, an agent could drive the pendulum for one time step using $+u_{max}$ torque, and then deduce, from the observed next state, the length of the pendulum. From then on, the appropriate Lyapunov function could be used to constrain action choices. This is an example of a system-identification approach. (Usually, identifying a system is not so easy as it is in this case, particularly when the dynamics are stochastic.)

Learning Lyapunov functions does have several benefits. First, explicit loss bounds can be derived. Second, the procedure reports failure if none of the hypothesized Lyapunov functions are valid. Thus, the agent's designer gets feedback on whether sufficient domain knowledge has been provided, or if further domain analysis is in order. Third, a candidate Lyapunov function can be valid for an environment even if it is based on a dynamics model that is not entirely correct. More study is warranted to distinguish the relative strengths of the LL approach and more standard approaches from control theory, such as system identification or the robust control techniques used by Kretchmar [44].

CHAPTER 10

CONCLUSION

Researchers in artificial intelligence have made great strides in imbuing computers with the ability to solve complex, real-world reasoning and decision problems. From planning space missions, to assisting the elderly, from playing chess, to acting as museum tour-guides, the development and fielding of AI systems is just beginning to blossom. As intelligent agents are increasingly deployed in settings where they must operate autonomously, with little or no supervision, new research issues come to the fore. In particular, it becomes highly desirable that agents act safely (not harming themselves, their environments, or others) and reliably.

In this thesis, we have demonstrated that control-theoretic techniques and domain knowledge can be useful in addressing these issues. In particular, we showed that safe, reliable performance can be guaranteed by using control-theoretic domain knowledge to design, or constrain, the ways in which an agent is allowed to interact with its environment. So designed, an agent can use existing, standard AI techniques to choose how to act, and we can rest assured that acceptable behavior will result. Below, we catalog the specific theoretical and experimental contributions of the thesis, preceded by the chapter and section numbers where they appear.

5.2 We defined six types of Lyapunov functions, two for Markov processes and four for Markov decision processes, that are particularly useful for establishing safety and reliability of AI systems.

7.1 We established sufficient conditions, based on Lyapunov functions, for the existence of action-sequence solutions (open-loop controls) and of optimal action-sequence solutions to deterministic, minimum cost-to-goal Markov decision processes.

7.2 We established sufficient conditions for the completeness of best-first search, uniform-cost search, depth-first branch-and-bound, and real-time search. We also discussed connections between Lyapunov functions, admissible heuristics, and roll-outs.

7.3,7.4 We demonstrated the process of designing safe, reliable agents for pendulum and robot arm control problems, and reported experimental results studying the safety, quality, and complexity of various heuristic search approaches.

8.1 We established sufficient conditions for safe, reliable behavior for very general classes of agents acting in stochastic environments.

8.2, 8.3, 8.4 We presented experiments studying the safety, reliability, and quality of reinforcement learning solutions to deterministic and stochastic pendulum and robot arm control problems.

9.1 We showed that Lyapunov functions need not be known in advance, but can be hypothesized by the agent designer and then tested by a learning agent. We established loss bounds on one algorithm for learning which, if any, of several hypothesized Lyapunov functions are truly Lyapunov.

9.2 We presented experiments using a pendulum swing-up problem that demonstrate the empirical benefits of the Learning Lyapunov functions approach.

Our work constitutes a beginning look at how control theory and artificial intelligence techniques can be usefully combined, but many questions remain. We mention several directions for future work below.

Safe anytime approximate optimal control: One thing demonstrated by the state-space search experiments in Chapter 7 is that when one varies the duration of a control choice, the complexity of finding a solution and the quality of the solution also vary. One approach to anytime optimal control would be to begin by solving the problem at a coarse time scale and then re-solve it at successively finer time scales as long as computational resources remain available. This recalls work on hierarchical planning, for example, in which planning is done on a high-level, and then the plan is refined until all the details are determined [74]. One of the challenges of this work is that a coarse, high-level plan is not always realizable. For example, a high-level travel plan might call for driving to the airport and then flying to New York, but it may turn out that one's car is in the shop. If Lyapunov or other control-theoretic principles are used to design an agent's actions, as proposed in this thesis, then a solution of any temporal resolution can always be executed successfully. Beyond exploring the potential advantages of such a scheme, relevant research questions include: How should the initial temporal resolution be chosen, and how should it be refined? Is it useful to vary temporal resolutions within a single solution attempt? How can information from temporally coarser solutions be used to inform the solution process for finer resolutions?

