Goal-Directed Acting with Incomplete Information

Sven Koenig

November 3, 1997
CMU-CS-97-199

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee

Reid Simmons, Chair
Tom Mitchell
Andrew Moore
Richard Korf (UCLA)

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored by the National Aeronautics and Space Administration, the Wright Laboratory, the Aeronautical Systems Center, the United States Air Force Materiel Command, and the Defense Advanced Research Projects Agency. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations and agencies or the U.S. Government.

Keywords: actuator uncertainty, agent architecture, agent-centered search, artificial intelligence, Baum-Welch method, deadlines, decision theory, decomposability of planning tasks, delivery robots, discounting, dynamic programming, Edge Counting method, Euler graphs (Eulerian domains), exponential utility functions, grid-worlds, high-stake decision making, interleaving of planning and plan execution, landmark-based navigation, localization, map uncertainty, metric-based navigation, minimax search, Min-Max Learning Real-Time A* (Min-Max LRTA*) method, moving-target search, Node Counting method, non-deterministic planning domains, one-shot planning (single-instance planning), passive learning, path planning, position estimation, preference models for planning, Q-Learning method, quicksand state space, race-track domain, real-time heuristic search, reinforcement learning, representation changes, reset state space, risk attitudes, robot navigation, sensor uncertainty, sliding-tile puzzles (eight puzzle), test-bed selection, totally and partially observable Markov decision process models, unsupervised learning, utility theory, Value-Iteration method.

Abstract

In the not too distant future, delivery robots will distribute parcels in office buildings and exploratory robots will roam the surface of other planets. Such situated agents must exhibit goal-directed behavior in real-time, even if they have only incomplete knowledge of their environment, imperfect abilities to manipulate it, limited or noisy perception, or insufficient reasoning speed. In this thesis, we develop efficient general-purpose decision-making methods for one-shot (that is, single-instance) planning tasks that enable situated agents to exhibit goal-directed behavior in the presence of incomplete information. The decision making methods combine search and planning methods from artificial intelligence with methods from operations research and utility theory.

We demonstrate how to use Partially Observable Markov Decision Process (POMDP) models to act, plan, and learn despite the uncertainty that results from actuator and sensor noise and missing information about the environment. We show how to use exponential utility functions to act in the presence of deadlines or in high-risk situations and demonstrate how to perform representation changes that transform planning tasks with exponential utility functions to planning tasks that standard search and planning methods from artificial intelligence can solve. Finally, we show how to decrease the planning time by interleaving planning and plan execution and present a real-time search method that allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks.

We use goal-directed robot-navigation tasks to illustrate the methods throughout the thesis, and present theoretical analyses, simulations, and experiments on a real robot.

Acknowledgements

Thanks to my advisor, Reid Simmons, for seven years of help. The other members of my thesis committee were Tom Mitchell, Andrew Moore, and Richard Korf. Thanks to them for their support during the whole process. Thanks also to the other academic mentors that I had over the years, both in computer science and business administration. This includes Ramakrishna Akella, Avrim Blum, Matthias Jantzen, Eric Krotkov, Bernd Neumann, Peter Norvig, Martha Pollack, Bernd Pressmar, Stuart Russell, Frieder Schwenkel, and Lotfi Zadeh.

The following colleagues influenced my thinking greatly: Lonnie Chrisman, Richard Goodwin, Matthias Heger, Illah Nourbakhsh, Jiří Sgall, Yury Smirnov, and Sebastian Thrun. Lonnie Chrisman, Richard Goodwin, Karen Haigh, Reid Simmons, and Joseph O'Sullivan implemented the parts of the robot system that my methods use. Swantje Willms performed some of the experiments. Boleslaw Szymanski thought about one of my open problems and apparently solved it (page 63). Joseph Pemberton made his maze generation program available to me. Other researchers that provided help and comments on this or earlier work include Jim Blythe, Justin Boyan, Howie Choset, Alan Christiansen, Fabio Cozman, Marek Druzdzel, Eugene Fink, Diana Gordon, Geoffrey Gordon, Karen Haigh, Robert Holte, Toru Ishida, Long-Ji Lin, Michael Littman, Matt Mason, Joseph O'Sullivan, Ronald Parr, Joseph Pemberton, Antony Stentz, Thomas Stricker, and Michael Wellman. Thanks to them for their help.

Lots of other people contributed indirectly to this thesis – far too many to list all of them. Thanks to all members of Reid Simmons' mobile robotics group, Tom Mitchell's learning lab, and the reinforcement learning group for many valuable discussions on a variety of topics. Thanks also to all of my friends and colleagues at Carnegie Mellon University, the University of California at Berkeley, the University of Hamburg, and elsewhere. I am sorry that I could not list all of your names here.

Most importantly, thanks to my wife and my parents for being so patient with me. Thanks for everything!

Contents

Chapter 1

Introduction

In the not too distant future, autonomous robots will traverse the surface of Mars, delivery robots will distribute parcels in office buildings, and softbots will help users navigate the World Wide Web. This thesis is about how agents like these should act in the world. It centers around efficient general-purpose decision-making methods for one-shot (that is, single-instance) planning tasks that enable them to exhibit goal-directed behavior in the presence of incomplete information. This uncertainty can arise from incomplete knowledge of the environment, imperfect abilities to manipulate it, limited or noisy perception, and insufficient reasoning speed. We develop component technologies for goal-directed acting in the presence of incomplete information and show where they might be useful (that is, provide a "proof of concept").

The Problem: An agent acts in a goal-directed way if it executes actions that change the current state of the domain to a goal state. Search and planning methods from artificial intelligence have investigated acting in this way, but traditionally they do not handle incomplete information. They solve tasks that can be interpreted as graph-search tasks in which the agent corresponds to a token, the domain to a graph, the states to vertices, and the actions to directed edges. The token always occupies a vertex and moves along the edges. The task is to find a plan that moves the token from its current vertex to a given goal vertex on the known graph. Since there is no uncertainty, this plan can be executed blindly in the world. These plans are called open-loop plans, sequential (or linear) plans, action sequences, or paths. Notice that we have not specified the planning process: artificial intelligence has investigated a variety of methods for finding such plans.

Incomplete information complicates the graph-search tasks. For example, the graph might not be known in advance, the current vertex of the token might not be certain, and edge traversals might move the token to one of a set of successor vertices. In all of these cases, the vertex of the token after an edge traversal cannot be predicted with certainty, and sensing might be required to solve the graph-search tasks. However, sensing might not uniquely identify the current vertex of the token either. Thus, the task is to find a plan with sensing that moves the token from its current vertex to a given goal vertex despite the uncertainty. These plans are called closed-loop plans or conditional plans.

Figure 1.1 illustrates the problem of goal-directed acting in the presence of incomplete information from the agent's point of view [Genesereth and Nilsson, 1986]. We develop solutions for the following problems: **Problem 1:** The agent needs to generate plans that reach the goal state although it might never be sure what its current state is. – If more than one such plan exists, the agent needs to choose one of them. Preferring the plan that minimizes the average plan-execution cost is an obvious choice, but this preference model has two problems for one-shot planning tasks (where a plan can be used only once). **Problem 2a:** First, search and planning methods that minimize the average plan-execution cost are time-consuming in the presence of incomplete information, and planning has to take this cost into account. A good plan for one-shot planning tasks is often one that minimizes the sum of planning and plan-execution cost. **Problem 2b:** Second, people tend not to minimize average cost for one-shot planning tasks but rather maximize average utility, for example, to take risk aspects into account. This is why the general public buys insurance even though the insurance

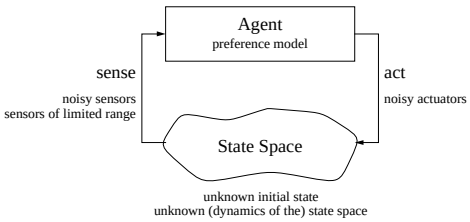


Figure 1.1: Sources of Incomplete Information

premium is usually much larger than the average loss from the insurance cause.

Our Approach: Search and planning methods from artificial intelligence provide a good basis for goal-directed acting since they can solve large search tasks efficiently. Contributions of artificial intelligence that we make use of in this thesis include, for example, taking advantage of heuristic knowledge (such as estimates of the goal distances) and the graph structure. This includes representing the graphs compactly (for example, with STRIPS rules), decomposing planning tasks into subtasks that can be solved independently (for example, with means-ends analysis), and planning on abstractions of the graphs. However, search and planning methods from artificial intelligence need to be extended to handle incomplete information. Research in artificial intelligence has recently begun to address this issue. We believe that it is promising to combine existing search and planning methods from artificial intelligence with other methods, including methods from artificial intelligence and methods from related disciplines, such as operations research and utility theory. We use the following methods to handle incomplete information:

- **Artificial Intelligence and Robotics**: We use agent-centered search methods to reduce the sum of planning and plan-execution cost. Agent-centered search methods are search methods that interleave planning and plan execution and plan only in the part of the domain around the current state of the agent. This is the part of the domain that is immediately relevant for the agents in their current situation. We also use minimax search from game-playing to plan with incomplete information if probabilities are not available.

- **Operations Research**: We use completely and partially observable Markov decision process models to model incomplete information with probabilities. Completely observable Markov decision process models are graphs with transition uncertainty. Partially observable Markov decision process models also have sensor uncertainty (and thus uncertainty about the current state). Operations research has investigated how to plan with these models and how to learn them. We also use methods from Markov game theory to efficiently represent and reason with minimax trees.

- **Utility Theory**: We use exponential utility functions to model both immediate soft deadlines and risk attitudes of people in high-stake one-shot planning domains, including risk-seeking behavior (such as gambling) and risk-averse behavior (such as holding insurance). Utility theory has investigated how to use utility functions to express preferences over plans.

This thesis presents three efficient methods for goal-directed acting in the presence of incomplete information. We apply them to different preference models and different kinds of incomplete information. We provide feasibility studies as "proof of concept" whenever necessary, but the emphasis of the thesis is on the development of the methods, and not their detailed experimental evaluation. Whether we stress theoretical results or applications depends on the subfield. If many applications already exist but the theory is not well understood, we concentrate on the theoretical aspects, and vice versa.

The thesis is organized as follows: It consists of three main chapters that are self-contained and can be read in any order. Each chapter contains a method that addresses one aspect of goal-directed acting in the presence of incomplete information. The chapters are structured similarly. They start with a general overview of the method. Then, the first section "Traditional Approaches" motivates the method and contrasts it with

traditional methods. The following sections explain the method, describe formal and experimental results, and contain pointers to related work. The section "Future Work" points out limitations and describes possible extensions. Finally, the section "Conclusions" summarizes the results of the chapter.

Chapter 2 on **Acting with Agent-Centered Search** addresses Problem 2a. It is about interleaving planning and plan execution to reduce the sum of planning and plan-execution cost. It assumes that probabilities are not available and develops minimax methods that attempt to minimize the worst-case plan-execution cost. Chapter 3 on **Acting with POMDPs** addresses Problem 1. It is about using partially observable Markov decision process models (POMDPs) to represent uncertainty and reason with it, including planning and learning improved models. It assumes that probabilities are available and develops methods that attempt to minimize the average plan-execution cost. Finally, Chapter 4 on **Acting with Nonlinear Utility Functions** addresses Problem 2b. It is about using exponential utility functions to plan in the presence of immediate soft deadlines and in high-stake one-shot planning domains. It assumes that probabilities are available and develops methods that attempt to maximize the average utility of the plan-execution cost, where the utility is an exponential function of the cost.

An Application: Although our methods apply to a variety of agents and tasks, we believe in using a common domain to ground the research and illustrate the resulting search and planning methods. We use autonomous mobile robots and goal-directed navigation tasks (how to get to a given goal location) as examples. These are real-world tasks that are important for delivery robots, including office or hospital delivery robots. They are interesting for us because robots have to deal with various kinds of incomplete information, and the amount of uncertainty can be fairly substantial.

- **Actuator Uncertainty**: For example, motion commands are not always carried out perfectly due to wheel slippage and mechanical tolerances, resulting in dead-reckoning error (Chapter 3).
- **Sensor Uncertainty**: Noisy sensors can produce false positives (report features that are not present) or false negatives (fail to detect features that are present). For example, ultrasonic waves can bounce around several surfaces before returning to the robot and therefore do not necessarily give correct information on the distance to the closest surface (Chapter 3).
- **Limited Sensing Information**: For example, ultrasonic sensors provide only distance information to nearby objects. They are not able to perceive other information (such as colors) and their sensing range is restricted to only a few meters around the robot (Chapters 2, 3, and 4).
- **Uncertainty in the Interpretation of the Sensor Data**: For example, ultrasonic sensor data often do not allow the robot to clearly distinguish between walls, closed doors, and lines of people blocking a corridor opening (Chapter 3).
- **Map Uncertainty**: For example, the lengths of corridors might not be known exactly. It might only be known, for instance, that a particular corridor is between two and nine meters long (Chapter 3).
- **Uncertainty about the Initial Pose of the Robot** (Position and Orientation) (Chapters 2 and 3)
- **Uncertainty about the Dynamic State of the Environment**: For example, blockages can change over time as people close and open doors and block and unblock corridors (Chapter 4).

Robots have to deal with more than just incomplete information. They also have to service navigation requests efficiently, otherwise people will not use them: time is an extremely limited resource for robots. Also, the memory, running time, and (for learning) training data requirements of the methods have to be reasonably small, making goal-directed robot navigation a challenging task for goal-directed acting in the presence of incomplete information.

Throughout the chapters, we show how our search and planning methods fit into a standard mobile robot architecture (Figure 1.2). Such an architecture is usually based on layers that reflect the various functions that mobile robots have to perform. Higher layers work with more abstract representations of the sensor

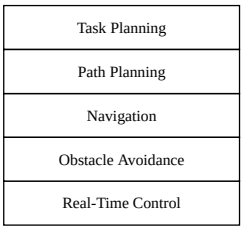


Figure 1.2: Robot Architecture

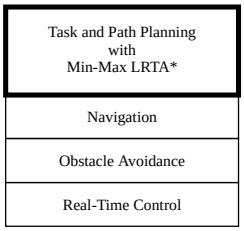


Figure 1.3: Robot Architecture (Chapter 2)

Navigation Task	goal-directed maze navigation or localization
Actuator Uncertainty	perfect actuators
Sensor Uncertainty	perfect sensors with limited range
Map Uncertainty	known map
Initial Pose Uncertainty	arbitrary pose uncertainty

Figure 1.4: Robot-Navigation Tasks (Chapter 2)

data and control actions and thus less detail. This allows them to concentrate on the essentials and be more efficient. Robots have to determine which location to visit next (task planning), plan a path to that location (path planning), follow that path reliably (navigation), and avoid obstacles in the process (obstacle avoidance). They also have to control the motors and interpret the raw sensor data (real-time control). We apply our search and planning methods to the higher layers: the navigation, path-planning, and task-planning layers.

Overview of Chapter 2 on Acting with Agent-Centered Search: In Chapter 2, we study agent-centered search methods. Our goal is to reduce the sum of planning and plan-execution time over planning methods that first plan and then execute the resulting optimal plan, which can be intractable. Interleaving planning and plan execution is a general principle for reducing the planning time, that also decreases the sum of planning and plan-execution time for sufficiently fast moving agents that solve one-shot planning tasks.

We develop Min-Max LRTA*, an efficient agent-centered search method for nondeterministic domains that is a generalization of LRTA* [Korf, 1990]. Min-Max LRTA* assumes that probabilities are not available and uses a minimax method to attempt to minimize the worst-case plan-execution cost. We apply Min-Max LRTA* to path and task planning for goal-directed navigation and localization in mazes (Figure 1.3). The robot has perfect actuators, perfect sensors, and perfect knowledge of the maze, but is uncertain about its start pose (Figure 1.4). Min-Max LRTA* allows the robot to gather information early that can be used to reduce its pose uncertainty, which can cut down the planning time needed. Min-Max LRTA* differs from previous methods that interleave planning and plan execution in nondeterministic domains in that it allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks.

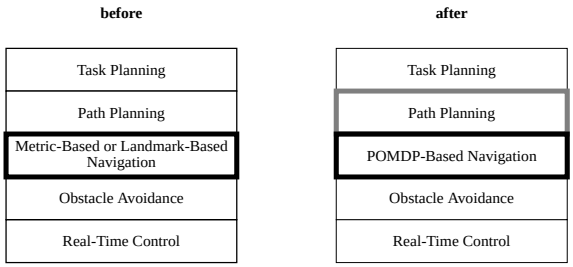


Figure 1.5: Robot Architecture (Chapter 3)

Navigation Task	goal-directed office navigation
Actuator Uncertainty	noisy (that is, unreliable) actuators
Sensor Uncertainty	noisy sensors with limited range
Map Uncertainty	known topology, but uncertain or unknown distances
Initial Pose Uncertainty	limited (in our implementation) or arbitrary (in theory) pose uncertainty

Figure 1.6: Robot-Navigation Tasks (Chapter 3)

Min-Max LRTA* is a general-purpose (domain-independent) search method that applies not only to robot navigation but to all search tasks in nondeterministic domains in which the minimax goal distance of every state is finite. We study its performance, and the performance of related agent-centered search methods, to understand better why they work, when they work, and how well they work. The complexity results provide a theoretical foundation for interleaving planning and plan execution. They can, for example, be used to choose test-beds for real-time search methods and representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods.

Overview of Chapter 3 on Acting with POMDPs: In Chapter 3, we study partially observable Markov decision process models. POMDPs represent uncertainty with probabilities and reason with them, including planning and learning improved models. Our goal is to make goal-directed acting with incomplete information more reliable. This addresses the problem that agents have to cope with a substantial amount of uncertainty, more than we assumed in Chapter 2.

We develop efficient methods for POMDP planning and POMDP learning that have small memory, running time, and training data requirements. The POMDP planning methods assume that probabilities are available or can be learned and use methods that attempt to minimize the average plan-execution cost. We use them to implement the navigation layer of a robot architecture, including pose estimation, planning how to follow a given path, navigation, and learning (Figure 1.5). We also show how they could be used to implement the path-planning layer. In our experiments, the robot has to move to a goal pose in an office environment with known topology but perhaps unknown distances as well as unreliable hardware (Figure 1.6). Our POMDP-based navigation architecture provides the robot with a uniform and theoretically grounded framework for modeling uncertainty in actuation, uncertainty in sensing and sensor data interpretation, uncertainty in the start pose of the robot, and uncertainty of the distance information in the topological map. It maintains a probability distribution over all poses (rather than a single estimate of the current robot pose) and updates the probability distribution using both motion reports and sensor reports about landmarks in the environment. Using both kinds of information improves upon metric-based navigation methods, that do not utilize sensor reports, and landmark-based navigation methods, that do not utility motion reports. It also improves upon Kalman filter-based navigation methods, that utilize both sensor and motion reports but cannot represent arbitrary probability distributions. To summarize, our work results both in a new robot navigation architecture and a novel application area for POMDPs.