Does ensuring safety and reliability require sacrificing the quality of control? One of the recurring observations we have made is that constraining an agent so that safety and reliability are ensured can ultimately limit the performance that the agent is able to obtain. Is this tradeoff really necessary, or can one get both? How can different tradeoffs be achieved? Certainly, the methods we proposed for using Lyapunov functions to constrain agent behavior are simple. In pilot studies for the thesis, we experimented with stochastic Lyapunov descent constraints; that is, on each time step, with some probability an agent was constrained to descend on a Lyapunov function and otherwise the agent was not constrained [66]. We found that by varying the probability that the agent was constrained, the empirical performance of the agent varied smoothly between that of a fully-constrained agent and that of an unconstrained agent. Other ideas for more sophisticated uses of Lyapunov-based constraints include constraining the agent only after some time has passed, and allowing some amount of "uphill" actions in a given trial or in a given period of time. If this is done properly, safety and reliability properties can be retained, while affording the agent greater freedom to optimize the quality of control. What are the relative advantages of these different approaches? Can one design a method for using Lyapunov domain knowledge that can be guaranteed not to rule out the optimal policy?

Do Lyapunov ideas apply in more structured domains? Lyapunov methods are usually applied to lower-level control problems such as robot control and navigation, stabilization, and tracking. Can Lyapunov methods be applied to the more structurally-complex tasks often studied in the heuristic search, planning, and, sometimes, reinforcement learning communities? A suggestive piece of evidence comes from a recent paper by Hoffmann [58], who studied the "topology" of planning search spaces with respect to a particular estimate of plan quality based on relaxations of constraints. He found that for many benchmark planning problems, plan quality is nearly a Lyapunov function—often no uphill moves, or just a few, are necessary to arrive at an optimal plan. This suggests that methods based on Lyapunov function might have important applications in domains quite unlike those to which Lyapunov methods have traditionally been applied.

Robust/adaptive control problems: In Chapter 9 we presented one Lyapunov-based approach to dealing with environments with unknown dynamics. In control theory there are many approaches to this problem, including system identification, robust control and adaptive control. How can robust or adaptive control techniques be incorporated into AI agents? What is the relationship between those approaches and the method we propose?

Historically, research in control theory and AI has focussed on different sorts of problems. Control theory has largely been concerned with "low-level" regulation and tracking problems of the sort that arise in designing mechanical and electrical devices. AI has largely been concerned with "higher-level" problems such as planning, scheduling, theorem proving, and game-playing. (Research in reinforcement learning has been somewhat unusual in this regard, with problems of both sorts being considered.) In the drive to create autonomous agents that can operate successfully in complex, real-world environments, techniques from both disciplines are relevant. While many issues are yet to be explored, the approaches described in this thesis provide principled means for integrating techniques from control theory and AI in ways that ensure safe, reliable agent behavior and that allow efficient, tractable decision-making and control.

APPENDIX A

PROOF OF THEOREM 6.1

We recall the theorem being proved:

Theorem 6.1 *For any $\Delta > 0$, any $\mu > 0$, and appropriate ϵ_θ and $\epsilon_{\dot\theta}$, there exists $\delta > 0$, such that for any initial conditions $(\theta_0, \dot\theta_0)$, if the pendulum is controlled according to $MEA_{\mu,\epsilon_\theta,\epsilon_{\dot\theta}}$ for Δ time, then the mechanical energy of the pendulum is greater by at least δ at the end of that time.*

Proof: Let Δ, μ, ϵ_θ, and $\epsilon_{\dot\theta}$ be fixed, and assume $\epsilon_\theta < \sin^{-1}(\mu) - \sin^{-1}(\frac{1}{2}\mu)$. We show that for any initial conditions, $(\theta_0, \dot\theta_0)$, the increase in the pendulum's energy cannot be arbitrarily close to zero. First note that the change in ME is

$$\begin{aligned} \int_{t=0}^{\Delta} \dot\theta(t)u(t)dt &\geq \int_{t=0}^{\Delta} \frac{1}{2}|\dot\theta(t)|\mu dt \\ &\propto \int_{t=0}^{\Delta} |\dot\theta(t)|dt \ . \end{aligned} \tag{A.1}$$

We show that the last integral cannot be arbitrarily small.

First, note that if $|\dot\theta_0|$ is sufficiently large, the integral cannot be small because, with $|\ddot\theta|$ bounded at all times, $\dot\theta$ cannot get close to zero quickly enough to make the integral small. Suppose without loss of generality that $\dot\theta_0 > 0$ or $\dot\theta_0 = 0$ and $\theta_0 \geq 0$, and let $\theta_1 = -\pi + \sin^{-1}(u_{max}) - \epsilon_\theta$ and $\theta_2 = -\sin^{-1}(u_{max}) + \epsilon_\theta$. Consider Figure A.1. In the range of positions denoted by the solid line, and marked by A, MEA applies $+u_{max}$ control torque, which strictly exceeds the influence of gravity and accelerates the pendulum in the $+\theta$ direction. In the B range of positions, MEA

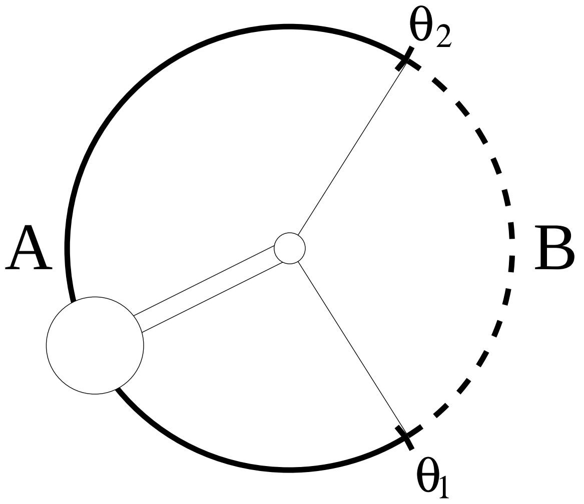

Figure A.1. Regions of positive and negative acceleration.

applies $+\frac{1}{2}u_{max}$ or $+u_{max}$ torque, which is outweighed by gravity, and the pendulum decelerates. Thus, at all times, $|\ddot{\theta}| > \ddot{\theta}_{min}$ for some $\ddot{\theta}_{min} > 0$.

Because we are only concerned with small velocities, $\dot{\theta}$ and $\ddot{\theta}$ can change sign at most a fixed number of times within Δ time. $\ddot{\theta}$ changes sign at the boundaries of regions A and B, and $\dot{\theta}$ can only change sign from positive to negative in the region B. If $\dot{\theta}$ changes sign in B, then the pendulum must swing all the way to the other side before another change in the sign of $\dot{\theta}$ may occur. So, within Δ time, only a fixed number of changes of sign of $\dot{\theta}$ and $\ddot{\theta}$ are possible. Let us call that number K.

For some $t_0 = 0 \leq t_1 \leq t_2 \leq t_K = \Delta$, the integral we are trying to bound above zero, A.1, can be rewritten as:

$$\sum_{k=0}^{K-1} \int_{t=t_k}^{t_{k+1}} |\dot{\theta}(t)| dt \ ,$$

where, in each summand, $\dot{\theta}$ and $\ddot{\theta}$ maintain the same sign over the course of the integral. Let us look at term i and let us assume, without loss of generality, that $\dot{\theta} \geq 0$ and $\ddot{\theta} \geq 0$ between t_i and t_{i+1}. Then:

$$
\begin{aligned}
\int_{t=t_i}^{t_{i+1}} \dot{\theta}(t)dt &= \int_{t=t_i}^{t_{i+1}} (\dot{\theta}(t_i) + \int_{\tau=t_i}^{t} \ddot{\theta}(\tau)d\tau)dt \\
&\geq \int_{t=t_i}^{t_{i+1}} \int_{\tau=t_i}^{t} \ddot{\theta}_{min} d\tau dt \\
&= \int_{t=t_i}^{t_{i+1}} \ddot{\theta}_{min}(t-t_i)dt \\
&= \ddot{\theta}_{min}\left(\frac{1}{2}t^2 - tt_i\right)\Big|_{t=t_i}^{t_{i+1}} \\
&= \ddot{\theta}_{min}\left(\frac{1}{2}t_{i+1}^2 - t_{i+1}t_i - \frac{1}{2}t_i^2 + t_i^2\right) \\
&= \frac{1}{2}\ddot{\theta}_{min}(t_{i+1}-t_i)^2 \ .
\end{aligned}
$$