Overview of Chapter 4 on Acting with Nonlinear Utility Functions: In Chapter 4, we study exponential

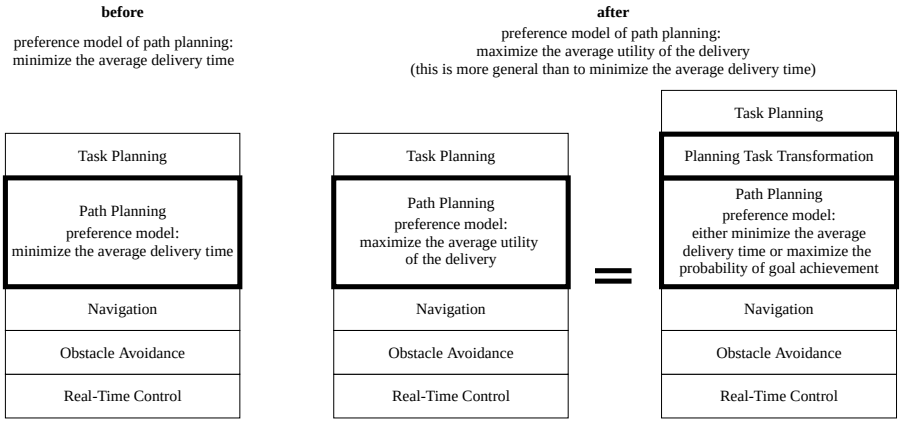


Figure 1.7: Robot Architecture (Chapter 4)

utility functions. In Chapters 2 and 3, we studied planning methods that attempt to find plans that achieve the goal with minimal worst-case plan-execution cost or minimal average plan execution cost. Most probabilistic search and planning methods from artificial intelligence attempt to find plans with maximal probability of goal achievement or, if the goal can be achieved for sure, plans that achieve the goal with minimal average plan-execution cost. Many planning methods from robotics, on the other hand, attempt to find plans that achieve the goal with minimal worst-case plan-execution cost. Our goal is to extend these preference models to enable planning with immediate soft deadlines (as specified for delivery tasks) and planning in high-stake one-shot planning domains with a continuum of risk attitudes, including risk-seeking behavior (such as gambling) and risk-averse behavior (such as holding insurance).

We develop efficient planning methods that assume that probabilities are available and attempt to find plans that achieve the goal with maximal average utility, where the utility is an exponential function of the plan-execution cost. Exponential utility functions preserve the decomposability of planning tasks, which allows for efficient planning. They can also trade-off between minimizing the worst-case, average, and best-case plan-execution cost. The key idea of our methods is to perform representation changes that transform planning tasks with exponential utility functions to planning tasks that standard search and planning methods from artificial intelligence can solve, including those that do not reason about plan-execution costs at all. The transformations, which we call the additive and multiplicative planning-task transformations, are simple context-insensitive representation changes that can be performed locally on various representations of planning tasks. We use path-planning for goal-directed navigation tasks in the presence of blockages to motivate the planning-task transformations and argue that they can easily be integrated into robot architectures by first transforming the path-planning task and then using the existing path planner of the architecture unchanged on the resulting planning task (Figure 1.7).

The results of Chapter 4 apply more generally to finding plans that achieve the goal with maximal average utility, where the utility is an exponentially decreasing function of the consumption of exactly one resource, such as time, energy, or money.

Summary: To summarize, we develop efficient general-purpose search and planning methods that solve one-shot planning tasks for goal-directed acting in the presence of incomplete information. We do this by combining search and planning methods from artificial intelligence with methods from other disciplines, namely, operations research and utility theory. We illustrate the resulting methods using goal-directed robot-navigation tasks and present theoretical analyses, simulations, and experiments on a real robot.

Now sit back, relax, and enjoy the rest of the thesis!

Chapter 2

Acting with Agent-Centered Search

Traditional search methods from artificial intelligence, such as the A* method [Nilsson, 1971], first plan and then execute the resulting plan. *Agent-centered search methods*, on the other hand, are similar to game-playing programs in that they interleave planning and plan execution, and plan only in the part of the domain around the current state of the agents. This is the part of the domain that is immediately relevant for the agents in their current situation. We study *Learning Real-Time A* (LRTA*)-type real-time search methods*, those agent-centered search methods that search forward from the current state of the agents and associate information with the states to prevent cycling. Experimental evidence indicates that LRTA*-type real-time search methods are efficient, and often outperform traditional search methods in deterministic domains. They have, for example, been applied to STRIPS-type planning tasks [Bonet *et al.*, 1997] and satisfiability problems [Smirnov and Veloso, 1997].

We illustrate that LRTA*-type real-time search methods can also be used to speed up problem solving in nondeterministic domains, where the successor states of action executions are not uniquely determined. LRTA*-type real-time search methods allow agents to gather information early. This information can then be used to resolve uncertainty caused by nondeterminism and reduce the amount of planning done for irrelevant (unencountered) situations. Instead of planning for every contingency, LRTA*-type real-time search methods execute the partial plan obtained after a bounded amount of planning, observe the resulting successor state, and then plan for only the resulting successor state, not for all states that could have resulted from the plan execution. We develop Min-Max LRTA*, a generalization of LRTA* [Korf, 1990] to nondeterministic domains. Min-Max LRTA* allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks. To illustrate its advantages, we apply Min-Max LRTA* to path and task planning for robots that have to perform goal-directed navigation and localization tasks in mazes, but are uncertain about their start pose (position and orientation).

The few existing performance characterizations of LRTA*-type real-time search methods are mostly of an experimental nature. We feel that it is not only important to extend the application domains of LRTA*-type real-time search methods, but also to provide some of the foundations for understanding their performance. We therefore study, both formally and experimentally, how heuristic knowledge and domain properties influence the performance of Min-Max LRTA* and related LRTA*-type real-time search methods. We study the influence of the number of states and actions of the domains, their maximal goal distance, and whether they are Eulerian. We show, for example, that Eulerian domains (a superset of undirected domains) are easy to search with many LRTA*-type real-time search methods, even those that can be intractable, and introduce reset state spaces, quicksand state spaces, and "complex state spaces" that are not as easy to search. We also introduce measures of task size that predict how easy it is to search domains with LRTA*-type real-time search methods. The measures explain, for example, why LRTA*-type real-time search methods can easily solve sliding-tile puzzles, grid-worlds, and other traditional search domains from artificial intelligence. The

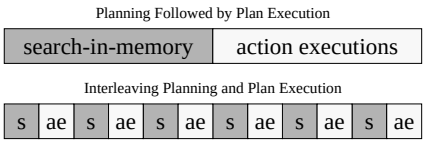


Figure 2.1: Planning and Plan Execution

measures can also be used to choose test-beds for LRTA*-type real-time search methods and representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods.

To summarize, we study how to act efficiently by interleaving planning and plan execution. Our main contributions are the following: First, we develop Min-Max LRTA*, an LRTA*-type real-time search method for nondeterministic domains and apply it to robot-navigation tasks. Second, we study the performance of LRTA*-type real-time search methods, both formally and experimentally, and apply the results to choosing test-beds for LRTA*-type real-time search methods and representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods.

In the following, Section 2.1 contrasts traditional search methods with LRTA*-type real-time search methods. Section 2.2 describes LRTA* as a prototypical example. Section 2.3 introduces Min-Max LRTA*, discusses its advantages and disadvantages, and presents both theoretical complexity results and experimental average-case performance results. Section 2.4 introduces the robot-navigation tasks, applies Min-Max LRTA* to them, gives experimental average-case performance results, and discusses related search methods. Section 2.5 studies in more detail how domain properties influence the performance of LRTA*-type real-time search methods and shows how these results can be used to choose test-beds for LRTA*-type real-time search methods and representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods. Finally, Section 2.6 describes possible extensions, Section 2.7 lists future work, and Section 2.8 summarizes our conclusions.

2.1 Traditional Approaches

Situated agents are agents that have to act in the world to achieve their goals. A central problem for them is how to achieve given goal states. Traditional search methods usually solve given search tasks off-line in a mental model of the world (search-in-memory). The agents then execute the resulting plans. This way, planning and plan execution are completely separated. Situated agents, however, can solve search tasks on-line by interleaving planning and plan execution (Figure 2.1). Search methods that interleave planning and plan execution have been developed in artificial intelligence and other disciplines of computer science, for example in the areas of

- Planning

  Examples include [Morgenstern, 1987, Ambros-Ingerson and Steel, 1988, Sanborn and Hendler, 1988, Boddy and Dean, 1989, Olawsky *et al.*, 1993, Dean *et al.*, 1995, Stone and Veloso, 1996].

- Reinforcement Learning (see Section 2.5.6)

  Examples include [Ring, 1992, McCallum, 1995b, Wiering and Schmidhuber, 1996].

- Applied Robotics

  Examples include [Balch and Arkin, 1993, Choset and Burdick, 1994, Stentz and Hebert, 1995, Nourbakhsh and Genesereth, 1996].

- Theoretical Computer Science and Theoretical Robotics

  Examples include [Lumelsky, 1987, Blum *et al.*, 1991, Papadimitriou and Yannakakis, 1991, Rao *et al.*, 1991, Zelinsky, 1992, Foux *et al.*, 1993, Betke *et al.*, 1995].

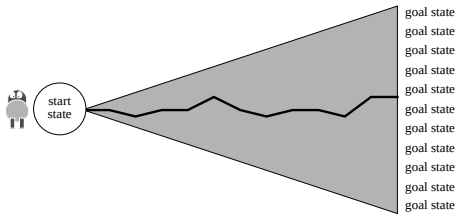


Figure 2.2: Planning Followed by Plan Execution

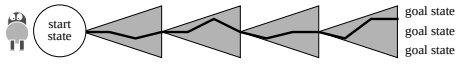


Figure 2.3: Interleaving Planning and Plan Execution

Interleaving planning and plan execution has several advantages [Koenig, 1995]. In this chapter, we study the following advantage: Situated agents have to take their planning time into account to make rational decisions [Good, 1971]. For one-shot (that is, single-instance) planning tasks, for example, they should attempt to minimize the sum of planning and plan-execution time. Finding plans that minimize the plan-execution time is often intractable. Thus, finding and then executing such plans is often not optimal. Interleaving planning and plan execution is a general principle that can decrease the sum of planning and plan-execution time for sufficiently fast moving agents because it reduces the planning time. Since actions are executed before their complete consequences are known, interleaving planning and plan execution is likely to incur some overhead in terms of the number of actions executed, but this is often outweighed by the computational savings gained. In the following, we illustrate this point for nondeterministic domains, where interleaving of planning and plan execution reduces the amount of planning done for irrelevant (unencountered) situations.

In nondeterministic domains, the successor states of action executions are not uniquely determined. Instead of planning for every contingency, an agent that interleaves planning and plan execution executes the partial plan obtained after a bounded amount of planning, observes the resulting successor state, and then plans for only the resulting successor state, not for all states that could have resulted from the execution of the partial plan. Figures 2.2 and 2.3 illustrate the ideal situation [Nourbakhsh, 1997]. Without interleaving of planning and plan execution, the agent has to find a large conditional plan that solves the planning task. When interleaving planning and plan execution, on the other hand, the agent has to find only the beginning of such a plan. After the execution of this partial plan, the agent repeats the process from the state that actually resulted from the execution of the partial plan. Assume that the planning effort is roughly linear in the size of a plan (the area of its triangle in Figures 2.2 and 2.3). One can then hope that the number of partial plans executed when interleaving planning and plan execution is small enough so that the total size of all partial plans is smaller than the size of the complete plan that solves the planning task without interleaving of planning and plan execution. This is not guaranteed since the agent executes partial plans before their complete consequences are known and thus cannot be sure that their execution is really as advantageous as anticipated. It follows that it is important to study the performance in detail that results from interleaving of planning and plan execution.

Figure 2.4 visualizes our classification of search methods that interleave planning and plan execution and gives examples of search methods that we discuss later in this chapter: *Agent-centered search methods* [Koenig, 1995, Koenig, 1996] or, synonymously, agent searching methods [Dasgupta *et al.*, 1994], restrict the search-in-memory to a small part of the domain that can be reached from the current state of the agent with a small number of action executions (Figure 2.5). This is the part of the domain that is immediately

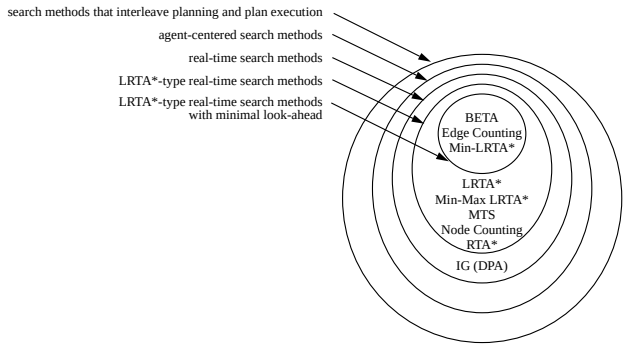


Figure 2.4: Subsets of Search Methods (including Example Search Methods)

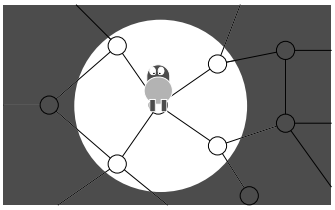


Figure 2.5: Agent-Centered Search

relevant for the agent in its current situation. *Real-time search methods* are agent-centered search methods that search forward from the current state of the agent. *LRTA*-type real-time search methods* are real-time search methods that associate information with the states to prevent cycling. Finally, *LRTA*-type real-time search methods with minimal look-ahead* do not even project one action execution ahead. We discuss them in the second part of this chapter.

LRTA*-type real-time search methods are promising search methods for interleaving of planning and plan execution because they are efficient general-purpose (domain-independent) search methods that allow for fine-grained control over how much planning to do between plan executions, use heuristic knowledge to guide planning, and improve their performance over time as they solve similar planning tasks. We therefore study them in more detail in the following.

2.2 LRTA*

In this section, we describe the prototypical example of an LRTA*-type real-time search method: the *Learning Real-Time A* method* (LRTA*) [Korf, 1990]. A more comprehensive introduction to LRTA* and other LRTA*-type real-time search methods is given in [Ishida, 1997]. We use the following notation: S denotes the finite set of states of the domain, $s_{start} \in S$ the start state, and $G \subseteq S$ the set of goal states. $A(s) \neq \emptyset$ is the finite, nonempty set of actions that can be executed in state $s \in S$. $succ$ is the *transition function*, a function from states and actions to sets of states: $succ(s, a)$ denotes the set of successor states that can result from the execution of action $a \in A(s)$ in state $s \in S$. We measure distances in action executions, which is reasonable if every action has the same immediate cost, for example, can be executed in the about the same amount of time. Section 2.6 discusses how to extend the results of this chapter to actions whose costs are nonhomogeneous. The number of states is $n := |S|$, and the number of state-action pairs (loosely called *actions*) is $e := \sum_{s \in S} |A(s)|$, that is, an action that is applicable in more than one state counts more than once. We also use two operators with the following semantics: Given a set X, the expression "one-of X" returns an element of X according to an arbitrary rule (that can, for example, include elements of chance). A subsequent

Initially, the u-values $u(s)$ are approximations of the negative goal distances (measured in action executions) for all $s \in S$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. $a :=$ one-of $\arg\max_{a \in A(s)} u(succ(s, a))$.
4. $u(s) := \min(u(s), -1 + u(succ(s, a)))$.
5. Execute action a, that is, change the current state to $succ(s, a)$.
6. $s :=$ the current state.
7. Go to 2.

Figure 2.6: LRTA* with Look-Ahead One

invocation of "one-of X" can return the same or a different element. The expression "$\arg\max_{x \in X} f(x)$" returns the elements $x \in X$ that maximize $f(x)$, that is, the set $\{x \in X | f(x) = \max_{x' \in X} f(x')\}$. We use this notation and these assumption throughout this chapter. In particular, we always assume that the number of states is finite and that at least one action can be executed in every state. LRTA* operates on deterministic domains only. Thus, the set $succ(s, a)$ contains only one state and we use $succ(s, a)$ also to denote this state.

LRTA* associates a small amount of information with the states that allows it to remember where it has already searched. In particular, it associates a *u-value* $u(s)$ with each state $s \in S$.[1] The u-values approximate the negative goal distances of the states. The negative goal distance of the start state of the domain in Figure 2.8 is, for example, -14 action executions.[2] The u-values are updated as the search progresses and used to determine which actions to execute. Here, we describe LRTA* with look-ahead (search horizon) one (Figure 2.6). It consists of a *termination-checking step* (Line 2), an *action-selection step* (Line 3), a *value-update step* (Line 4), and an *action-execution step* (Line 5). LRTA* first checks whether it has already reached a goal state and thus can terminate successfully (Line 2). If not, it decides which action a to execute in the current state s (Line 3). It looks one action execution ahead and always greedily chooses an action that leads to a successor state with a maximal u-value (ties are broken arbitrarily). If the u-value of this successor state minus one is smaller than the u-value of the current state, then it replaces the u-value of the current state (Line 4). Finally, LRTA* executes the selected action (Line 5), updates the current state (Line 6), and iterates the procedure (Line 7). The planning time of LRTA* between plan executions is linear in the number of actions available in the current state. If this number does not depend on the number of states, then the planning time between plan executions does not depend on the number of states either.

The action-selection and value-update steps of LRTA* can be explained as follows. **The action-selection step** attempts to get to a goal state as fast as possible by always choosing an action that leads to a successor state with a maximal u-value. Since the u-values approximate the negative goal distances, this is a successor state that is believed to have a smallest goal distance. **The value-update step** makes the u-value of the current state approximate the goal distance of the state better. The goal distance of a nongoal state equals one plus the minimum of the goal distances of its successor states. Consequently, $-(1 + \min_{a \in A(s)}[-u(succ(s, a))])$ approximates the negative goal distance of nongoal state s. Assume that all u-values overestimate the negative goal distances. (This corresponds to admissible heuristic values for A* search, except that we use negative heuristic values instead of positive ones.) Then, both $-(1 + \min_{a \in A(s)}[-u(succ(s, a))])$ and $u(s)$ overestimate the negative goal distance of nongoal state s and the smaller of these values is the better estimate. The value-update step uses this value to replace the u-value of nongoal state s. LRTA* was originally stated with a different value-update step, namely "$u(s) := -1 + u(succ(s, a))$" [Korf, 1990]. This is a simplified variant of the value-update step, that we explain in Section 2.3.3.1.

This description shows that LRTA* is an asynchronous incremental dynamic programming method [Barto *et*

[1] The term "u-values" is commonly used in reinforcement learning (Section 2.5.6) for values that are associated with the states.