Defining $\Delta_i = (t_{i+1} - t_i)$, we have

$$
\sum_{k=0}^{K-1} \int_{t=t_k}^{t_{k+1}} |\dot{\theta}(t)|dt \geq \sum_{k=0}^{K-1} \frac{1}{2}\ddot{\theta}_{min}\Delta_k^2 \ .
$$

How small can the right hand side be? Suppose we minimize it with respect to the Δ_k, subject to the constraint that $\sum_k \Delta_k = \Delta$. This is *the* textbook problem for the method of Lagrange multipliers, and the solution is the solution to the unconstrained minimization problem:

$$
\min_{\Delta_0,\ldots,\Delta_{K-1},\lambda} \frac{1}{2}\ddot{\theta}_{min}\sum_k \Delta_k^2 + \lambda\left(\sum_k \Delta_k - \Delta\right) \ .
$$

Omitting the details of setting derivatives to zero and solving the resulting system of equations, one finds that the minimum is attained when $\Delta_i = \Delta/K$ for all i. Thus,

$$
\sum_{k=0}^{K-1} \frac{1}{2}\ddot{\theta}_{min}\Delta_k^2 \geq \frac{1}{2}\ddot{\theta}_{min}\Delta^2/K \geq 0 \ ,
$$

which concludes the proof. $\square$

APPENDIX B

THE SARSA(λ) ALGORITHM

We briefly describe the Sarsa(λ) algorithm and provide some intuition behind its operation. A far more thorough discussion can be found in Sutton and Barto [88].

Recall that for finite MDPs with tabular storage of Q-values, the Sarsa algorithm uses an experience of the form $(s, a) \to (c, s', a')$ to update its action value estimates as:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(c + \gamma Q(s', a')) \ .$$

This can also be written as:

$$\begin{aligned} Q(s, a) &\leftarrow Q(s, a) + \alpha(c + \gamma Q(s', a') - Q(s, a)) \\ &= Q(s, a) + \alpha\delta(s, a, c, s', a') \ , \end{aligned}$$

where the definition of δ is obvious. The term δ can be viewed as an error. If $c + \gamma Q(s', a')$ is, say, greater than $Q(s, a)$, then $Q(s, a)$ may be too low and should be increased. In particular, the change in Q is taken to be α times the difference between the two terms. We say "may be too low" because the experience is just a random sample, and even if $Q(s, a)$ is perfectly correct, $c + \gamma Q(s', a')$ can be higher or lower than it.

Now, suppose that experience is followed by another one: $(s', a') \to (c', s'', a'')$. Suppose $\delta(s', a', c', s'', a'')$ is negative, indicating that $Q(s', a')$ may be too high. Working backwards, one suspects that $Q(s, a)$ may also be too high, if only because it was updated based on $Q(s', a')$, which may be high. Sarsa would decrease $Q(s', a')$ by α

times $\delta(s', a', c', s'', a'')$ and that is all. Sarsa(λ) additionally decreases $Q(s, a)$ by a similar amount, but downweighted by $\lambda\gamma$. In general, for a trajectory $\langle s_0, a_0, c_0, s_1, a_1, c_1, \ldots \rangle$, on time step $t \geq 1$, the Sarsa(λ) learning rule is to perform the following update for all $i \in \{0, \ldots, t-1\}$:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha(\lambda\gamma)^{t-i-1}\delta(s_{t-1}, a_{t-1}, c_{t-1}, s_t, a_t)$$

In the case that $s_t \in G$, so that there is no a_t, one uses $\delta(s_{t-1}, a_{t-1}, c_{t-1}, s_t) = c_{t-1} - Q(s_{t-1}, a_{t-1})$. In the case that state-action values are approximated using a differentiable function approximator, $\hat{Q}$, the updates above can be performed in gradient fashion. That is, defining $\delta(s, a, c, s', a') = c + \gamma\hat{Q}(s', a', \theta) - \hat{Q}(s, a, \theta)$ and $\delta(s, a, c, s') = c - \hat{Q}(s, a, \theta)$, the updates occurring at time t are:

$$\theta \leftarrow \begin{cases} \theta + \alpha(\lambda\gamma)^{t-i-1}\delta(s_{t-1}, a_{t-1}, c_{t-1}, s_t, a_t) & \text{if } s_t \notin G \\ \theta + \alpha(\lambda\gamma)^{t-i-1}\delta(s_{t-1}, a_{t-1}, c_{t-1}, s_t) & \text{if } s_t \in G \end{cases},$$

for all $i \in \{0, \ldots, t-1\}$.