[2] LRTA* is usually stated with u-values that approximate the positive goal distances. We deviate from this to be consistent with the subsequent chapters. This has the added advantage that our terminology is consistent with the terminology of reinforcement learning, to which we apply our complexity results later in this chapter. Reinforcement learning associates immediate rewards, not costs, with actions. The immediate rewards are obtained when the corresponding actions are executed. We use negative immediate rewards to model costs.

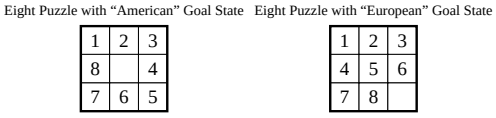


Figure 2.7: Two Possible Goal States of the Eight Puzzle

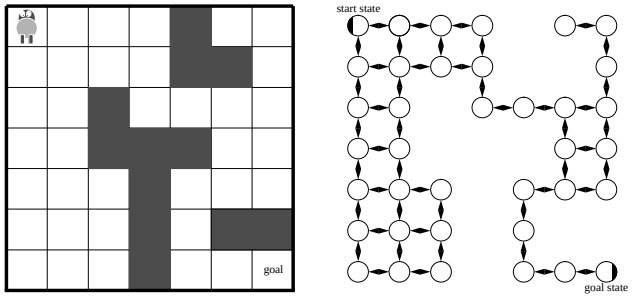


Figure 2.8: Grid-World

al., 1989]. It has the following properties (under reasonable assumptions): It interleaves planning with plan execution, and its look-ahead determines how much it plans between plan executions. Planning is guided via the u-values, that are initialized using heuristic knowledge of the domain, namely with approximations of the negative goal distances of the states. Finally, LRTA* reaches a goal state with a finite number of action executions and, if it is reset to the start state whenever it reaches a goal state, eventually converges to a behavior that reaches a goal state with a minimal number of action executions [Korf, 1990, Koenig, 1992]. We discuss these and other properties of LRTA* and the underlying assumptions in more detail in Section 2.3.3 when we discuss a generalization of LRTA*.

LRTA* and other LRTA*-type real-time search methods have mostly been applied to deterministic domains, predominantly to traditional search domains from artificial intelligence. Later in this chapter, we discuss the performance of LRTA*-type real-time search methods in these domains. They include:

Sliding-Tile Puzzles: Examples include [Korf, 1987, Korf, 1988, Korf, 1990, Russell and Wefald, 1991, Knight, 1993, Korf, 1993, Ishida, 1995]. Sliding-tile puzzles consist of a square frame that is completely filled with square tiles, except for one tile that is missing. An agent has to achieve a specified configuration of the tiles. It can repeatedly slide a tile that is adjacent to the empty square into the empty square. Two popular variants of sliding-tile puzzles are the eight puzzle with the “American” goal state and the eight puzzle with the “European” goal state (Figure 2.7). Calling these two goal states “American” and “European” is nonstandard, but makes it easy for us to refer to them. Both variants of the eight puzzle have 181,440 states from which the goal state can be reached, but the American goal state cannot be reached from the European goal state and vice versa.

Grid-Worlds (Figure 2.8[3]): Examples include [Barto *et al.*, 1989, Korf, 1990, Pirzadeh and Snyder, 1990, Sutton, 1990, Ishida and Korf, 1991, Ishida, 1992, Pemberton and Korf, 1992, Thrun, 1992b, Peng and Williams, 1992, Singh, 1992, Whitehead, 1992, Matsubara and Ishida, 1994, Stentz, 1995, Ishida, 1995]. Grid-worlds are popular abstractions of robot navigation domains. They discretize two-dimensional space into square cells. An agent has to move to a specified cell. It can repeatedly move to one of the four neighboring squares of its current square as long as it stays on the grid and the target square does not contain an obstacle.

LRTA*-type real-time search methods have also been applied to other domains, for example blocks-worlds. Examples include [Knight, 1993]. In all of these domains, LRTA*-type real-time search methods have proved to be efficient alternatives to traditional search methods, such as the A* method [Nilsson, 1971]. For example,

[3] In the figure, we abbreviate two edges $s \rightarrow s'$ and $s \leftarrow s'$ with one edge $s \leftrightarrow s'$.

they are among the few search methods that are able to find (possibly suboptimal) plans for the twenty-four puzzle, a sliding-tile puzzle with more than 7×10^{24} states [Korf, 1993]. The reason why LRTA*-type real-time search methods are efficient in deterministic domains is because they trade-off minimizing planning cost and minimizing plan execution cost. In the following, we extend LRTA* to nondeterministic domains.

2.3 Min-Max LRTA*

Traditional search domains from artificial intelligence are deterministic, but many domains from robotics, control, and scheduling are not. We study an example in Section 2.4. The *Min-Max Learning Real-Time A* Method* (Min-Max LRTA*) [Koenig and Simmons, 1995b] uses minimax search to extend LRTA* to nondeterministic domains. Both Min-Max LRTA* and LRTA* behave identically in deterministic domains. Min-Max LRTA* also generalizes the IG method [Genesereth and Nourbakhsh, 1993] (Section 2.4.4.1) and the $\hat{\text{Q}}$-Learning method [Heger, 1996] (Section 2.5.6.4). In the following, we first give an overview of Min-Max LRTA* and describe its assumptions, then describe Min-Max LRTA* itself, and finally analyze its performance.

2.3.1 Overview and Assumptions

Acting in nondeterministic domains can be viewed as a two-player game: the search method chooses an action from the actions available in the current state. The action determines the possible successor states from which a fictitious agent, called *nature*, chooses one. Acting in deterministic domains is then simply a special case where every action uniquely determines the successor state. Min-Max LRTA* uses *minimax search* to solve search tasks in nondeterministic domains, a worst-case search method that assumes that nature is vicious and always chooses the worst possible successor state. Minimax search decreases the goal distance with every action execution as much as possible under the assumption that nature increases it as much as possible. It is commonly used in game-playing and robotics (Section 4.4.1).

An advantage of minimax search is that it does not depend on assumptions about the behavior of nature. If it can reach a goal state for the most vicious behavior of nature, it also reaches a goal state if nature uses a different and therefore less vicious behavior. If one could make reliable assumptions about the behavior of nature, one would not necessarily use minimax search. For example, if one knows that nature chooses successor states randomly according to given probabilities, then one can use Trial-Based Real-Time Dynamic Programming [Barto *et al.*, 1995], another generalization of LRTA*. Unfortunately, it is often not possible to make reliable assumptions about the behavior of nature. Consider the following example: Agents with limited capabilities, such as actuator uncertainty, imperfect sensors, imprecise world models, or limited knowledge, can perceive deterministic domains to be nondeterministic. Such an agent might model a deterministic domain with only low granularity, for example to keep the model small and thus decrease the memory, running time, and training-data requirements of search and learning methods that operate on them. In this case, a state of the coarse domain really corresponds to a set of system states. The agent can predict the successor state that results from an action execution with certainty if it knows which system state it executed the action in. However, the agent might not be able to predict the exact successor state if all it knows is the coarse state it executed the action in. In this case, the agent might only be able to predict the set of possible successor states, and many planning approaches, including approaches that assume that nature chooses successor states randomly according to given transition probabilities, suffer from the perceptual aliasing problem [Chrisman, 1992]. The main reason why we use an approach based on minimax search is that minimax search does not suffer from this problem. In Section 2.4, we discuss a another example with perceptual aliasing and use Min-Max LRTA* to solve it. A side benefit of minimax search is that methods that are based on worst-case action outcomes are easier to analyze than methods that are based on average action outcomes.

We use the following notation to explain the assumptions behind Min-Max LRTA*: The *minimax distance* $d(s, s') \in [0, \infty]$ from state $s \in S$ to state $s' \in S$ is the smallest number of action executions with which state

s' can be reached from state s, even for the most vicious behavior of nature. The minimax distances can be calculated by solving the following set of equations:

$$d(s, s') = \begin{cases} 0 & \text{if } s = s' \\ 1 + \min_{a \in A(s)} \max_{s'' \in succ(s,a)} d(s'', s') & \text{otherwise} \end{cases} \qquad \text{for all } s, s' \in S$$

with $0 \leq d(s, s') \leq \infty$ for all $s, s' \in S$. The *minimax goal distance* $gd(s)$ of state $s \in S$ is $gd(s) := \min_{s' \in G} d(s, s')$. The *maximal minimax goal distance* is $d := \max_{s \in S} gd(s)$. In deterministic domains, the definitions of minimax distances and minimax goal distances simplify to the traditional definitions of distances and goal distances, respectively.

Min-Max LRTA* interleaves minimax searches with plan execution and thus is not guaranteed to solve all search tasks. A minimax search limits the solvable search tasks because it is overly pessimistic. It is only guaranteed to solve search tasks for which the minimax goal distance of the start state is finite. Interleaving planning and plan execution limits the solvable search tasks because it executes actions before their complete consequences are known. Thus, even if the minimax goal distance of the start state is finite, it is possible that Min-Max LRTA* accidentally executes actions that lead to a state with infinite minimax goal distance, such as actions that "blow up the world," at which point the search task becomes unsolvable for the agent. Robots, for instance, can fall down staircases if they are not careful.

We avoid both problems by assuming that the domains are safely explorable. Domains are *safely explorable* if the minimax goal distances of all states are finite, that is, $d < \infty$. (A finite d implies that $d \leq n - 1$, where n is the number of states). To be precise: First, all states of the domain that cannot possibly be reached from the start state, or can be reached from the start state only by passing through a goal state can be deleted. The minimax goal distances of all remaining states have to be finite.[4] Safely explorable domains guarantee that Min-Max LRTA* is able to reach a goal state no matter which actions it has executed in the past and what the behavior of nature is. Sliding-tile puzzles and grid-worlds are deterministic domains that are safely explorable (provided that a solution exists) because they are undirected. Although the precision of robot actuators is limited, many nondeterministic domains from robotics are safely explorable, including many manipulation and assembly domains. Consequently, robotics often uses minimax methods in either real-world or prototypical test domains, including the "peg in the hole" domain.

We also assume throughout this chapter that every action execution in a nongoal state necessarily results in a state change. In other words, action executions cannot leave the current state unchanged. This assumption is not necessary but simplifies the complexity results. Section 2.6 describes in more detail which results are affected when the assumption is dropped. Sliding-tile puzzles and grid-worlds satisfy the assumption, and all domains can be modified to satisfy it by removing all actions whose execution could leave the current state unchanged. These actions can safely be deleted because there always exists a minimax solution that does not use them if there exists a minimax solution at all. For example, the optimal minimax solution does not use them.

2.3.2 Description of the Method

Min-Max LRTA* with look-ahead one is shown in Figure 2.9. The only difference between Min-Max LRTA* with look-ahead one and LRTA* with look-ahead one (Figure 2.6) is the following: LRTA* operates in deterministic domains. Its u-values approximate the negative goal distances of the states and it uses $u(succ(s, a))$ in the action-selection and value-update steps. This expression is the u-value of the state that results from the execution of action a in state s. Min-Max LRTA*, on the other hand, operates in nondeterministic domains. Its u-values approximate the negative minimax goal distances of the states and it

[4] This requirement can be relaxed in known domains by increasing the look-ahead of Min-Max LRTA* to ensure that the action that it is about to execute does not make it impossible to achieve the goal. For this purpose, one can use the techniques described in [Nourbakhsh, 1996].

Initially, the u-values $u(s)$ are approximations of the negative minimax goal distances (measured in action executions) for all $s \in S$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. $a :=$ one-of $\arg\max_{a \in A(s)} \min_{s' \in succ(s,a)} u(s')$.
4. $u(s) := \min(u(s), -1 + \min_{s' \in succ(s,a)} u(s'))$.
5. Execute action a, that is, change the current state to a state in $succ(s, a)$ (according to the behavior of nature).
6. $s :=$ the current state.
7. Go to 2.

Figure 2.9: Min-Max LRTA* with Look-Ahead One

Initially, the u-values $u(s)$ are approximations of the negative minimax goal distances (measured in action executions) for all $s \in S$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. Generate a local search space S_{lss} with $s \in S_{lss}$ and $S_{lss} \cap G = \emptyset$.
4. Update $u(s)$ for all $s \in S_{lss}$ (Figure 2.11).
5. $a :=$ one-of $\arg\max_{a \in A(s)} \min_{s' \in succ(s,a)} u(s')$.
6. Execute action a, that is, change the current state to a state in $succ(s, a)$ (according to the behavior of nature).
7. $s :=$ the current state.
8. (If $s \in S_{lss}$, then go to 5.)
9. Go to 2.

Figure 2.10: Min-Max LRTA* (with Arbitrary Look-Ahead)

The minimax-search method uses the temporary variables $u'(s)$ for all $s \in S_{lss}$.

1. For all $s \in S_{lss}$: $u'(s) := u(s)$ and $u(s) := -\infty$.
2. If $u(s) > -\infty$ for all $s \in S_{lss}$, then return.
3. $s' :=$ one-of $\arg\max_{s \in S_{lss} | u(s) = -\infty} \min(u'(s), -1 + \max_{a \in A(s)} \min_{s'' \in succ(s,a)} u(s''))$.
4. If $\min(u'(s'), -1 + \max_{a \in A(s')} \min_{s'' \in succ(s',a)} u(s'')) = -\infty$, then return.
5. $u(s') := \min(u'(s'), -1 + \max_{a \in A(s')} \min_{s'' \in succ(s',a)} u(s''))$.
6. Go to 2.

Figure 2.11: Minimax-Search Method

uses $\min_{s' \in succ(s,a)} u(s')$ in the action-selection and value-update steps. This expression is the worst u-value of all states that can result from the execution of action a in state s.

Min-Max LRTA* with arbitrary look-ahead (short: Min-Max LRTA*) is shown in Figure 2.10. It generalizes Min-Max LRTA* with look-ahead one to arbitrary look-aheads by searching an arbitrary part of the domain, the *local search space* $S_{lss} \subseteq S$, before deciding which action to execute in the current state s. While we require only that $s \in S_{lss}$ and $S_{lss} \cap G = \emptyset$, in practice we construct S_{lss} by searching forward from s.[5]

Min-Max LRTA* allows for fine-grained control over how much planning to do between plan executions. Thus, it is an any-time contract algorithm [Russell and Zilberstein, 1991]. For example, Min-Max LRTA* with $S_{lss} = S \setminus G = S \cap \overline{G}$ (that is, sufficiently large look-ahead) performs a complete minimax search without interleaving planning and plan execution, which is slow but produces plans that are worst-case optimal. On the other hand, Min-Max LRTA* with $S_{lss} = \{s\}$ performs almost no planning between plan executions. It behaves identically to Min-Max LRTA* with look-ahead one (provided that, as we assume throughout this chapter, every action execution in a nongoal state necessarily results in a state change). This is so, because the value-update step of Min-Max LRTA* with $S_{lss} = \{s\}$ updates only one u-value, namely

[5]Thus, Min-Max LRTA* does not have to be used as a real-time search method. It could, for example, also be used in conjunction with the DYNA framework [Sutton, 1991], which allows search methods to update the u-values of arbitrary states and has been used to study in which order to update the u-values [Peng and Williams, 1992, Moore and Atkeson, 1993].

the u-value of the current state, and the action-selection step does not depend on this u-value. This makes the value-update and action-selection steps interchangeable. Furthermore, the minimax search of Min-Max LRTA* with $S_{lss} = \{s\}$ sets the u-value of the current state temporarily to minus infinity but never uses this value. Thus, this assignment can be skipped. After these two changes, that do not change the behavior of Min-Max LRTA* with $S_{lss} = \{s\}$, it has been transformed to Min-Max LRTA* with look-ahead one.

Small look-aheads can have advantages for sufficiently fast moving agents, especially if the heuristic knowledge guides the search sufficiently: First, small look-aheads minimize the time required to determine which plan to execute next. This prevents the agents from being idle if planning and plan execution are overlapped. Second, small look-aheads have the potential to minimize the total planning time, which in turn can also minimize the sum of total planning and plan-execution time if planning and plan execution are interleaved. Korf [Korf, 1990], for example, studies which look-ahead minimizes the planning time of the Real-Time A* method (RTA*) [Korf, 1987], a variant of LRTA*. He reports that a look-ahead of one is optimal for the eight puzzle and a look-ahead of two is optimal for the fifteen and twenty-four puzzles if the Manhattan distance is used to initialize the u-values. Knight [Knight, 1993] reports similar results. Notice that both advantages of small look-aheads apply only to agents that can execute plans with a similar speed as they can generate them. Thus, they apply only to sufficiently fast moving agents (such as simulated agents) and larger look-aheads are better for more slowly moving agents (such as robots).

Line 4 in Figure 2.10 implements the value-update step of Min-Max LRTA*, that is, the minimax search in the local search space. Min-Max LRTA* could represent the local search space as a minimax tree. Minimax trees can be searched with traditional minimax-search methods, including variants of alpha-beta search. However, this has the disadvantage that the number of states contained in the tree and thus both the memory requirements and the search effort can be exponential in the depth of the tree (the look-ahead of Min-Max LRTA*). Since the number of *different* states contained in the tree often grows only polynomially in the depth of the tree, Min-Max LRTA* represents the local search space more compactly as a graph that contains every state at most once. This requires a more sophisticated minimax-search method because there can now be paths of different lengths between any two states in the graph. Figure 2.11 shows our minimax search method. It updates all states in the local search space in the order of their decreasing new u-values. This ensures that the u-value of each state is updated only once. More general dynamic programming methods from Markov game theory, such as the ones described by Littman [Littman, 1996], could also be used to perform the minimax search.

The following theorem proves the correctness of the u-values after the minimax search. The time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update step of Min-Max LRTA* (Line 4 in Figure 2.10).

Theorem 1 *For all times $t = 0, 1, 2, \ldots$ (until termination), assume that $-\infty \leq u^t(s) < \infty$ for all $s \in S$. Define*

$$u_{opt}^{t+1}(s) = \begin{cases} \min(u^t(s), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u_{opt}^{t+1}(s')) & \text{if } s \in S_{lss}^t \\ u^t(s) & \text{otherwise} \end{cases} \qquad \text{for all } s \in S$$

with $-\infty \leq u_{opt}^{t+1}(s) < \infty$ for all $s \in S$. Then, the minimax search terminates with $u^{t+1}(s) = u_{opt}^{t+1}(s)$ for all $s \in S$.

Proof: We first prove that the minimax search terminates and then that it determines the correct u-values.

The minimax search terminates: It terminates if the u-values of all states in the local search space are larger than minus infinity (Line 2). Otherwise, it either changes the u-value of another state from minus infinity to some other value (Line 5) or, if that is not possible, terminates (Line 4). Thus, it terminates eventually.