APPENDIX C

CMAC FUNCTION APPROXIMATORS

CMACs are a popular function approximation architecture among reinforcement learning researchers for approximating value functions or action-value functions for continuous-state MDPs [87, 88]. A CMAC covers a hyper-rectangular region of state space, $R \subset \mathbb{R}^n$, with a finite set of layers (a.k.a. tilings or grids). Each layer divides R into a number of hyper-rectangular subregions, or cells, as depicted in Figure C.1. Assuming some tie-breaking rule for the boundaries between cells, each possible input point $x \in R$ falls into precisely one cell in each layer. A CMAC associates a "weight" to each cell. The output of the CMAC for an input x is the sum of the weights for each of the cells into which x falls.

Proper weights can be learned using gradient-based updating. Note that for a given input, the derivative of the output with respect to a weight is just one if the input falls into the cell to which that weight is associated, and zero otherwise. On an error of E resulting from an input x, the weight w associated to each cell containing the input is updated according to $w \leftarrow w + \alpha \frac{1}{N} E$, where α is a step size parameter and N is the total number of layers. The effect of the CMAC's structure is to generalize this error update to surrounding states. After such an update, for example, the CMAC output for all states in the intersection of the regions covered by the cells containing x changes by αE. For states that are covered by i of the N cells containing x, the output of the CMAC changes by $\alpha \frac{i}{N} E$. Thus, for states "farther" from x in the sense of i being smaller, the change in the CMAC's output is smaller. The CMAC thus "smooths" its updates over the state space.

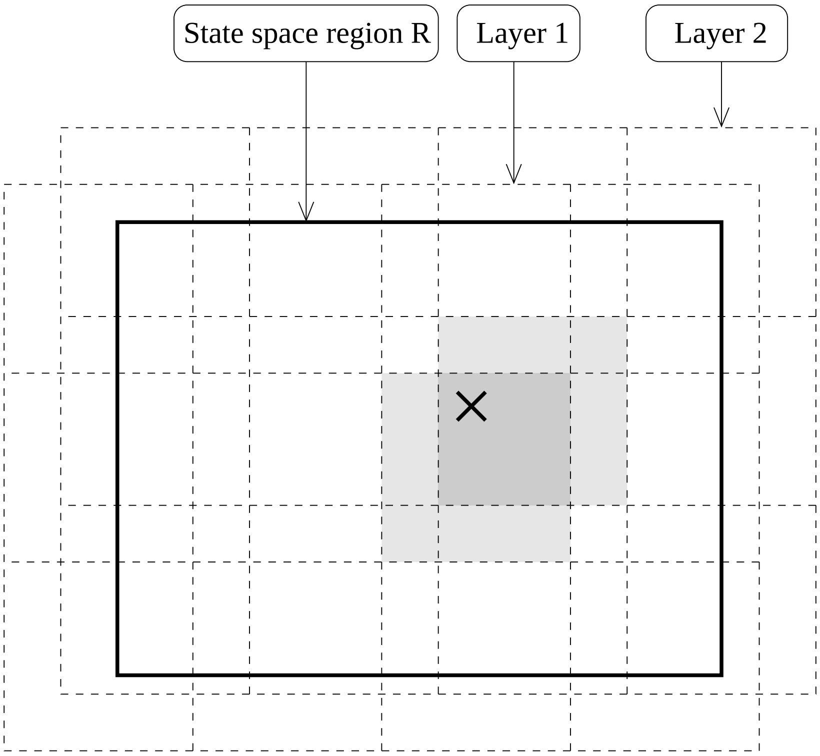

Figure C.1. Depiction of a CMAC.

High-resolution approximations can be achieved in two ways. The cells in each layer can be small, covering only a small portion of R, or there can be many layers with larger cells. Generally, the latter is preferred because the CMAC can then generalize broadly from a few data points, which is useful early in learning, while still being able to approximate a value or action-value function accurately in the limit.

BIBLIOGRAPHY

[1] AI Topics website at http://www.aaai.org/AITopics.

[2] Atkeson, C. G., and Schaal, S. Robot learning from demonstration. In *Machine Learning: Proceedings of the Fourteenth International Conference (ICML '97)* (San Francisco, CA, 1997), Jr. D. H. Fisher, Ed., Morgan Kaufmann, pp. 12–20.

[3] Başar, T. A robust adaptive algorithm for ABR congestion control. In *Proceedings of the Eleventh Yale Workshop on Adaptive and Learning Systems* (2001), pp. 153–160.