The minimax search determines the correct u-values: Consider the $(t+1)$st execution of the minimax-search method for any time $t = 0, 1, 2, \ldots$ (until termination). The u-values $u^{t+1}(s)$ are correct for all states s

that do not belong to the local search space S^t_{lss} because they do not change during the minimax search and thus $u^{t+1}(s) = u^t(s) = u^{t+1}_{opt}(s)$. To show that the u-values $u^{t+1}(s)$ are also correct for all states of the local search space consider any time during the minimax search. Then, $u(s) = u^{t+1}_{opt}(s)$ for all $s \in S^t_{lss}$ with $u(s) > -\infty$, as we prove below. It follows that, after the minimax search, the u-values are correct for all states of the local search space whose u-values are larger than minus infinity. To show that the u-values are also correct for all states of the local search space whose u-values equal minus infinity, suppose that this is not the case. Then, the minimax search terminates on Line 4 and there are states in the local search space whose u-values are, incorrectly, minus infinity. Among these states, consider the state s with the maximal value $u^{t+1}_{opt}(s)$, any action $a :=$ one-of $\arg\max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}_{opt}(s')$, and any state $s' \in succ(s,a)$. Since $u^{t+1}_{opt}(s') > u^{t+1}_{opt}(s) > -\infty$, it holds that either $s' \in S \setminus S^t_{lss}$, which implies $u(s') = u^{t+1}_{opt}(s') > -\infty$, or $s' \in S^t_{lss}$ with $u(s') > -\infty$ according to our choice of s. Also, $u^t(s) > -\infty$ since $u^{t+1}_{opt}(s) > -\infty$. Then, however, the minimax search could not have terminated on Line 4 because $\min(u^t(s), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u(s')) > -\infty$, which is a contradiction. Thus, the minimax search determines the correct u-values.

We prove by induction that, at any time during the $(t+1)$st execution of the minimax-search method, $u(s) = u^{t+1}_{opt}(s)$ for all $s \in S^t_{lss}$ with $u(s) > -\infty$. This holds initially since $u(s) = -\infty$ for all $s \in S^t_{lss}$. Now suppose that the induction hypothesis holds when an $\overline{s} \in S^t_{lss}$ is chosen on Line 3 and let $u(s)$ denote the u-values at this point in time. Suppose further that the subsequent assignment on Line 5 results in $u_{new}(\overline{s}) \neq u^{t+1}_{opt}(\overline{s})$. In general, $u(s') \leq u^{t+1}_{opt}(s')$ for all $s' \in S^t_{lss}$ since either $u(s') = -\infty$ or $u(s') = u^{t+1}_{opt}(s')$ according to the induction hypothesis. Then,

$$\begin{aligned} u_{new}(\overline{s}) &= \min(u^t(\overline{s}), -1 + \max_{a \in A(\overline{s})} \min_{s' \in succ(\overline{s},a)} u(s')) \\ &\leq \min(u^t(\overline{s}), -1 + \max_{a \in A(\overline{s})} \min_{s' \in succ(\overline{s},a)} u^{t+1}_{opt}(s')) \\ &= u^{t+1}_{opt}(\overline{s}). \end{aligned}$$

Since $u_{new}(\overline{s}) \overset{\text{Assumption}}{\neq} u^{t+1}_{opt}(\overline{s})$, it holds that $u_{new}(\overline{s}) = \min(u^t(\overline{s}), -1 + \max_{a \in A(\overline{s})} \min_{s' \in succ(\overline{s},a)} u(s')) < u^{t+1}_{opt}(\overline{s})$. Now consider any state $s :=$ one-of $\arg\max_{s \in S^t_{lss} | u(s) = -\infty} u^{t+1}_{opt}(s)$, any action $a :=$ one-of $\arg\max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}_{opt}(s')$, and any state $s' \in succ(s,a)$. Since $u^{t+1}_{opt}(s') > u^{t+1}_{opt}(s) \geq u^{t+1}_{opt}(\overline{s}) > u_{new}(\overline{s}) > -\infty$, it holds that either $s' \in S \setminus S^t_{lss}$ or $s' \in S^t_{lss}$ with $u(s') > -\infty$ according to our choice of s. In either case, $u^{t+1}_{opt}(s') = u(s')$ according to the definition of $u^{t+1}_{opt}(s')$ or the induction hypothesis. Then,

$$\begin{aligned} &\min(u^t(\overline{s}), -1 + \max_{a \in A(\overline{s})} \min_{s' \in succ(\overline{s},a)} u(s')) \\ &< u^{t+1}_{opt}(\overline{s}) \\ &\leq u^{t+1}_{opt}(s) \\ &= \min(u^t(s), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}_{opt}(s')) \\ &= \min(u^t(s), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u(s')). \end{aligned}$$

But then the minimax search could not have chosen $\overline{s}$. This is a contradiction. Thus, $u_{new}(\overline{s}) = u^{t+1}_{opt}(\overline{s})$. After $\overline{s}$ has been assigned this value on Line 5, it cannot be assigned another value later, since $u_{new}(\overline{s}) > -\infty$. ■

Min-Max LRTA* with Line 8 is a special case of Min-Max LRTA* without Line 8: After one has run the minimax-search method (Figure 2.11) on some local search space, the u-values do not change if one runs it

again on the same local search space or a subset thereof. Whenever Min-Max LRTA* with Line 8 jumps to Line 5, the new current state is still part of the local search space and thus not a goal state. Consequently, Min-Max LRTA* can skip the termination-checking step (Line 2). Min-Max LRTA* could now generate a subset of the previous local search space that contains the new current state (Line 3). Since the minimax search in this local search space does not change the u-values, Min-Max LRTA* can, in this case, also skip the value-update step (Line 4). In the experiments, we use Min-Max LRTA* with Line 8, because it utilizes more information of the searches in the local search spaces.

2.3.3 Performance Analysis

Min-Max LRTA* is similar to game-playing programs that use minimax search, except that Min-Max LRTA* modifies its evaluation function during the search. This is interesting since LRTA* was originally inspired by game-playing programs. Both game-playing programs and Min-Max LRTA* have to search large nondeterministic domains where the behavior of the opponent is unknown. Both make the search task tractable by combining real-time search with minimax search. However, their planning objectives differ. Game-playing programs distinguish terminal states of different quality (wins, losses, and draws). Their objective is to get to a winning terminal state. How long this takes is usually unimportant. Min-Max LRTA*, on the other hand, has only winning terminal states (the goal states). Its objective is to get to a terminal state fast. Thus, we have to analyze how quickly Min-Max LRTA* reaches a goal state. We have already argued in Section 2.1 that such an analysis is important. In this section, we therefore study the performance of Min-Max LRTA* and how it depends on the number of states of a domain, its maximal and average minimax goal distance, and the heuristic knowledge that one has about the domain (that is, the initial u-values).

The *performance* of Min-Max LRTA* is the number of actions that it executes until it reaches a goal state. Thus, its performance coincides with its solution quality. This is motived by the fact that, for sufficiently fast moving agents that solve one-shot planning tasks, the sum of planning and plan execution time is determined by the planning time, which is roughly proportional to the number of action executions if Min-Max LRTA* performs only a constant amount of computations between action executions. (Thus, the performance measure cannot be used to choose how much planning to do between plan executions but it can be used to compare two LRTA*-type real-time search methods that both perform about the same amount of planning between plan executions. We make use of this property in Section 2.3.3.6.) For sufficiently slowly moving agents that solve one-shot planning tasks, on the other hand, the sum of planning and plan-execution time is determined by the plan-execution time, which is roughly proportional to the number of action executions if every action can be executed in about the same amount of time.

The *complexity* of Min-Max LRTA* is its worst-case performance (often expressed in big-O notation) over all possible topologies of domains with a given property (such as safely explorable domains), all possible start and goal states, all tie-breaking rules among indistinguishable actions, and all strategies of nature. We are interested in determining how the complexity of Min-Max LRTA* scales as the tasks get larger. We measure the size of the tasks as nonnegative integers and use five different measures: First, $x = d$, the maximal minimax goal distance; second, $x = n$, the number of states; third, $x = e$, the number of actions; fourth, $x = nd$, the product of the number of states and the maximal minimax goal distance; and fifth, $x = ed$, the product of the number of edges and the maximal minimax goal distance. In the following, we formalize how we characterize the complexity of Min-Max LRTA* as a function of the task size. An upper complexity bound (a function of x) has to hold for every x that is larger than some constant, that is, basically every domain. Since this thesis is about artificial intelligence and not theoretical computer science, we are mostly interested in the general trend (but not outliers) for the lower complexity bound. A lower complexity bound (also a function of x) thus has to hold only for infinitely many different x. Furthermore, we vary only x. If x is a product, we do not vary both of its factors independently. This is sufficient for our purposes.

Let f be a positive function of x. We say that the complexity of Min-Max LRTA* is at most $f(x)$ action executions if Min-Max LRTA* needs at most $f(x)$ action executions for all domains whose x is larger than some constant. We say that the complexity of Min-Max LRTA* is at least $f(x)$ action executions if Min-Max

LRTA* needs at least $f(x)$ action executions for infinitely many domains with different x. We say that the complexity of Min-Max LRTA* is tight at $f(x)$ action executions if it is both at most $f(x)$ action executions and at least $f(x)$ action executions. Similarly, we say that the complexity of Min-Max LRTA* is at most $O(f(x))$ action executions if there is a positive constant c such that Min-Max LRTA* needs at most $c\,f(x)$ action executions for all domains whose x is larger than some constant. We say that the complexity of Min-Max LRTA* is at least $O(f(x))$ action executions if there is a positive constant c such that Min-Max LRTA* needs least $c\,f(x)$ action executions for infinitely many domains with different x. Finally, we say that the complexity of Min-Max LRTA* is tight at $O(f(x))$ action executions if it is both at most $O(f(x))$ action executions and at least $O(f(x))$ action executions.

Experimental researchers sometimes consider a complexity analysis of search methods to be unimportant, because they are more interested in their average-case performance than their worst-case performance. The reason why upper complexity bounds are interesting is because they provide performance guarantees (the complexity is an upper bound on the average-case performance). A proof that Min-Max LRTA* has a small complexity, for example, prevents unpleasant surprises since we are guaranteed that there are no domains beyond a certain size in which its actual performance deteriorates completely. In the following, we analyze the complexity of Min-Max LRTA*.

2.3.3.1 Properties of U-Values

In this section, we define the properties of u-values needed for the complexity results.

Definition 1 *U-values are* uniformly initialized *with x (or, synonymously, x-initialized) if and only if initially*

$$u(s) = \begin{cases} 0 & \text{if } s \in G \\ x & \text{otherwise} \end{cases} \qquad \text{for all } s \in S.$$

Definition 2 *U-values are consistent if and only if, for all $s \in G$, $u(s) = 0$, and, for all $s \in S \setminus G$, $-1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u(s') \leq u(s) \leq 0$. U-values are admissible if and only if, for all $s \in S$, $-gd(s) \leq u(s) \leq 0$.*

Consistency (or, equivalently, monotonicity) means that the triangle inequality holds and admissibility means that the negative u-values never overestimate the minimax goal distances. In deterministic domains, these definitions reduce to the traditional definitions of consistent and admissible heuristic values for A* search, except that we use negative heuristic values instead of positive ones. Notice that zero-initialized u-values are consistent, and consistent u-values are admissible.

For deterministic domains, consistent (or admissible) initial u-values can be obtained as follows: Assume that heuristic values $h(s)$ are known that are consistent (or admissible) for A* search. Then, the u-values $u(s) = -h(s)$ are consistent (or admissible). Consistent (or admissible) heuristic values are known for many deterministic domains, including sliding-tile puzzles and grid-worlds. For nondeterministic domains, consistent (or admissible) u-values can be obtained as follows: One can assume that nature decides in advance which successor state $g(s,a) \in succ(s,a)$ to choose whenever action $a \in A(s)$ is executed in state $s \in S$; all possible states are fine. If nature really behaved this way, then the domain would effectively be deterministic. U-values that are consistent (or admissible) for this deterministic domain are consistent (or admissible) for the nondeterministic domain as well, regardless of the actual behavior of nature. This is so because additional action outcomes allow a vicious nature to cause more harm: U-values that are consistent in the deterministic domain are also consistent in the nondeterministic domain because, for each state, the maximal u-value that the agent is guaranteed to reach from this state after one action execution in the nondeterministic domain is at most as large as the maximal u-value that the agent is guaranteed to reach from this state after one action execution in the deterministic domain. U-values that are admissible in the deterministic domain are

also admissible in the nondeterministic domain because, for each state, its minimax goal distance in the nondeterministic domain is at least as large as its goal distance in the deterministic domain. In both cases, how informed the obtained u-values in the nondeterministic domain are depends on how close the assumed behavior of nature is to its most vicious behavior.

Consistent or admissible u-values have the following important property.

Theorem 2 *Consistent (or admissible) initial u-values remain consistent (or admissible) after every action execution of Min-Max LRTA* and are monotonically nonincreasing.*

Proof: In the following, the time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update step of Min-Max LRTA* (Line 4 in Figure 2.10). Thus, $u^t(s)$ denotes the u-values before the $(t+1)$st execution of the minimax search and $u^{t+1}(s)$ the u-values after its termination. The u-values are monotonically nonincreasing because either they do not change or, according to Line 5 of the minimax-search method, $u^{t+1}(s) = \min(u^t(s), \ldots) \leq u^t(s)$. We distinguish two cases:

- The u-values $u^t(s)$ are consistent.

  For all $s \in G$, $u^{t+1}(s) = u^t(s) \overset{\text{Consistency}}{=} 0$. Furthermore, for all $s \in S \setminus (S^t_{lss} \cup G)$, $-1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s') \overset{\text{Monotonicity}}{\leq} -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^t(s') \overset{\text{Consistency}}{\leq} u^t(s) = u^{t+1}(s) = u^t(s) \overset{\text{Consistency}}{\leq} 0$. Finally, for all $s \in S^t_{lss}$ (which implies $s \in S \setminus G$ per definition of S^t_{lss})

$$
\begin{aligned}
& -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s') \\
&= \min(-1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s'), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s')) \\
&\overset{\text{Monotonicity}}{\leq} \min(-1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t}(s'), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s')) \\
&\overset{\text{Consistency}}{\leq} \min(u^t(s), -1 + \max_{a \in A(s)} \min_{s' \in succ(s,a)} u^{t+1}(s')) \\
&\overset{\text{Theorem 1}}{=} u^{t+1}(s) \overset{\text{Monotonicity}}{\leq} u^t(s) \overset{\text{Consistency}}{\leq} 0. \qquad (2.1)
\end{aligned}
$$

  Therefore, the u-values $u^{t+1}(s)$ are consistent as well.

- The u-values $u^t(s)$ are admissible.

  For all $s \in S \setminus S^t_{lss}$, $-gd(s) \overset{\text{Admissibility}}{\leq} u^t(s) = u^{t+1}(s) = u^t(s) \overset{\text{Admissibility}}{\leq} 0$. Furthermore, for all $s \in S^t_{lss}$, $-gd(s) \leq u^{t+1}(s)$ as we show below. It follows that, for all $s \in S^t_{lss}$, $-gd(s) \leq u^{t+1}(s) \overset{\text{Monotonicity}}{\leq} u^t(s) \overset{\text{Admissibility}}{\leq} 0$. Therefore, the u-values $u^{t+1}(s)$ are admissible as well.

  We prove by induction that, for all $s \in S^t_{lss}$, $-gd(s) \leq u^{t+1}(s)$. Consider an ordering s_i for $i = 1, 2, \ldots, |S^t_{lss}|$ of all $s \in S^t_{lss}$ according to their increasing minimax goal distance. We show by induction on i that $-gd(s_i) \leq u^{t+1}(s_i)$. We consider two cases: First, $gd(s_i) = \infty$. Then, it holds trivially that $-gd(s_i) \leq u^{t+1}(s_i)$. Second, $gd(s_i) < \infty$. Then, $s_i \in S^t_{lss}$ implies $s_i \in S \setminus G$ per definition of S^t_{lss}. Thus, $gd(s_i) = 1 + \min_{a \in A(s_i)} \max_{s' \in succ(s_i,a)} gd(s')$. Let $a' :=$ one-of $\arg\min_{a \in A(s_i)} \max_{s' \in succ(s_i,a)} gd(s')$. Then, $gd(s_i) = 1 + \max_{s' \in succ(s_i,a')} gd(s')$ and thus $gd(s') < gd(s_i)$ for all $s' \in succ(s_i, a')$. In general, $-gd(s') \leq u^{t+1}(s')$ for all $s' \in succ(s_i, a')$ since either $s' \in S \setminus S^t_{lss}$ (see above) or $s' = s_j$ with $j < i$ per induction hypothesis. Then,

$$
-gd(s_i) = \min(-gd(s_i), -gd(s_i))
$$

$$
\begin{aligned}
&= \min(-gd(s_i), -(1 + \max_{s' \in succ(s_i,a')} gd(s'))) \\
&= \min(-gd(s_i), -1 + \min_{s' \in succ(s_i,a')} [-gd(s')]) \\
&\leq \min(-gd(s_i), -1 + \min_{s' \in succ(s_i,a')} u^{t+1}(s')) \\
&\leq \min(-gd(s_i), -1 + \max_{a \in A(s_i)} \min_{s' \in succ(s_i,a)} u^{t+1}(s')) \\
&\overset{\text{Admissibility}}{\leq} \min(u^t(s_i), -1 + \max_{a \in A(s_i)} \min_{s' \in succ(s_i,a)} u^{t+1}(s')) \\
&\overset{\text{Theorem 1}}{=} u^{t+1}(s_i). \ \blacksquare
\end{aligned}
$$

This theorem generalizes a theorem in [Ishida and Korf, 1991], that they state only for initially admissible u-values of LRTA* in deterministic domains. Their theorem is about the Moving-Target Search method (MTS) [Ishida and Korf, 1991], but MTS behaves identically to LRTA* if the target does not move.

Theorem 2 enables us to simplify the value-update step of Min-Max LRTA* if its initial u-values are consistent. Consider the $(t+1)$st value-update step of Min-Max LRTA* (Line 4 in Figure 2.10). Let $u^t(s)$ and $u^{t+1}(s)$ refer to the u-values immediately before and after, respectively, the value-update step. Let $u(s)$ and $u'(s)$ denote the u-values and u'-values, respectively, of the minimax-search method at any point in time directly before Line 5 in Figure 2.11 is executed and let s' be any state in the local search space with $u(s') = \infty$. Then, $u(s'') \leq u^{t+1}(s'')$ for all $s'' \in S$ since either $u(s'') = -\infty$ or $u(s'') = u^{t+1}(s'')$. Furthermore, u-values that are initially consistent remain consistent according to Theorem 2 and thus the u-values $u^t(s)$ are consistent. Put together, $-1 + \max_{a \in A(s')} \min_{s'' \in succ(s',a)} u(s'') \leq -1 + \max_{a \in A(s')} \min_{s'' \in succ(s',a)} u^{t+1}(s'') \overset{\text{Formula 2.1}}{\leq} u^t(s') = u'(s')$. Therefore, Line 5 can be simplified to $u(s') := -1 + \max_{a \in A(s')} \min_{s'' \in succ(s',a)} u(s'')$ without changing the u-value assigned to s'. (Lines 3 and 4 can be simplified in the same way.) LRTA* was originally stated with this value-update step only [Korf, 1990]. We refer to it as the *simplified value-update step* of Min-Max LRTA*.

2.3.3.2 Upper Complexity Bound

In this section, we provide an upper bound on the complexity of Min-Max LRTA*. The upper complexity bound is smaller the fewer states the domain has, the smaller the maximal minimax goal distance is, and the better the initial heuristic knowledge is (that is, the more negative the initial u-values are).

Our complexity analysis is centered around the invariant from Theorem 3. This invariant shows that the number of executed actions is always bounded from above by an expression that depends only on the initial and current u-values. The time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update step of Min-Max LRTA* (Line 4 in Figure 2.10).

Theorem 3 *For all times $t = 0, 1, 2, \ldots$ (until termination)*

$$t \leq \sum_{s \in S} \left[u^0(s) - u^t(s)\right] - (u^0(s^0) - u^t(s^t))$$

for Min-Max LRTA with initially admissible u-values in safely explorable domains (regardless of the behavior of nature).*[6]

[6] Sums have a higher precedence than other operators. For example, $\sum_i x + y = \sum_i [x] + y = (\sum_i [x]) + y \neq (\sum_i [x + y])$.

Proof by induction: The u-values are admissible at time t according to Theorem 2. Thus, they are finite, since the domain is safely explorable. For $t = 0$, the inequality reduces to $t \leq 0$, which is true. Now assume that the theorem holds at time t. The left-hand side of the inequality increases by one between time t and $t + 1$. The right-hand side of the inequality increases by

$$
\begin{aligned}
& \sum_{s \in S \setminus \{s^t\}} u^t(s) - \sum_{s \in S \setminus \{s^{t+1}\}} u^{t+1}(s) \\
= & \sum_{s \in S \setminus \{s^t, s^{t+1}\}} [u^t(s) - u^{t+1}(s)] + u^t(s^{t+1}) - u^{t+1}(s^t) \\
\overset{\text{Theorem 1}}{=} & \sum_{s \in S \setminus \{s^t, s^{t+1}\}} [u^t(s) - u^{t+1}(s)] + u^t(s^{t+1}) - \min(u^t(s^t), -1 + \max_{a \in A(s^t)} \min_{s' \in succ(s^t, a)} u^{t+1}(s')) \\
\geq & \sum_{s \in S \setminus \{s^t, s^{t+1}\}} [u^t(s) - u^{t+1}(s)] + u^t(s^{t+1}) + 1 - \max_{a \in A(s^t)} \min_{s' \in succ(s^t, a)} u^{t+1}(s') \\
\geq & \sum_{s \in S \setminus \{s^t, s^{t+1}\}} [u^t(s) - u^{t+1}(s)] + u^t(s^{t+1}) + 1 - u^{t+1}(s^{t+1}) \\
= & \sum_{s \in S \setminus \{s^t\}} [u^t(s) - u^{t+1}(s)] + 1 \\
\overset{\text{Theorem 2}}{\geq} & 1. \ \blacksquare
\end{aligned}
$$

According to the proof of Theorem 3, the potential $\sum_{s \in S \setminus \{s^t\}} u^t(s)$ decreases with every action execution by at least one. The potential is bounded from below since the u-values are bounded from below by the negative minimax goal distances. This follows directly from the definition of admissible u-values and the fact that they remain admissible after every action execution (Theorem 2). Theorem 4 uses this fact to derive an upper bound on the number of action executions.

Theorem 4 *Min-Max LRTA* with initially admissible u-values reaches a goal state in safely explorable domains after at most* $\sum_{s \in S}[u^0(s) + gd(s)] - u^0(s^0)$ *action executions (regardless of the behavior of nature).*

Proof:

$$
\begin{aligned}
t & \overset{\text{Theorem 3}}{\leq} \sum_{s \in S} \left[u^0(s) - u^t(s)\right] - (u^0(s^0) - u^t(s^t)) \\
& \overset{\text{Admissibility}}{\leq} \sum_{s \in S} \left[u^0(s) + gd(s)\right] - u^0(s^0). \ \blacksquare
\end{aligned}
$$

Now, let $\overline{d}$ denote the average minimax goal distance over all states. Then, $\sum_{s \in S}[u^0(s) + gd(s)] - u^0(s^0) \leq \sum_{s \in S} gd(s) \leq n\overline{d} \leq nd$ for initially admissible u-values in safely explorable domains. Notice that the minimax goal distances of all states influence the upper complexity bound, not merely the minimax goal distance of the start state. In general, Theorem 4 implies that Min-Max LRTA* with initially admissible u-values has a complexity of at most $O(nd)$ action executions. If the domain is not safely explorable, then d is infinite and so is the upper complexity bound since Min-Max LRTA* might not reach a goal state. In safely explorable domains, however, $d \leq n - 1$, and Min-Max LRTA* with initially admissible u-values reaches a goal state after at most $nd \leq n(n-1)$ action executions or, in big-O notation, $O(n^2)$ action executions. Thus, it reaches a goal state after a finite number of action executions, which proves its correctness.

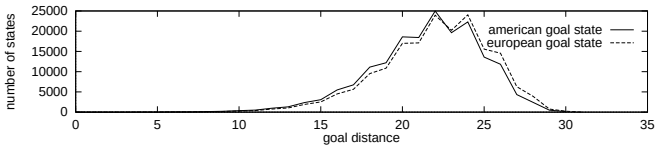


Figure 2.12: Goal Distances of the Eight Puzzle

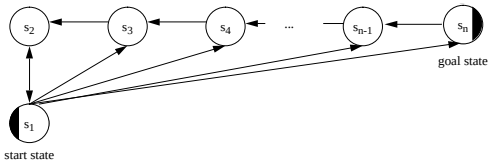


Figure 2.13: Worst-Case Domain for Min-Max LRTA*

To understand the usefulness of the upper complexity bound consider sliding-tile puzzles, which are sometimes considered to be hard search tasks because they have a small goal density. The eight puzzle, for example, has 181,440 states from which the goal state can be reached, but only one goal state. Although increasing the goal density tends to decrease the average minimax goal distance, there are search tasks with small goal density *and* small average minimax goal distance. The eight puzzle is an example: Figure 2.12 shows for every goal distance how many of the 181,440 states have this particular goal distance. It turns out that the average goal distance of the eight puzzle with the American goal state (Figure 2.7) is only 21.5 and its maximal goal distance is 30. Similarly, the average goal distance of the eight puzzle with the European goal state is 22.0 and its maximal goal distance is 31. (Reinefeld [Reinefeld, 1993] contains extensive statistics on the eight puzzle.) In both cases, the average goal distances are much smaller than the number of states. Thus, the eight puzzle is not particularly hard to search with Min-Max LRTA* among all domains with the same number of states: even if Min-Max LRTA* executes actions that do not decrease its goal distance, it can never move far away from the goal state. This does not imply, of course, that Min-Max LRTA* can search sliding-tile puzzles with a huge number of tiles, since its complexity depends not only on the average goal distance of the sliding-tile puzzle (relative to the number of states), but also the number of states, which is exponential in the number of tiles. This makes large sliding-tile puzzles difficult to search with any search method. Section 2.5 discusses this and other domain properties in more detail.

2.3.3.3 Lower Complexity Bound for Uninformed Min-Max LRTA*

Min-Max LRTA* is uninformed if its u-values are initialized with zero, that is, $u(s) := 0$ for all $s \in S$. Theorem 4 predicts that uninformed Min-Max LRTA* reaches a goal state after at most $\sum_{s \in S} gd(s) \leq \sum_{i=0}^{n-1} i = n^2/2 - n/2$ action executions in safely explorable domains. In this section, we prove, by example, that this upper complexity bound is tight for uninformed Min-Max LRTA* with look-ahead one. We use only deterministic example domains. This shows that the upper complexity bound is tight for this important subclass of nondeterministic domains and that deterministic domains, in general, are not easier to search with Min-Max LRTA* than nondeterministic domains. It also shows that the upper complexity bound is tight for uninformed LRTA* with look-ahead one since, in deterministic domains, Min-Max LRTA* behaves identically to LRTA*.

Theorem 5 *Zero-initialized Min-Max LRTA* with look-ahead one has a tight complexity of $O(nd)$ action executions over all nondeterministic domains and over all deterministic domains. It has a tight complexity of $O(n^2)$ action executions over all safely explorable, nondeterministic domains and over all safely explorable, deterministic domains.*

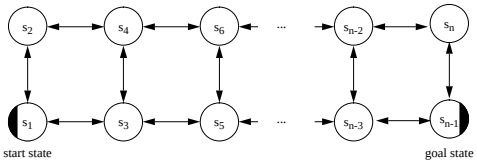


Figure 2.14: Rectangular Grid-World

Proof: (Upper Bound) Zero-initialized u-values are consistent and thus admissible. Theorem 4 implies that zero-initialized Min-Max LRTA* has a complexity that is at most $\sum_{s \in S} gd(s) \leq nd$ action executions over all nondeterministic domains and thus also over all deterministic domains, a subset of nondeterministic domains. Thus, zero-initialized Min-Max LRTA* has a complexity that is at most $O(nd)$ action executions over all nondeterministic domains and over all deterministic domains. $\sum_{s \in S} gd(s) \leq \sum_{i=0}^{n-1} i = n^2/2 - n/2$ for safely explorable domains. Thus, zero-initialized Min-Max LRTA* has a complexity that is at most $O(n^2)$ action executions over all safely explorable, nondeterministic domains and over all safely explorable, deterministic domains.

(Lower Bound) Figure 2.13 shows an example of a safely explorable, deterministic (and thus nondeterministic) domain for which the number of action executions that zero-initialized Min-Max LRTA* with look-ahead one needs in the worst case to reach a goal state is at least $n^2/2 - n/2$. Rather than explaining what occurs, we provide pseudo-code that prints the sequence of states that Min-Max LRTA* traverses. The scope of the for-statements is shown by indentation. The statements in their scope get executed only if the range of the for-variable is not empty. The default step size of the for-statements is either one ("to") or minus one ("downto"). It is easy to determine from an inspection of the code how many actions Min-Max LRTA* executes. Min-Max LRTA* traverses the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller indices.

```
for i := 1 to n-1
  for j := i downto 1
    print j
print n
```

In this case, Min-Max LRTA* executes $n^2/2 - n/2$ actions before it reaches the goal state (for $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_1, s_3, s_2, s_1, s_4, s_3, s_2, s_1, and s_5. Notice that $d = n - 1$ for the domain in Figure 2.13 (for $n \geq 1$). Thus, the complexity is both at least $O(nd)$ and $O(n^2)$ action executions. ■

The domain in Figure 2.13 was artificially constructed. Zero-initialized Min-Max LRTA* with look-ahead one has a tight complexity of $O(nd)$ action executions or, over all safely explorable domains, $O(n^2)$ action executions even for more realistic domains, such as grid-worlds. In fact, the complexity of zero-initialized Min-Max LRTA* with look-ahead one is tight at $O(nd)$ action executions and, over all safely explorable domains, $O(n^2)$ action executions even if the domains are undirected and the number of actions that can be executed in any state is bounded from above by a small constant (here: three). Figure 2.14 shows a safely explorable grid-world for which the number of action executions that zero-initialized Min-Max LRTA* with look-ahead one needs in the worst case to reach a goal state is at least $O(nd)$ or, alternatively, $O(n^2)$. In particular, it can traverse the state sequence that is printed by the following program in pseudo code.

```
for i := n-3 downto n/2 step 2
  for j := 1 to i step 2
    print j
  for j := i+1 downto 2 step 2
    print j
for i := 1 to n-1 step 2
  print i
```

Figure 2.15: One-Dimensional Grid-World (1)

In this case, Min-Max LRTA* executes $3n^2/16 - 3/4$ actions before it reaches the goal state (for $n \geq 2$ with $n \bmod 4 = 2$). For example, for $n = 10$, it traverses the state sequence s_1, s_3, s_5, s_7, s_8, s_6, s_4, s_2, s_1, s_3, s_5, s_6, s_4, s_2, s_1, s_3, s_5, s_7, and s_9. Notice that $d = n/2$ for the domain in Figure 2.14 (for even $n \geq 2$). Thus, the complexity of zero-initialized Min-Max LRTA* with look-ahead one is both at least $O(nd)$ and $O(n^2)$ action executions.

2.3.3.4 Lower Complexity Bound for Fully Informed Min-Max LRTA*

Min-Max LRTA* is fully informed if its u-values are initialized with the negative minimax goal distances of the states, that is, $u(s) := -gd(s)$ for all $s \in S$. Theorem 4 predicts that fully informed Min-Max LRTA* reaches a goal state after at most $gd(s_{start}) \leq n - 1$ action executions in safely explorable domains. In this section, we prove, by example, that this upper complexity bound is tight for fully informed Min-Max LRTA* with arbitrary look-ahead, including look-ahead one. No method can do better in the worst case. We use only deterministic example domains. This also shows that the upper complexity bound is tight for fully informed LRTA* since, in deterministic domains, Min-Max LRTA* behaves identically to LRTA*.

Theorem 6 *Min-Max LRTA* with arbitrary look-ahead whose u-values are initialized with the negative goal distances of the states has a tight complexity of $O(d)$ action executions over all nondeterministic domains and over all deterministic domains. It has a tight complexity of $O(n)$ action executions over all safely explorable, nondeterministic domains and over all safely explorable, deterministic domains. Furthermore, the complexity of every method is at least as large over these domains.*

Proof: (Upper Bound) The u-values of fully informed Min-Max LRTA* are consistent and thus admissible. Theorem 4 implies that fully informed Min-Max LRTA* has a complexity that is at most $gd(s_{start}) \leq d$ action executions over all safely explorable, nondeterministic domains and thus also over all safely explorable, deterministic domains, a subset of nondeterministic domains. Thus, fully informed Min-Max LRTA* has a complexity that is at most $O(d)$ action executions over all nondeterministic domains and over all deterministic domains. Its complexity is also bounded by $gd(s_{start})$ for domains that are not safely explorable since fully informed Min-Max LRTA* cannot accidentally execute actions that lead to a state with infinite minimax goal distance. $gd(s_{start}) \leq n - 1$ for safely explorable domains. Thus, fully informed Min-Max LRTA* has a complexity that is at most $O(n)$ action executions over all safely explorable, nondeterministic domains and over all safely explorable, deterministic domains.

(Lower Bound) Figure 2.15 shows an example of a safely explorable, deterministic (and thus nondeterministic) domain for which the number of action executions that fully informed Min-Max LRTA* needs in the worst case to reach a goal state is at least $n - 1$. Min-Max LRTA* traverses the state sequence that is printed by the following program in pseudo code.

```
for i := 1 to n
  print i
```

In this case, Min-Max LRTA* executes $n - 1$ actions before it reaches the goal state (for $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_3, s_4, and s_5. Every other method also traverses either this state sequence or a super sequence thereof. Notice that $d = n - 1$ for the domain in Figure 2.15 (for $n \geq 1$). Thus, the complexity is both at least $O(d)$ and $O(n)$ action executions. ■

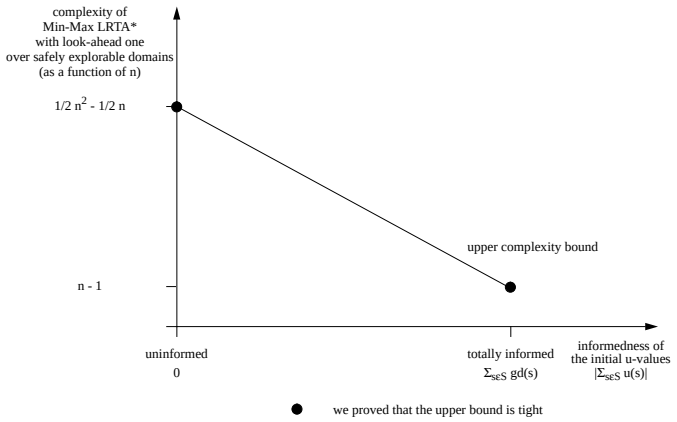


Figure 2.16: Upper Complexity Bound

initial u-values	informedness	rank	average-case performance	rank
Manhattan distance	2,661,120	1	326.61	1
Gaschnig's heuristic	1,461,168	2	2,235.62	3
tiles-out-of-order	1,290,240	3	1,409.81	2
zero heuristic	0	4	85,570.42	4

Figure 2.17: Informedness for the Eight Puzzle with the American Goal State

initial u-values	sum of goal distances	−	informedness	−	average initial u-value	=	average upper complexity bound	average-case performance
Manhattan distance	3,901,468	−	2,661,120	+	2,661,120 / 181,440	=	1,240,362.67	326.61
Gaschnig's heuristic	3,901,468	−	1,461,168	+	1,461,168 / 181,440	=	2,440,308.05	2,235.62
tiles-out-of-order	3,901,468	−	1,290,240	+	1,290,240 / 181,440	=	2,611,235.11	1,409.81
zero heuristic	3,901,468	−	0	+	0	=	3,901,468.00	85,570.42

Figure 2.18: Upper Complexity Bound for the Eight Puzzle with the American Goal State

2.3.3.5 Complexity Bounds and Performance Prediction

In this section, we show that the upper complexity bound of Theorem 4 does not necessarily reflect the performance of Min-Max LRTA* well for common test-beds of Min-Max LRTA*. We show that it can underestimate the performance dramatically. Furthermore, we show that the performance of Min-Max LRTA* can deteriorate as the initial u-values become better informed although the upper complexity bound is guaranteed to decrease. Ishida [Ishida, 1995] contains other experimental results on how properties of heuristic values affect the performance of LRTA*-type real-time search methods.

In the previous sections, we showed that uninformed Min-Max LRTA* with look-ahead one has a tight complexity of $n^2/2 - n/2$ action executions over all safely explorable domains. Fully informed Min-Max LRTA* with look-ahead one has a tight complexity of $n - 1$ action executions over all safely explorable domains. Thus, the complexity of Min-Max LRTA* eventually decreases as it becomes better informed (Figure 2.16). This does not mean, however, that the performance of Min-Max LRTA* in a given domain monotonically increases as the admissible u-values become better informed. We use two examples to illustrate this. Both example domains are deterministic. This also shows that the performance of LRTA* does not increase monotonically either since, in deterministic domains, Min-Max LRTA* behaves identically to LRTA*.

First Example: Consider the following heuristic functions for A* search on the eight puzzle with the American goal state (Figure 2.7) [Pearl, 1985]:

- *the Manhattan distance heuristic:* the smallest number of moves needed to achieve the goal configuration if two or more tiles can occupy the same square and it counts as one move when a tile is slid from its current square to an adjacent square (up, down, left, or right);
- *Gaschnig's heuristic:* the smallest number of moves needed to achieve the goal configuration if it counts as one move when a tile is removed from its current square and placed on the empty square;
- *the Tiles-Out-Of-Order heuristic:* the smallest number of moves needed to achieve the goal configuration if two or more tiles can occupy the same square and it counts as one move when a tile is removed from its current square and placed on any other square;
- *the zero-heuristic*, which always returns zero.