[4] Baird, L. C. Reinforcement learning in continuous time: Advantage updating. In *Proceedings of the International Conference on Neural Networks* (1994).

[5] Baird, L. C. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning* (1995), Morgan Kaufmann, pp. 30–37.

[6] Baird, L. C., and Moore, A. W. Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems 11* (1999), MIT Press.

[7] Baxter, J., and Bartlett, P. L. Reinforcement learning in POMDP's via direct gradient ascent. In *Proceedings of the Seventeenth International Conference on Machine Learning* (2000), P. Langley, Ed., pp. 41–48.

[8] Bertsekas, D. P. *Dynamic Programming and Optimal Control, Vol. 1.* Athena Scientific, 1995.

[9] Bertsekas, D. P. *Dynamic Programming and Optimal Control, Vol. 2.* Athena Scientific, 1995.

[10] Bertsekas, D. P., and Shreve, S. E. *Stochastic Optimal Control: The Discrete Time Case.* Academic Press, New York, 1978.

[11] Bertsekas, D. P., and Tsitsiklis, J. N. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

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

[13] Blondel, V. D., and Tsitsiklis, J. N. A survey of computational complexity results in systems and control. *Automatica 36*, 9 (2000), 1249–1274.

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

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

[16] Brafman, R. I., and Tennenholtz, M. R-MAX – a general polynomial time algorithm for near-optimal reinforcement learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (2001), pp. 953–958.

[17] Clouse, J., and Utgoff, P. A teaching method for reinforcement learning. In *Proceedings of the Ninth International Conference on Machine Learning, Aberdeen, Scotland* (1992), pp. 92–101.

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

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

[20] Crites, R. H., and Barto, A. G. Elevator group control using multiple reinforcement learning agents. *Machine Learning 33* (1998), 235–262.

[21] Deep Space One mission website at http://nmp.jpl.nasa.gov/ds1.

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

[23] Dietterich, T. G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research 13* (To Appear.), 227–303.

[24] Doran, J., and Michie, D. Experiments with the graph traverser program. *Proceedings of the Royal Society of London 294*, Series A (1966), 235–259.

[25] Fleming, W. H., and Soner, H. M. *Controlled Markov Processes and Viscosity Solutions*. Springer-Verlag, 1993.

[26] Freeman, R. A., and Kokotović, P. V. *Robust Nonlinear Control Design: State-Space and Lyapunov Techniques*. Birkhäuser, Boston, 1996.

[27] Gordon, D., and Subramanian, D. A multistrategy learning scheme for agent knowledge acquisition. *Informatica 17* (1994), 331–346.

[28] Gordon, D. F. Asimovian adaptive agents. *Journal of Artificial Intelligence Research 13* (2000), 95–153.

[29] Gullapalli, V. *Reinforcement Learning and its Application to Control.* PhD thesis, University of Massachusetts Amherst, 1992.

[30] Gullapalli, V., Barto, A. G., and Grupen, R. A. Learning admittance mappings for force-guided assembly. In *Proceedings of the 1994 International Conference on Robotics and Automation* (1994), pp. 2633–2638.

[31] Higham, D. J. An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM Review 43*, 3 (2001), 525–546.

[32] Hopfield, J., and Tank, D. W. 'Neural' computation of decisions in optimization problems. *Biological Cybernetics 52* (1985), 141–152.

[33] Huber, M., and Grupen, R. A. A feedback control structure for on-line learning tasks. *Robots and Autonomous Systems 22*, 2–3 (1997), 303–315.

[34] Huber, M., and Grupen, R. A. A control structure for learning locomotion gaits. In *Proceedings of the Seventh International Symposium on Robotic and Applications* (1998).

[35] Huber, M., and Grupen, R. A. Learning robot control–using control policies as abstract actions. In *NIPS'98 Workshop: Abstraction and Hierarchy in Reinforcement Learning, Breckenridge, CO* (1998).

[36] Kalman, R. E., and Bertram, J. E. Control system analysis and design via the "second method" of Lyapunov I: Continuous-time systems. *Transactions of the ASME: Journal of Basic Engineering* (1960), 371–393.

[37] Kearns, M., and Koller, D. Efficient reinforcement learning in factored MDPs. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence* (1999), pp. 740–747.

[38] Kearns, M., Mansour, Y., and Ng, A. Y. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (1999).