All four heuristic functions are consistent. Thus, consistent initial u-values can be obtained by using the negative of their heuristic values to initialize the u-values of Min-Max LRTA*. We use the absolute value of the sum of the u-values over all 181,440 states to measure how informed the resulting u-values are. We use this measure since it directly influences the upper complexity bound from Theorem 4: If we draw a start state randomly, then the average upper complexity bound is

$$\begin{aligned}\sum_{s\in S}[u^0(s)+gd(s)]-u^0(s^0) &= \sum_{s\in S}[u^0(s)+gd(s)]-\frac{1}{n}\sum_{s\in S}u^0(s)\\ &= \sum_{s\in S}gd(s)+(1-\frac{1}{n})\sum_{s\in S}u^0(s)\\ &= \sum_{s\in S}gd(s)-(1-\frac{1}{n})|\sum_{s\in S}u^0(s)|.\end{aligned}$$

Thus, the larger the informedness, the smaller the average upper complexity bound. Table 2.17 lists both the informedness and the number of actions executed by Min-Max LRTA* with look-ahead one averaged over 25,000 runs with randomly broken ties. The same 25,000 randomly chosen start states were used for all heuristic functions. Table 2.18 uses this data to show that the actual performance of Min-Max LRTA* can be much better than the upper complexity bound of Theorem 4 suggests. Table 2.17 shows that better informed initial u-values do not necessarily increase the performance of Min-Max LRTA* for a given domain. We say that u-values $u(s)$ are *state-wise better informed* than u-values $u'(s)$ if and only if $u(s) \leq u'(s)$ for all $s \in S$ and $u(s) < u'(s)$ for at least one $s \in S$. The u-values of both the Manhattan distance and Gaschnig's heuristic are state-wise better informed than the u-values of the tile-out-of-order heuristic, which in turn are state-wise better informed than the u-values of the zero heuristic. The table shows that the average number of actions executed by Min-Max LRTA* with look-ahead one is larger for Gaschnig's heuristic than for the tiles-out-of-order heuristic although Gaschnig's heuristic is state-wise better informed than the tiles-out-of-order heuristic. Min-Max LRTA* with look-ahead one and the tiles-out-of-order heuristic needs, on average, 1,410 action executions to reach the goal state, compared to 2,236 action executions for Min-Max LRTA* with look-ahead one and Gaschnig's heuristic. Out of the 25,000 runs, Min-Max LRTA* with look-ahead one and the tiles-out-of-order heuristic outperforms Min-Max LRTA* with look-ahead one and Gaschnig's heuristic 15287 times, is beaten 9682 times, and ties 31 times.

Second Example: The first example involved sampling: the start states were determined randomly and ties among indistinguishable actions were broken randomly as well. We therefore present another example that is not subject to sampling error. Consider the one-dimensional grid-world in Figure 2.15. Zero-initialized Min-Max LRTA* with look-ahead one traverses the state sequence printed by the following program in pseudo code.

```
for i := 1 to n
  print i
```

Thus, Min-Max LRTA* executes $n-1$ actions before it reaches the goal state (for $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_3, s_4, and s_5. Now assume that the u-value of the state marked X is decreased to -1. The resulting u-values are still consistent and state-wise better informed than before. However, Min-Max LRTA* with look-ahead one can now traverse the state sequence printed by the following program in pseudo code if ties are broken in favor of successor states with smaller indices.

```
for i := 1 to n-2
  print i
for i := n-3 downto 1
  print i
for i := 2 to n
  print i
```

In this case, Min-Max LRTA* executes $3n-7$ actions before it reaches the goal state (for $n \geq 3$), and $3n-7 > n-1$ for $n > 3$. For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_3, s_2, s_1, s_2, s_3, s_4, and s_5.

Both examples illustrate that the performance of Min-Max LRTA* is not completely correlated with the informedness of the initial u-values. Why is this so? In deterministic domains, the action-selection step of Min-Max LRTA* always chooses the action that leads to the successor state with the maximal u-value. We therefore expect Min-Max LRTA* to do well if there is a good chance that it comes across a goal state when it mostly performs steepest ascent on the initial u-values. This means that the differences in u-values of the successor states are more important than how close the u-values are to the negative goal distances. Consequently, the fewer local maxima there are in the initial u-value surface, the better we expect the performance of Min-Max LRTA* to be. Ishida [Ishida, 1992] calls these local maxima "depressions" (the local optima are local minima for him, since he works with u-values that approximate the positive minimax goal distances). Min-Max LRTA* differs in this respect from A*, that cannot expand more states if consistent heuristic values are state-wise better informed than others, except maybe for states whose f-values equal the goal distance of the start state [Pearl, 1985].

2.3.3.6 Convergence

If Min-Max LRTA* solves the same search task repeatedly (even with different start states) it improves its behavior over time. This is an important difference between Min-Max LRTA* and many other planning methods that interleave planning and plan execution, because no method that executes actions before it knows their complete consequences can guarantee a good behavior right away. One-shot planning methods do not anticipate that they might have to solve a planning task more than once. If they are not able to improve their behavior over time if they unexpectedly have to solve the same planning tasks multiple times, then they are not efficient in the long run.

Min-Max LRTA* improves its behavior over time by transferring domain knowledge, in the form of u-values, between search tasks in the same domain with the same set of goal states. This is a familiar concept: One can argue that the searches that Min-Max LRTA* performs between plan executions are independent of one another and that they are connected only by the u-values that transfer domain knowledge between them. To see why Min-Max LRTA* can do the same thing between search tasks, assume that a series of search tasks in the same domain with the same set of goal states is given. If the initial u-values of Min-Max LRTA* are admissible for the first search task, then they are also admissible after Min-Max LRTA* has solved the task and are state-wise at least as informed as initially (Theorem 2). Thus, they are also admissible for the second search task and Min-Max LRTA* can continue to use the same u-values across search tasks. The start states of the search tasks can differ since the admissibility of the u-values does not depend on the start states. This way, Min-Max LRTA* can transfer acquired domain knowledge from one search task to the next, thereby making its u-values better informed. The resulting improvement in performance is often nonmonotonic,

since more informed u-values do not necessarily improve the performance of Min-Max LRTA* immediately (Section 2.3.3.5). The following theorems formalize this knowledge transfer:

Theorem 7 *Min-Max LRTA* with initially admissible u-values reaches a goal state in safely explorable domains after at most $gd(s_{start})$ action executions (regardless of the behavior of nature) if its u-values do not change during the search.*

Proof:

$$\begin{aligned}
u^{t+1}(s^{t+1}) - u^t(s^t) &= u^{t+1}(s^{t+1}) - u^{t+1}(s^t) && \text{since the u-values do not change} \\
&\overset{\text{Theorem 1}}{=} u^{t+1}(s^{t+1}) - \min(u^t(s^t), -1 + \max_{a \in A(s^t)} \min_{s' \in succ(s^t,a)} u^{t+1}(s')) \\
&\geq u^{t+1}(s^{t+1}) + 1 - \max_{a \in A(s^t)} \min_{s' \in succ(s^t,a)} u^{t+1}(s') \\
&\geq u^{t+1}(s^{t+1}) + 1 - u^{t+1}(s^{t+1}) \\
&= 1.
\end{aligned}$$

Thus, the difference in u-values between the next state and the current state is at least one. Since the u-values of all goal states are zero and the u-value of the start state is at least $gd(s_{start})$, by induction Min-Max LRTA* needs at most $gd(s_{start})$ action executions to reach a goal state. ■

Theorem 7 can be used to derive the following convergence result for Min-Max LRTA* in the mistake-bounded error model. The mistake-bounded error model is one way of analyzing learning methods by bounding the number of mistakes (for example, wrong predictions) that they make:

Theorem 8 *Assume that Min-Max LRTA* maintains u-values across a series of search tasks in the same safely explorable domain with the same set of goal states. Then, the number of search tasks for which Min-Max LRTA* with initially admissible u-values reaches a goal state after more than $gd(s_{start})$ action executions is bounded from above by a constant that depends only on the domain and goal states.*

Proof: The u-values are always admissible during all search tasks. If Min-Max LRTA* reaches a goal state after more than $gd(s_{start})$ action executions, then at least one u-value has changed (Theorem 7). This can happen only a finite number of times since the u-values are monotonically nonincreasing and bounded from below by the negative minimax goal distances. ■

In this context, it counts as one mistake when Min-Max LRTA* reaches a goal state after more than $gd(s_{start})$ action executions. According to Theorem 8, the u-values converge after a bounded number of mistakes. Determining the number of mistakes is not important in the following. The u-values do not necessarily converge after a bounded number of search tasks, even if Min-Max LRTA* does not change its behavior, that is, always selects its local search spaces in the same way and always breaks ties among equally good actions in the same way. In particular, it is not true that, when the u-values have not changed during one search task, they cannot change during any future search task. This is so because the behavior of Min-Max LRTA* is not determined exclusively by the u-values: Min-Max LRTA* does not perform a complete minimax search but partially relies on observing the actual successor states of action executions, and nature can prevent Min-Max LRTA* from experiencing some of the action outcomes (for example, the worst-case action outcomes) for an arbitrarily long time or choose not to let Min-Max LRTA* experience them at all. However, whenever nature lets Min-Max LRTA* experience a novel action outcome, Min-Max LRTA* learns from that, which allows it to bound the number of mistakes it makes. For the same reason, the action sequence after convergence depends on the behavior of nature. However, it is guaranteed that the action sequence has $gd(s_{start})$ *or fewer*

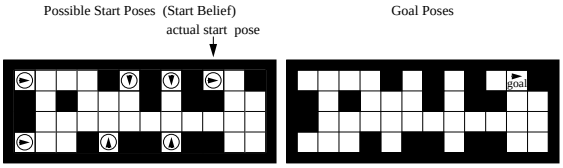


Figure 2.19: Goal-Directed Navigation Task

actions, that is, it is either worst-case optimal or better than worst-case optimal. This is so because nature might not be as malicious as a minimax search assumes.

To summarize, Min-Max LRTA* finds suboptimal plans. During the search it gains experience with the search task. It uses this experience subsequently to find better plans for similar planning tasks, that is, planning tasks that can differ in their start states. Thus, Min-Max LRTA* improves its behavior over time. This has two advantages: First, Min-Max LRTA* amortizes learning over several search episodes. This allows it to find suboptimal plans fast. Second, since Min-Max LRTA* partially relies on observing the actual successor states of action executions it does not plan for all possible successor states and thus can still have computational advantages even over several search episodes compared to a complete minimax search if nature is not as malicious as a minimax search assumes and some successor states do not occur in practice. This also means that Min-Max LRTA* might be able to solve search tasks in domains that are not safely explorable, although this is not guaranteed.

2.3.4 Summary of Results on Min-Max LRTA*

Min-Max LRTA* extends LRTA* to nondeterministic domains by interleaving minimax searches with plan executions. Its complexity is finite in domains that are safely explorable, that is, if the minimax goal distance of every state is finite. Min-Max LRTA* has three features: It allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge (in the form of admissible initial u-values) to guide planning, and improves its performance over time as it solves similar planning tasks. Since Min-Max LRTA* partially relies on observing the actual successor states of action executions it does not plan for all possible successor states and thus can still have computational advantages even over several search episodes compared to a complete minimax search if nature is not as malicious as a minimax search assumes.

2.4 An Application: Robot-Navigation Tasks

In this section, we present a case study that illustrates the application of Min-Max LRTA* to nondeterministic robot navigation domains. We study goal-directed navigation tasks in mazes [Genesereth and Nourbakhsh, 1993]. The robot knows the maze, but is uncertain about its start pose, where a *pose* is a location (square) and orientation (north, east, south, west). We assume that there is no uncertainty in actuation and sensing.[7] The sensors on-board the robot tell it in every pose whether there are walls immediately adjacent to it in the four directions relative to the orientation of the robot (front, left, behind, right). The robot's actions are to move forward one square (unless there is a wall directly in front of it), turn left ninety degrees, or turn right ninety degrees. The task of the robot is to navigate to a given goal pose and stop. Since there can be many poses that produce the same sensor reports as the goal poses, solving the goal-directed navigation task includes localizing the robot sufficiently so that it knows that it is at a goal pose when it stops.

Goal-directed navigation tasks with initial pose uncertainty are good tasks for interleaving planning and plan execution because interleaving planning and plan execution allows the robot to gather information early,

[7] Chapter 3 discusses planning methods for noisy actuators and sensors.

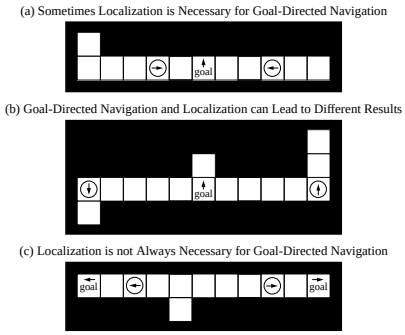


Figure 2.20: Goal-Directed Navigation and Localization

which reduces its pose uncertainty and thus the number of situations that its plans have to cover. This makes subsequent planning more efficient. For example, assume that the robot has no knowledge of its start pose for the goal-directed navigation task from Figure 2.19, but observes walls around it except in its front. This results in the seven possible start poses shown in the figure. If the robot executes a forward action and then observes walls in its front and to its left and openings on the other two sides, it can infer that it has solved the goal-directed navigation task and does not need to plan any further. In particular, it does not need to decide what it should have done had the observation been different.

We require that the mazes be strongly connected (every pose can be reached from every other pose) and not completely symmetrical (localization is possible). This modest assumption makes all goal-directed navigation tasks solvable, since the robot can always first localize itself and then move to a goal pose. It can indeed be optimal for the robot to solve goal-directed navigation tasks this way. Figure 2.20(a) shows an example. Since the observations at the goal pose do not uniquely identify the goal pose, it is best for the robot to first localize itself by turning around and moving forward twice. At this point, the robot has localized itself and can navigate to the goal pose. However, it is not always optimal for the robot to solve goal-directed navigation tasks this way. Figure 2.20(b) shows an example. To solve the goal-directed navigation task, it is best for the robot to turn left and move forward until it sees a corridor opening on one of its sides. At this point, the robot has localized itself and can navigate to the goal pose. On the other hand, to solve the corresponding localization task, it is best for the robot to move forward once. At this point, the robot has localized itself. Finally, some goal-directed navigation tasks can be solved without localizing the robot at all. Figure 2.20(c) shows an example. To solve the goal-directed navigation task, it is best for the robot to move forward twice. At this point, the robot knows that it is in a goal pose but cannot be certain which of the two goal poses it is in.

2.4.1 Formalizing the Robot-Navigation Tasks

In this section, we formalize the goal-directed navigation tasks: P is the set of possible robot poses. $A(p)$ is the set of possible actions that the robot can execute in pose $p \in P$: left, right, and possibly forward. $succ(p, a)$ is the pose that results from the execution of action $a \in A(p)$ in pose $p \in P$. $o(p)$ is the observation that the robot makes in pose $p \in P$: whether or not there are walls immediately adjacent to it in the four directions (front, left, behind, right).

The robot starts in pose $p_{start} \in P$ and then repeatedly makes an observation and executes an action until it decides to stop. It knows the maze, but is uncertain about its start pose. It could be in any pose in $P_{start} \subseteq P$. We require only that $o(p) = o(p')$ for all $p, p' \in P_{start}$, which automatically holds after the first observation, and $p_{start} \in P_{start}$, which automatically holds for $P_{start} = \{p | p \in P \wedge o(p) = o(p_{start})\}$. The robot has to navigate to any pose in $\emptyset \neq P_{goal} \subseteq P$ and stop.

When the robot is not certain about its pose, the best it can do is to maintain a belief about its current pose. We assume that the robot cannot associate probabilities or other likelihood estimates with the poses. Then,

all it can do is to maintain a set of possible poses ("beliefs"). We use the following notation: B is the set of beliefs, b_{start} the start belief, and B_{goal} the set of goal beliefs. $A(b)$ is the set of actions that can be executed when the belief is b. $O(b, a)$ is the set of possible observations that can be made after the execution of action a when the belief was b. $succ(b, a, o)$ is the successor belief that results if observation o is made after the execution of action a when the belief was b.

$$\begin{aligned} B &= \{b | b \subseteq P\} \\ b_{start} &= P_{start} \\ B_{goal} &= \{b | b \subseteq P_{goal}\} \\ A(b) &= A(p) \text{ for any } p \in b \\ O(b, a) &= \{o(succ(p, a)) | p \in b\} \\ succ(b, a, o) &= \{succ(p, a) | p \in b \wedge o(succ(p, a)) = o\} \end{aligned}$$

All beliefs b that do not satisfy that $o(p) = o(p')$ for all $p, p' \in b$ could be removed from B because they cannot be encountered. To understand the definition of $A(b)$, notice that $A(p) = A(p')$ for all $p, p' \in b$ after the preceding observation since the observation determines the actions that can be executed. To understand the definition of B_{goal}, notice that the robot knows that it is in a goal pose if the belief is $b \subseteq P_{goal}$. If the belief contains more than one pose, however, the robot does not know which goal pose it is in. An example was discussed in the context of Figure 2.20(c). If it is important that the robot knows which goal pose it is in, we define $B_{goal} = \{b | b \subseteq P_{goal} \wedge |b| = 1\}$ instead of $B_{goal} = \{b | b \subseteq P_{goal}\}$.

The robot navigation domain is deterministic and small (*pose space*). However, the beliefs of the robot depend on its observations, which the robot cannot predict with certainty since it is uncertain about its pose. We therefore formulate a goal-directed navigation task as a search task in a domain whose states are the beliefs of the robot (*belief space*). Beliefs are sets of poses. Thus, the number of beliefs is exponential in the number of poses, and the belief space is not only nondeterministic but can also be large. It can be described as follows: $S = B$, $s_{start} = b_{start}$, and $G = B_{goal}$. The set of actions that can be executed in state s is $A(s) = A(b)$ for $s = b$. The set of successor states that can result from the execution of action a in state s is $succ(s, a) = \{succ(b, a, o) | o \in O(b, a)\}$ for $s = b$. The actual successor state that results from the execution of action a in state s is determined by the observation o made after the action execution. It is $succ(b, a, o)$ for $s = b$.

The belief space satisfies our assumptions: It is safely explorable (since we assume that the robot can always localize itself and then move to a goal pose) and every action execution necessarily results in a state change. Traversing any path from the start state in the belief space to a goal state solves the corresponding goal-directed navigation task. The path does not have to be optimal or repeatable. Assume, for example, that the robot has successfully solved some goal-directed navigation task and now has to solve the same kind of goal-directed navigation task in the same maze with the same start and goal beliefs. Even if the robot attempts to execute the same plan again, it can make different observations since, unknown to the robot, its start pose might have changed. This is possible since many start poses can be consistent with the same start belief. The change can result in different beliefs during plan execution and thus different trajectories through the domain. Since there was no reason to plan for the new beliefs, the previous plan might not cover them.

2.4.2 Features of Min-Max LRTA* for the Robot-Navigation Tasks

Earlier, we discussed the following features of Min-Max LRTA*: that it allows for fine-grained control over how much planning to do between plan executions, that it uses heuristic knowledge to guide planning, and that it improves its performance over time as it solves similar planning tasks. In this section, we discuss these features in the context of the robot-navigation tasks.

Fine-Grained Control: Min-Max LRTA* allows for fine-grained control over how much planning to do between plan executions. We have argued earlier that look-aheads of one or two action executions can be optimal for sufficiently fast moving agents, that is, agents that can execute plans with a similar speed as they can generate them. Robots, however, cannot execute actions that fast and thus larger look-aheads can be expected to outperform small look-aheads.

Heuristic Knowledge: Min-Max LRTA* uses heuristic knowledge (in the form of admissible initial u-values) to guide planning. For goal-directed navigation tasks, one can use the *goal-distance heuristic* to initialize the u-values, that is, $u(s) = -\max_{p \in s} gd(\{p\})$. The calculation of $gd(\{p\})$ involves no pose uncertainty and can be done efficiently without interleaving planning and plan execution, by using traditional search methods in the pose space. This is possible because the pose space is deterministic and small. The u-values $u(s)$ are admissible because the robot believes that it might start in pose $p_{start} =$ one-of $\arg\max_{p \in b} gd(\{p\})$. If it starts in this pose and knows that, it needs $gd(\{p_{start}\}) = \max_{p \in b} gd(\{p\})$ action executions in the worst case to solve the goal-directed navigation task. If it starts in this pose but does not know that, it might need more (but cannot need fewer) action executions in the worst case to solve the goal-directed navigation task, because it might have to execute additional localization actions to overcome its pose uncertainty. Thus, the u-values are admissible but often only partially informed.

Knowledge Transfer: Min-Max LRTA* improves the performance of the robot over time as it solves similar planning tasks. In particular, it can transfer domain knowledge between goal-directed navigation tasks with the same goal poses in the same maze. The actual start poses or the beliefs of the robot about its start poses do not need to be identical. This way, the robot learns action sequences over time that are at least worst-case optimal. Assume, for example, that the robot repeatedly solves the same goal-directed navigation task with the same start pose but does not know that its start pose remains the same.[8] Then, the behavior of nature does not change over time since it is completely determined by the actual start pose of the robot, that remains the same for all tasks. Thus, nature cannot exhibit the behavior described in Section 2.3.3.6 and fool Min-Max LRTA* for an arbitrary long time. Assume further that the way how Min-Max LRTA* selects its local search spaces does not change over time and that Line 5 in Figure 2.10 breaks ties systematically according to a predetermined ordering on $A(s)$ for all states s. Then, when the u-values do not change during one goal-directed navigation task, they cannot change during any future navigation task because the behavior of Min-Max LRTA* is now determined completely by the u-values. Since the u-values are monotonically nonincreasing and bounded from below, they and thus also the executed action sequence converge after a bounded number of search tasks. From then on, the robot solves the goal-directed navigation task with at most $gd(s_{start})$ action executions.

2.4.3 Extensions of Min-Max LRTA* for the Robot-Navigation Tasks

Min-Max LRTA* uses a minimax search in the local search spaces to update its u-values. For the robot-navigation tasks, it is possible to combine this with updates over a greater distance, with only a small amount of additional effort. For example, we know that $gd(s) \geq gd(s')$ for any two states $s, s' \in S$ with $s \supseteq s'$ (recall that states are sets of poses). Thus, we can set $u(s) := \min(u(s), u(s'))$ for selected states $s, s' \in S$ with $s \supset s'$. If the u-values are admissible before the update, they remain admissible afterwards. The assignment could be done immediately before the local search space is generated on Line 3 in Figure 2.10.

2.4.4 Related Search Methods

In this section, we describe various kinds of robot-navigation tasks and special-purpose methods for solving them. We compare Min-Max LRTA* to these methods.

[8] If the robot knew that its start pose remains the same, it could use the decreased uncertainty about its pose after solving the goal-directed navigation task to narrow down its start pose and improve its navigation performance this way.

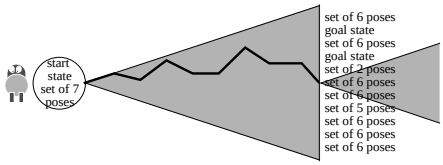


Figure 2.21: IG Method

1. $S_{lss} = \{s\}$.
2. Update $u(s)$ for all $s \in S_{lss}$ (Figure 2.11).
3. $s' := s$.
4. $a :=$ one-of $\arg\max_{a' \in A(s')} \min_{s'' \in succ(s',a')} u(s'')$.
5. If $|succ(s', a)| > 1$, then return.
6. $s' := s''$, where $s'' \in succ(s', a)$ is unique.
7. If $s' \in S_{lss}$, then go to 4.
8. If $s' \in G$, then return.
9. $S_{lss} := S_{lss} \cup \{s'\}$ and go to 2.

Figure 2.22: Generating Local Search Spaces with Larger Look-Aheads

2.4.4.1 Goal-Directed Navigation: IG Method

So far, we have applied Min-Max LRTA* to goal-directed navigation tasks. The *Information-Gain Method* (IG method) [Genesereth and Nourbakhsh, 1993] is an earlier method that demonstrated the advantage of interleaving of planning and plan execution for goal-directed navigation tasks.[9] It uses breadth-first search (to be precise: iterative deepening) on an and-or graph around the current state in conjunction with pruning rules to find subplans that achieve a gain in information, in the following sense: after the execution of a subplan, the robot has either solved the goal-directed navigation task or at least reduced the number of poses the robot can be in (Figure 2.21). This way, the IG method guarantees progress towards the goal.

There are **similarities between the IG method and Min-Max LRTA***: Both methods combine real-time search with minimax search. In fact, Min-Max LRTA* can exhibit a similar behavior as the IG method: zero-initialized Min-Max LRTA* that generates the local search spaces with the method in Figure 2.22 also performs a breadth-first search around the current state until it finds a subplan whose execution results in a gain in information. The method does this by starting with the minimal local search space, that contains only the current state. It performs a minimax search in the local search space and then simulates the action executions of Min-Max LRTA* starting from the current state. If the simulated action executions reach a goal state or lead to a gain in information, then the method returns. However, if the simulated action executions leave the local search space, the method halts the simulation, adds the state outside of the local search space to the local search space, and repeats the procedure. Notice that, when the method returns, it has already updated the values of all states of the local search space. Thus, Min-Max LRTA* does not need to update the values of these states again and can skip the value-update step. Its action-selection step and the simulation have to break ties identically. Then, Min-Max LRTA* with Line 8 in Figure 2.10 executes actions until it either reaches a goal state or gains information.

There are also **differences between the IG method and Min-Max LRTA***: The IG method does not need to maintain information between plan executions, whereas Min-Max LRTA* has to maintain information in

[9] Genesereth and Nourbakhsh [Genesereth and Nourbakhsh, 1993] refer to the IG method as the Delayed Planning Architecture (DPA) with the viable plan heuristic. They also state some improvements on the variant of the IG method discussed here, that do not change its character.

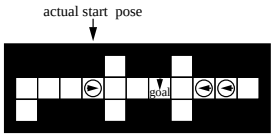


Figure 2.23: Another Goal-Directed Navigation Task

the form of u-values. To save memory, Min-Max LRTA* can generate the initial u-values on demand and never store u-values that are identical to their initial values. However, even then its memory requirements are bounded only by the number of states, but Section 2.4.5 shows that they are small in practice.

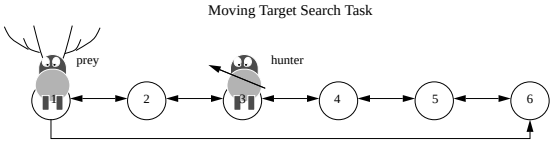
The goal-directed robot-navigation tasks can be solved with the IG method. Different from the IG method, Min-Max LRTA* is a general-purpose planning method in nondeterministic domains that allows for fine-grained control over how much planning to do between plan executions (including small look-aheads that do not guarantee a gain in information – the decrease of the potential $\sum_{s \in S \setminus \{s^t\}} u^t(s)$ (Page 22) can be interpreted as a virtual gain in information), uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks. Being able to improve the performance of the robot over time is a particularly important advantage of Min-Max LRTA* because no method that interleaves planning and plan execution can guarantee an optimal behavior on the first run even if, like Min-Max LRTA*, it uses heuristic knowledge to guide planning. For instance, consider the goal-directed navigation task in Figure 2.23 and assume that Min-Max LRTA* generates the local search spaces with the method in Figure 2.22. Then, both the IG method and zero-initialized Min-Max LRTA* move forward, because this is the fastest way to eliminate a possible pose, that is, to gain information. Even Min-Max LRTA* with the goal-distance heuristic moves forward, since it follows the gradient of the u-values. However, moving forward is suboptimal. It is best for the robot to first localize itself by turning around and moving to a corridor end. If the goal-directed navigation task is repeated a sufficient number of times with the same start pose, Min-Max LRTA* eventually learns this behavior.

2.4.4.2 Localization: Homing Sequences

Min-Max LRTA* can also be applied to localization tasks in mazes. These tasks are identical to the goal-directed navigation tasks studied earlier except that the robot has to achieve only certainty about its pose. Min-Max LRTA* can be applied to these tasks unchanged if the definition of G is changed to $G = B_{goal} = \{b | b \subseteq P \wedge |b| = 1\}$. For the localization tasks, it is difficult to obtain better informed initial u-values than those provided by the zero heuristic (zero-initialized u-values).

Localization tasks are related to finding *homing sequences* or adaptive homing sequences for deterministic finite state automata whose states are colored. A homing sequence is a sequential plan (action sequence) with the property that the observations made during its execution uniquely determine the resulting state [Kohavi, 1978]. An adaptive homing sequence is a conditional plan with the same property [Schapire, 1992]. For every reduced deterministic finite state automaton, there exists a homing sequence that contains at most $(n-1)^2$ actions. (A reduced finite state automaton is one where, for every pair of different states, there exists some action sequence that distinguished them.) Finding a shortest homing sequence is NP-complete but a suboptimal homing sequence of at most $(n-1)^2$ actions can be found in polynomial time [Schapire, 1992]. Thus, finding homing sequences is an example of tasks for which optimal planning is intractable but suboptimal planning is tractable and determines plans of good quality. In this case, suboptimal planning can reduce the sum of planning and plan-execution time for one-shot planning tasks because the computational savings of finding suboptimal plans can outweigh the overhead of executing them for sufficiently fast moving agents.

Robot-localization tasks can be solved with homing sequences since the pose space is deterministic and thus can be modeled as a deterministic finite state automaton. More generally, homing sequences can be used for



Corresponding Non-Deterministic Single-Agent Domain

state = (hunter, prey)	action of hunter	successor states
(done) = goal state	—	—
(3,1) = start state	move to 2	(2,6) or (done)
	move to 4	(4,2) or (4,6)
(2,6)	move to 1	(1,5)
	move to 3	(3,5)
(4,2)	move to 3	(3,1) or (done)
	move to 5	(5,1) or (5,3)
(4,6)	move to 3	(3,5)
	move to 5	(done)
...	...	...

Figure 2.24: Simple Moving-Target Search Task

planning in deterministic domains that appear to be nondeterministic due to state uncertainty. Different from homing sequences, Min-Max LRTA* is a general-purpose planning method in nondeterministic domains that reduces the amount of planning done for irrelevant (unencountered) situations. It allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks.

2.4.4.3 Moving-Target Search: MTS Method

Min-Max LRTA* can also be applied to moving-target search, the task being for a hunter to catch an independently moving prey. We discuss the variant of moving-target search where both agents move on a known directed graph. The hunter moves first, then they alternate moves to adjacent vertices. Both agents can always sense the current vertex of themselves and the other agent, but the hunter does not know in advance where the prey moves. The hunter catches the prey if both agents occupy the same vertex. In our framework, the agent is the hunter. It is straightforward to map the (two-agent) moving-target search task to a (single-agent) search task against nature in a nondeterministic domain (Figure 2.24). The hunter can catch the prey for sure if the minimax goal distance of the start state in the nondeterministic domain is finite. Min-Max LRTA* can be used in the nondeterministic domain to determine a behavior for the hunter that catches the prey if the nondeterministic domain is safely explorable.

This application of Min-Max LRTA* is interesting since it makes different assumptions than other LRTA*-type real-time search methods that have been used for moving-target search. Min-Max LRTA* can solve moving-target search tasks that these search methods cannot solve. In particular, the *Moving-Target Search method* (MTS) [Ishida and Korf, 1991] is another LRTA*-type real-time search method that applies to moving-target search, but utilizes LRTA* for the hunter in a different way. It learns the following behavior for the hunter: always move to an adjacent vertex that is on a shortest path to the current vertex of the prey. Ishida and Korf [Ishida and Korf, 1991] prove that the hunter eventually catches the prey on a strongly connected graph if it is faster moving than the prey. Notice the difference between the two methods: Obviously, one has to make some assumptions to ensure that the prey cannot force the hunter into a cycle in which the hunter cannot decrease its distance to the prey. Min-Max LRTA* assumes that the nondeterministic domain is safely explorable, which means that a skillful hunter is able to corner the prey. MTS, on the other hand, does not restrict the topology of the graph, but has to assume that the hunter has a speed advantage over the prey. Consider, for example, the graph from Figure 2.24 (notice that one of its edges is directed) and assume that both agents are equally fast moving. Fully informed MTS always follows the prey at the same distance if the prey runs around in an anti-clockwise cycle. Min-Max LRTA*, however, eventually goes left until the prey takes the one-way street and later goes right until the prey is caught, no matter how the prey behaves.

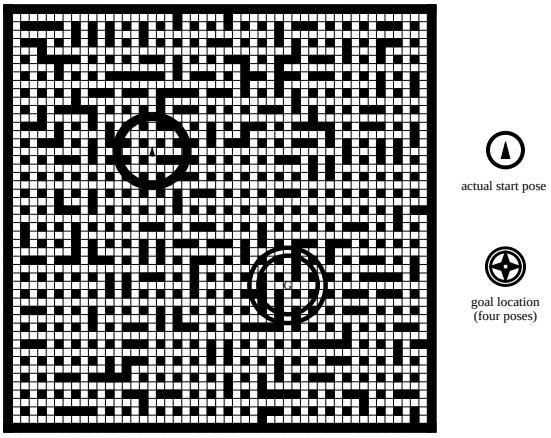


Figure 2.25: Sample Maze

after . . .	measuring . . .	using . . .	Min-Max LRTA* with look-ahead one	
			goal-directed navigation goal-distance heuristic	localization zero heuristic
the first run	plan execution time (performance)	action executions	113.32	13.33
	planning time	state expansions	113.32	13.33
	memory usage	u-values remembered	31.88	13.32
convergence	plan execution time (performance)	action executions	49.15	8.82
	planning time	state expansions	49.15	8.82
	memory usage	u-values remembered	446.13	1,782.26
number of runs until convergence			16.49	102.90

after . . .	measuring . . .	using . . .	Min-Max LRTA* with larger look-ahead (using the method in Figure 2.22)	
			goal-directed navigation goal-distance heuristic	localization zero heuristic
the first run	plan execution time (performance)	action executions	50.48	12.24
	planning time	state expansions	73.46	26.62
	memory usage	u-values remembered	30.28	26.62
convergence	plan execution time (performance)	action executions	49.13	8.81
	planning time	state expansions	49.13	8.81
	memory usage	u-values remembered	85.80	506.63
number of runs until convergence			3.14	21.55

Figure 2.26: Experimental Results

2.4.5 Experiments with a Simulator

Nourbakhsh [Nourbakhsh, 1996] has already shown that performing a complete minimax search to solve the goal-directed navigation tasks optimally can be infeasible. In this section, we take this result for granted and show that Min-Max LRTA* solves the goal-directed navigation tasks fast, converges quickly, and requires only a small amount of memory. We do this experimentally since the actual performance of Min-Max LRTA* can be better than the upper complexity bound of Theorem 4 suggests (Section 2.3.3.5).

We use a simulation of the robot navigation domain whose interface matches the interface of an actual robot that operates in mazes [Nourbakhsh and Genesereth, 1997]. Thus, Min-Max LRTA* could be run on that robot. We apply Min-Max LRTA* to goal-directed navigation and localization tasks with two different look-aheads each, namely look-ahead one and the larger look-ahead from Figure 2.22. As test domains, we use 500 randomly generated square mazes. The same 500 mazes are used for all experiments. All mazes have size 49×49 and the same obstacle density, the same start pose of the robot, and – for goal-directed navigation tasks – the same goal location (which includes all four poses). Figure 2.25 shows an example. The number of states of the belief space is exponential in the number of poses (although many of them are not relevant for the tasks), and there are more than 6,500 poses. In the example, the robot initially senses openings in all four directions. More than 1100 poses are consistent with this observation. Min-Max LRTA*

is provided with no additional knowledge of the start pose and runs repeatedly on the same task with the same start pose until its behavior converges. Figure 2.26 shows that Min-Max LRTA* indeed produces good plans in large domains quickly, while using only a small amount of memory. Since the performance of Min-Max LRTA* after convergence is no worse than the minimax goal distance of the start state, we know that its initial performance is at most 231, 151, 103, and 139 percent (respectively) of the optimal worst-case performance. Min-Max LRTA* also converges quickly. For goal-directed navigation tasks, for example, Min-Max LRTA* with look-ahead one converges in less than 20 runs and has doubled its performance. This demonstrates that this aspect of Min-Max LRTA* is important if the heuristic values do not guide planning sufficiently well.

2.5 LRTA*-Type Real-Time Search Methods and Domain Properties

So far, we have introduced Min-Max LRTA* and applied it to goal-directed navigation tasks. This extended the application domains of LRTA*-type real-time search methods. Although people have studied which factors influence the performance of traditional search methods, currently not much is known about the performance of LRTA*-type real-time search methods. Thus, it is not only important to extend their application domains, but also to provide some of the foundations for understanding their performance. This includes how domain properties, heuristic knowledge of the domains, and the amount of look-ahead influence the performance of LRTA*-type real-time search methods. We have already studied some of these properties in the context of Min-Max LRTA*.