[39] Kearns, M., Mansour, Y., and Ng, A. Y. Approximate planning in large POMDPs via reusable trajectories. In *Advances in Neural Information Processing Systems 12* (Cambridge, MA, 2000), MIT Press.

[40] Kearns, M., and Singh, S. Near optimal reinforcement learning in polynomial time. In *Machine Learning: Proceedings of the Fifteenth International Conference* (1998), pp. 260–268.

[41] Kearns, M., and Singh, S. Finite-sample convergence rates for Q-learning and indirect algorithms. In *Advances in Neural Information Processing Systems 11* (1999), M. S. Kearns, S. A. Solla, and D. A. Cohn, Eds., pp. 996–1002.

[42] Koenig, S., and Simmons, R. G. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning 22* (1996), 227–250.

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

[44] Kretchmar, R. M. *A Synthesis of Reinforcement Learning and Robust Control Theory.* PhD thesis, Colorado State University, 2000.

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

[46] Kushner, H. *Introduction to Stochastic Control.* Holt, Rinehart and Winston, Inc., New York, 1971.

[47] Kushner, H. J., and Dupuis, P. *Numerical Methods for Stochastic Control Problems in Continuous Time.* Springer-Verlag, 1992.

[48] Levine, W. S., Ed. *The Control Handbook.* CRC Press, Inc., Boca Raton, Florida, 1996.

[49] Line, L. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning 8* (1992), 293–321.

[50] Maclin, R., and Shavlik, J. W. Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, WA* (1994), pp. 694–699.

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

[52] Meyn, S., and Tweedie, R. *Markov Chains and Stochastic Stability.* Springer-Verlag, New York, 1993.

[53] Moriarty, D. E., Schultz, A. C., and Grefenstette, J. J. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research 11* (1999), 199–229.

[54] Munos, R. A study of reinforcement learning in the continuous case by the means of viscosity solutions. *Machine Learning Journal 40* (2000), 265–299.

[55] Munos, R., and Moore, A. Variable resolution discretization in optimal control. *Machine Learning Journal* (To Appear).

[56] Nakamura, M., Baral, C., and Bjareland, M. Maintainability: a weaker stabilizability like notion for high level control. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence* (2000), pp. 62–67.

[57] Narendra, K. S., Ed. *Proceedings of the Eleventh Yale Workshop on Adaptive and Learning Systems* (2001).

[58] Nebel, B., Ed. *Local Search Topology in Planning Benchmarks: An Empirical Analysis* (San Francisco, 2001), Morgan Kaufmann.

[59] Neuneier, R., and Mihatsch, O. Risk sensitive reinforcement learning. In *Advances in Neural Information Processing Systems 11* (Cambridge, MA, 1999), M. Kearns, S. Solla, and D. Cohn, Eds., MIT Press, pp. 1031–1037.

[60] Ng, A., and Jordan, M. PEGASUS: A policy search method for large MDPs and POMDPs. In *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference* (2000).

[61] Pareigis, S. Multi-grid methods for reinforcement learning in controlled diffusion processes. In *Advances in Neural Information Processing Systems 9* (1997), MIT Press, pp. 1033–1039.

[62] Pareigis, S. Adaptive choice of grid and time in reinforcement learning. In *Advances in Neural Information Processing Systems 10* (1998), MIT Press, pp. 1036–1042.

[63] Parr, R. *Hierarchical control and learning for Markov decision processes.* PhD thesis, University of California, Berkeley, 1998.

[64] Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1984.

[65] Peng, J., and Williams, R. J. Incremental multi-step Q-learning. *Machine Learning*, 22 (1996), 283–290.

[66] Perkins, T. J. Lyapunov methods for reinforcement learning: A PhD thesis proposal draft. Tech. Rep. UM-CS-1999-060, Department of Computer Science, University of Massachusetts Amherst, 1999.

[67] Perkins, T. J., and Barto, A. G. Heuristic search in infinite state spaces guided by Lyapunov analysis. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (San Francisco, 2001), B. Nebel, Ed., Morgan Kaufmann, pp. 242–247.

[68] Perkins, T. J., and Barto, A. G. Lyapunov-constrained action sets for reinforcement learning. In *Machine Learning: Proceedings of the Eighteenth International Conference* (San Francisco, 2001), C. E. Brodley and A. P. Danyluk, Eds., Morgan Kaufmann, pp. 409–416.

[69] Precup, D. *Temporal Abstraction in Reinforcement Learning.* PhD thesis, University of Massachusetts Amherst, 2000.

[70] Puterman, M. L. *Markov Decision Processes: Disrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc, New York, 1994.

[71] Ram, A., Arkin, R. C., Boone, G., and Pearce, M. Using genetic algorithms to learn reactive control parameters for autonomous robotic navigation. *Adaptive Behavior 2*, 3 (1994), 277–305.

[72] Rosenstein, M. T., and Barto, A. G. Robot weightlifting by direct policy search. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (2001), vol. 2, pp. 839–844.

[73] Rummery, G. A., and Niranjan, M. On-line Q-learning using connectionist systems. Tech. Rep. CUED/F-INFENG/TR 166, Engineering Department, Cambridge University, 1994.

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

[75] Santamaria, J. C., Sutton, R. S., and Ram, A. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior 6*, 2 (1998), 163–218.

[76] Schaal, S. Learning from demonstration. In *Advances in Neural Information Processing Systems 9* (Cambridge, MA, 1997), M. C. Mozer, M. I. Jordan, and T. Petsche, Eds., MIT Press, pp. 1040–1046.

[77] Schneider, J. G. Exploiting model uncertainty estimates for safe dynamic control learning. In *Advances in Neural Information Processing Systems 9* (Cambridge, MA, 1997), M. C. Mozer, M. I. Jordan, and T. Petsche, Eds., MIT Press, pp. 1047–1053.

[78] Sepulchre, R., Janković, M., and Kokotović, P. V. *Constructive Nonlinear Control*. Springer-Verlag, 1997.

[79] Singh, S., Jaakkola, T., Littman, M. L., and Szepesvari, C. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning 38*, 3 (2000), 287–308.

[80] Singh, S. P., Barto, A. G., Grupen, R., and Connolly, C. Robust reinforcement learning in motion planning. In *Advances in Neural Information Processing Systems 6* (Cambridge, MA, 1994), J. D. Cowan, G. Tesauro, and J. Alspector, Eds., MIT Press, pp. 655–662.

[81] Slotine, J., and Li, W. *Applied Nonlinear Control*. Prentice-Hall, 1990.

[82] Sontag, E. D. A Lyapunov-like characterization of asymptotic controllability. *SIAM Journal of Control and Optimization 21* (1983), 462–471.

[83] Sontag, E. D. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer-Verlag, New York, 1990.

[84] Spong, M. W. The swing up control problem for the acrobot. *Control Systems Magazine 15*, 1 (1995), 49–55.

[85] Strens, M. J. A., and Moore, A. W. Direct policy search using paired statistical tests. In *Proceedings of the Eighteenth International Conference on Machine Learning* (2001), pp. 545–552.

[86] Sutton, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, 1984.

[87] Sutton, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8* (Cambridge, MA, 1996), D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds., MIT Press, pp. 1038–1044.

[88] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press/Bradford Books, Cambridge, Massachusetts, 1998.

[89] Sutton, R. S., McAllister, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12* (2000), MIT Press.

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

[91] Tesauro, G. J. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6*, 2 (1994), 215–219.

[92] Thrun, S., Beetz, M., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., Hahnel, D., Rosenberg, C., Roy, N., Schulte, J., and Schulz, D. Probabilistic algorithms and the interactive museum tour-guide robot minerva. *Journal of Robotics Research* (To Appear).

[93] Thrun, S., and Mitchell, T. Integrating inductive neural network learning and explanation-based learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (1993), pp. 930–936.

[94] Tsitsiklis, J. N., and Van Roy, B. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control 42*, 5 (1997), 674–690.

[95] Utgoff, P., and Clouse, J. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence, Anaheim, CA* (1991), pp. 596–600.

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

[97] Walsh, T. Stochastic constraint programming. In *Proceedings of the 2001 AAAI Fall Symposium on Using Uncertainty in Computation* (2001).

[98] Wang, Chia-Yu E., Timoszyk, Wojciech K., and Bobrow, James E. Payload maximization for open chained manipulators: finding weightlifting motions for a Puma 762 robot. *IEEE Transactions on Robotics and Automation 17*, 2 (2001), 218–224.

[99] Watkins, C. J. C. H. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[100] Weld, D., and Etzioni, O. The first law of robotics (a call to arms). In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), pp. 1042–1047.

[101] Whitehead, S. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence, Anaheim, CA* (1991), pp. 607–613.

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