Ideally, however, we would like to make statements about how these properties affect LRTA*-type real-time search methods as a group. Given the current state of the art, this goal is ambitious. This thesis therefore provides only a first step in this direction. We study, both formally and experimentally, two domain properties and their effect on the performance of selected LRTA*-type real-time search methods. To simplify our task even further, we study the effect of these domain properties in isolation from the other properties: we study uninformed LRTA*-type real-time search methods with minimal look-ahead, that is, a look-ahead that is even smaller than that of Min-Max LRTA* with look-ahead one (we explain this in the next section). Thus, there is no difference in the kind and amount of domain knowledge that the LRTA*-type real-time search methods have available and how they utilize it. Consequently, the studied LRTA*-type real-time search methods are similar, which makes it easier to make statements that hold for all of them. Our assumptions are satisfied by several LRTA*-type real-time search methods that have successfully been used in the literature, including Edge Counting, variants of Min-Max LRTA*, BETA, and variants of reinforcement-learning methods (including Q-Learning). We discuss these methods in detail in the following sections. Since some of them operate only in deterministic domains, we restrict our analysis to deterministic domains.

In the following, we first discuss the skeleton that all of the studied LRTA*-type real-time search methods fit, then we discuss the LRTA*-type real-time search methods that we study and analyze their performance. Finally, we discuss the two domain properties. These domain properties are whether domains are Eulerian (a superset of undirected domains) and whether the product of their number of actions and the maximal goal distance is small compared to that of other domains with the same number of states. For example, sliding-tile puzzles have both of these properties. We show that these properties simplify the search tasks for the LRTA*-type real-time search methods studied. These results can help to distinguish easy LRTA*-type real-time search tasks from hard ones, which can help experimental researchers to decide when to use LRTA*-type real-time search methods. In subsequent sections, we apply our insights to choosing test-beds for LRTA*-type real-time search methods and representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods. Some of our results have also been used to study extensions of reinforcement-learning methods. For example, they are used by Lin [Lin, 1993] to study "Hierarchical Q-Learning," that he applied to learning robot control.

2.5.1 Skeleton of LRTA*-Type Real-Time Search Methods

Initially, $memory = 0$ and $q(s, a) = 0$ for all $s \in S$ and $a \in A(s)$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. Choose an action a from $A(s)$ possibly using $memory$ and $q(s, a')$ for $a' \in A(s)$.
4. Update $memory$ and $q(s, a)$ possibly using $memory$, $q(s, a)$, and $q(succ(s, a), a')$ for $a' \in A(succ(s, a))$.
5. Execute action a, that is, change the current state to $succ(s, a)$.
6. $s :=$ the current state.
7. Go to 2.

Figure 2.27: Skeleton of the Studied LRTA*-Type Real-Time Search Methods

All of the LRTA*-type real-time search methods that we study fit the skeleton from Figure 2.27. They can be characterized as uninformed revolving LRTA*-type real-time search methods with minimal look-ahead and greedy action selection that solve suboptimal one-shot planning tasks in deterministic domains. They are *uninformed*, because they do not have any initial domain knowledge. They are *revolving*, because they repeat the same planning procedure after every action execution. They have *minimal look-ahead* (Figure 2.4), because they use only their memory and the information local to the current state to determine which action to execute. For example, different from LRTA*, all LRTA*-type real-time search methods that fit our skeleton do not even project one action execution ahead. Thus, their look-ahead is truly minimal, even smaller than that of LRTA* with look-ahead one. This means that they do not need to learn an action model of the domain, which makes them applicable to situations that LRTA* and other LRTA*-type real-time search methods with larger than minimal look-aheads cannot handle, including situations where the action model is not known in advance and thus the successor state of an action cannot be predicted before the action has been executed at least once.

The LRTA*-type real-time search methods that fit our skeleton associate a small amount of information with the actions that allows them to remember where they have already searched. In particular, they associate a *q-value* $q(s, a)$ with each action $a \in A(s)$ that can be executed in state $s \in S$. The term "q-values" [Watkins, 1989] is commonly used in reinforcement learning (Section 2.5.6) for values that are associated with actions. An additional value is maintained across action executions in the variable $memory$. Many of the studied LRTA*-type real-time search methods are *memoryless* [Koenig, 1992], meaning that they do not use the variable $memory$. The q-values and the memory are updated as the search progresses and used to determine which actions to execute. Their semantics depend on the specific LRTA*-type real-time search method used, but all values are zero-initialized, reflecting that the LRTA*-type real-time search methods are initially uninformed.

The LRTA*-type real-time search methods that fit our skeleton consist of a termination-checking step (Line 2), an action-selection step (Line 3), a value-update step (Line 4), and an action-execution step (Line 5). Their action-selection and value-update steps can differ. They first check whether they have already reached a goal state and thus can terminate successfully (Line 2). If not, they decide which action a to execute in the current state s (Line 3). For this decision, they can consult the value stored in their memory and the q-values associated with the actions in the current state. Then, they update the q-value of the selected action and their memory, possibly also using the q-values associated with the actions in their new state (Line 4). Finally, they execute the selected action (Line 5), update the current state (Line 6), and iterate the procedure (Line 7).

2.5.2 Example LRTA*-Type Real-Time Search Methods and Their Complexity

In this section, we discuss some LRTA*-type real-time search methods that fit our skeleton and analyze their complexity. In later sections, we discuss additional LRTA*-type real-time search methods that fit our skeleton. We use three kinds of domains in the complexity analysis: reset state spaces, quick-sand state spaces, and "complex state spaces." These domains have in common that one has to choose the correct action $n - 2$ times in a row to reach the goal state. If one executes an action that is not on the optimal path to the goal state, one

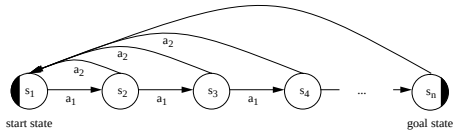


Figure 2.28: Reset State Space

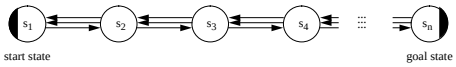


Figure 2.29: Quicksand State Space

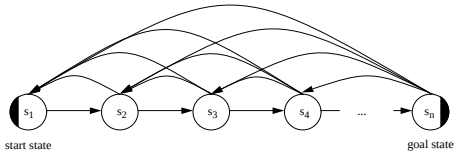


Figure 2.30: "Complex State Space"

ends up further away from the goal state.

A *reset state space* is shown in Figure 2.28. This is a domain in which all states (but the start state) have an action that leads back to the start state. We say that the action "resets" the LRTA*-type real-time search methods to the start state. A *quicksand state space* is shown in Figure 2.29. This is a domain in which all states (but the boundary states) have two actions that move the agent one action execution away from the goal state ("a bit into the quicksand") but only one action that moves it one action execution towards the goal state ("a bit out of the quicksand"). Quicksand state spaces differ from reset state spaces in the effort that is needed to recover from mistakes: all actions in quicksand state spaces have local effects only (it is possible to recover with only one action execution), whereas reset actions in reset state spaces do not have local effects. A *"complex state space"* is shown in Figure 2.30. This is a domain in which all states (but the start state) have several actions that lead back towards the start state. This includes actions that reset LRTA*-type real-time search methods to the start state and actions that move them only one action execution away from the goal state.

We first study the complexity of LRTA*-type real-time search methods that fit our skeleton over all domains. Let x be a measure for the task size. We are interested in two such measures: the product of the number of actions and the maximal goal distance ($x = ed$) and the number of states ($x = n$). When studying the complexity of LRTA*-type real-time search methods over all domains, one can freely choose the domain that maximizes the number of action executions for a given LRTA*-type real-time search method from all domains with the same x. Later, we restrict the possible choices and study the complexity of LRTA*-type real-time search methods over a subset of all domains. In this case, one can choose the domain that maximizes the number of action executions only from all domains with the same x that are contained in the subset. We are interested in the complexity of both efficient and inefficient LRTA*-type real-time search methods.

To be able to express the complexity of LRTA*-type real-time search methods in terms of n only, we often make the assumption that the domains are reasonable. *Reasonable domains* are safely explorable domains with $e \leq n^2$ (or, more generally, domains whose number of actions does not grow faster than the number of states squared). This assumption allows us to compare different LRTA*-type real-time search methods on the same scale. For example, different LRTA*-type real-time search methods have different upper complexity bounds, including $O(e)$, $O(nd)$, and $O(ed)$. Since $d \leq n - 1$ and $e \leq n^2$ for reasonable domains, we can derive upper complexity bounds from them that depend on n only, namely, $O(n^2)$, $O(n^2)$, and $O(n^3)$,

respectively. This allows us to compare the bounds better. It is reasonable to assume that domains are reasonable (which explains their name). For example, consider deterministic domains with no duplicate actions. In these domains, the execution of any two actions in the same state results in different successor states, which implies $e \leq n^2$. This includes reset state spaces and "complex state spaces." It also includes sliding-tile puzzles and grid-worlds. Even quicksand state spaces, that have duplicate actions, satisfy $e \leq n^2$.

2.5.2.1 Inefficient LRTA*-Type Real-Time Search Methods

In this section, we study the complexity of inefficient LRTA*-type real-time search methods that fit our skeleton. Notice that "the worst LRTA*-type real-time search method" does not exist, since one can construct LRTA*-type real-time search methods that perform arbitrarily badly, even if they fit our skeleton. They could, for example, deliberately avoid a goal state for an arbitrarily long time and only then move to a goal state. This problem should be addressed by performing a complexity analysis over a suitably defined class of "reasonable" LRTA*-type real-time search methods. In this section, however, we are content with studying examples of inefficient "reasonable" LRTA*-type real-time search methods.

Particularly bad LRTA*-type real-time search methods are ones that do not remember where they have already searched. Random walks are examples of such search methods. Strictly speaking, they are only real-time search methods but not LRTA*-type real-time search methods according to our classification in Figure 2.4. For convenience, we refer to them in the following as LRTA*-type real-time search methods because their deterministic counterpart, Edge Counting, is an LRTA*-type real-time search method. We illustrate that both random walks and Edge Counting are intractable (that is, not of polynomial complexity in the number of states) in reset state spaces, quicksand state spaces, and "complex state spaces."

Random Walks: Random walks always choose randomly from the actions available in the current state.

Random Walks	(see Figure 2.27)
action-selection step (Line 3)	a := pick an action from $A(s)$ with uniform probability
value-update step (Line 4)	(empty)

Random walks have look-ahead zero. They are memoryless LRTA*-type real-time search methods that do not store any information at all, and thus cannot remember where they have already searched unsuccessfully. As a consequence, the number of action executions that they need to reach a goal state can exceed any given bound with positive probability, implying an infinite complexity. However, the probability that they reach the goal state within a given number of action executions as the bound approaches infinity approaches one over all safely explorable domains. Furthermore, the average number of action executions that random walks need to reach a goal state is finite over all safely explorable domains, although it can be exponential in the number of states.

Remember that we consider only domains with a finite number of states in this thesis. As an aside, notice that the probability that random walks reach the goal state in *infinite* one- or two-dimensional (but not higher-dimensional) grid-worlds within a given number of action executions approaches one as the bound approaches infinity [Feller, 1966]; none of the other zero-initialized LRTA*-type real-time search methods studied in this chapter can guarantee this.

Theorem 9 *Random walks have infinite complexity. Their average-case performance is finite over all safely explorable, deterministic domains, but is at least exponential in n (even over all reasonable domains).*

Proof: It is easy to see that random walks have infinite complexity (just consider a domain with a cycle).

It is also easy see that random walks have finite average-case performance over all safely explorable domains: For every state s of the domain, consider a shortest path from it to a goal state. Let $y_s < \infty$ be the length of this path measured in action executions and $p_s > 0$ the probability that it is traversed by a random walk

that starts in s. Then, an upper bound on the average-case performance of random walks in this domain is $\max_{s \in S} y_s / \min_{s \in S} p_s < \infty$ no matter what the start state is.

More generally, the average-case performance of random walks for a safely explorable domain can be calculated as follows. For every state s, one introduces a variable x_s that represents the average number of action executions that random walks need to reach a goal state from state s. These values can be calculated by solving the following set of linear equations:

$$x_s = \begin{cases} 0 & \text{if } s \in G \\ 1 + \frac{1}{|A(s)|} \sum_{a \in A(s)} x_{succ(s,a)} & \text{otherwise} \end{cases} \qquad \text{for all } s \in S.$$

To solve these equations in closed form one usually uses generating functions. Feller [Feller, 1966], for example, gives a mathematical derivation and Whitehead [Whitehead, 1992] discusses an application in the context of reinforcement learning.

(Lower Bound) Reset state spaces (Figure 2.28) are examples of reasonable domains for which the number of action executions that random walks need on average to reach a goal state is at least exponential in n. We solve the following set of linear equations:

$$\begin{aligned} x_1 &= 1 + x_2 \\ x_s &= 1 + 0.5x_1 + 0.5x_{s+1} \qquad \text{for all } s \in \{2, 3, \ldots, n-1\} \\ x_n &= 0 \end{aligned}$$

They can be solved for $x_{s_{start}}$ as follows (without generating functions):

$$\begin{aligned} x_{s_{start}} &= x_1 \\ &= 1 + x_2 \\ &= 1 + 1 + 0.5x_1 + 0.5x_3 \\ &= 1 + (1 + 0.5) + 0.5x_1(1 + 0.5) + 0.5^2 x_4 \\ &= 1 + (1 + 0.5 + 0.5^2) + 0.5x_1(1 + 0.5 + 0.5^2) + 0.5^3 x_5 \\ &= \ldots \\ &= 1 + \sum_{i=0}^{s-3} 0.5^i + 0.5x_1 \sum_{i=0}^{s-3} 0.5^i + 0.5^{s-2} x_s \qquad \text{for all } s \in \{2, 3, \ldots, n\} \\ &= \ldots \\ &= 1 + \sum_{i=0}^{n-3} 0.5^i + 0.5x_1 \sum_{i=0}^{n-3} 0.5^i + 0.5^{n-2} x_n \\ &= 1 + \frac{1 - 0.5^{n-2}}{1 - 0.5} + 0.5x_1 \frac{1 - 0.5^{n-2}}{1 - 0.5} + 0.5^{n-2} \times 0 \\ &= 3 \times 2^{n-2} - 2 \end{aligned}$$

Thus, random walks execute on average $x_{s_{start}} = 3 \times 2^{n-2} - 2$ actions before they reach the goal state (for $n \geq 2$).

Quicksand state spaces (Figure 2.29) are another kind of reasonable domain for which the number of action executions that random walks need on average to reach a goal state is at least exponential in n. They bias

random walks to move away from the goal state: In every state (but the boundary states) random walks move away from the goal state with probability $2/3$ and towards the goal state with only probability $1/3$. We proceed as we did for reset state spaces and have to solve the following set of linear equations:

$$\begin{aligned} x_1 &= 1 + x_2 \\ x_s &= 1 + 2/3\, x_{s-1} + 1/3\, x_{s+1} \qquad \text{for all } s \in \{2, 3, \ldots, n-1\} \\ x_n &= 0 \end{aligned}$$

The result is that random walks execute on average $x_{s_{start}} = 2^{n+1} - 3n - 1$ actions before they reach the goal state (for $n \geq 1$).

Finally, "complex state spaces" (Figure 2.30) are a third kind of reasonable domain for which the number of action executions that random walks need on average to reach a goal state is at least exponential in n. In fact, random walks execute more than $(n-1)!$ actions on average before they reach the goal state (for $n \geq 2$). ■

Edge Counting: We derive a memoryless LRTA*-type real-time search method that shares many properties with random walks, but has finite complexity – basically, by "removing the randomness" from random walks. The q-values approximate the negative number of times the actions have been executed.

Edge Counting	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of arg $\max_{a' \in A(s)} q(s, a')$
value-update step (Line 4)	$q(s, a) := -1 + q(s, a)$

Random walks execute all actions in a state equally often in the long run. The action-selection step of Edge Counting always chooses the action for execution that has been executed the fewest number of times. This achieves the same result as random walks, but in a deterministic way. One particular tie-breaking rule, for example, is to execute all actions in turn. Shannon used this method as early as in the late 1940's to implement an exploration behavior for an electronic mouse that searched a maze [Sutherland, 1969]. To the best of our knowledge, however, its relationship to random walks has never been pointed out nor has its complexity been studied. The number of action executions that Edge Counting needs to reach a goal state is bounded from above in safely explorable domains, implying a finite complexity, but the complexity is at least exponential in the number of states.

Theorem 10 *The complexity of Edge Counting is finite over all safely explorable, deterministic domains, but is at least exponential in n (even over all reasonable domains).*

Proof: The argument that Edge Counting reaches a goal state eventually in safely explorable domains is by contradiction. If Edge Counting did not reach a goal state eventually, then there must be some cycle. Since the domain is safely explorable, there must be some way out of the cycle. We show that Edge Counting eventually executes an action that takes it out of the cycle, which is a contradiction: If Edge Counting did not reach a goal state eventually, it would execute actions forever. In this case, there is a time t from which on Edge Counting only executes those actions that it executes infinitely often. Eventually, the q-values of these actions drop below any given bound, since – every time an action is executed – its q-value is decremented by one. In particular, they drop below the q-value of an action that Edge Counting considers infinitely often for execution, but never executes after time t. Such an action exists, since in safely explorable domains one can reach a goal state from every state. Then, however, Edge Counting is forced to execute this action after time t, which is a contradiction.

(Lower Bound) Reset state spaces (Figure 2.28) are examples of reasonable domains for which the number of action executions that Edge Counting needs in the worst case to reach a goal state is at least exponential in n. In particular, Edge Counting traverses the state sequence that is printed by `f(n)` if ties are broken in favor of successor states with smaller indices.

```
proc f(i) =
  if i = 2 then
    print 1
  else
    f(i-1)
    f(i-1)
  print i
```

In this case, Edge Counting executes $3 \times 2^{n-2} - 2$ actions before it reaches the goal state (for $n \geq 2$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_1, s_2, s_3, s_1, s_2, s_1, s_2, s_3, s_4, s_1, s_2, s_1, s_2, s_3, s_1, s_2, s_1, s_2, s_3, s_4, and s_5.

Quicksand state spaces (Figure 2.29) are another kind of reasonable domain for which the number of action executions that Edge Counting needs in the worst case to reach a goal state is at least exponential in n. In particular, Edge Counting traverses the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller indices:

```
print 1
print 2
for i := 3 to n
  print i-2
  f(i-1)
  print i-2
  f(i-1)
  print i
```

where

```
proc f(i) =
  if i = 2 then
    print 2
  else
    print i-2
    f(i-1)
    print i-2
    f(i-1)
    print i
```

In this case, Edge Counting executes $2^{n+1} - 3n - 1$ actions before it reaches the goal state (for $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_4, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_4, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_4, and s_5.

Finally, "complex state spaces" (Figure 2.30) are a third kind of reasonable domain for which the number of action executions that Edge Counting needs in the worst case to reach a goal state is at least exponential in n. In fact, Edge Counting executes more than $(n-1)!$ actions in the worst case before it reaches the goal state if ties are broken in favor of successor states with smaller indices (for $n \geq 2$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_1, s_2, s_3, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_3, s_4, s_1, s_2, s_1, s_2, s_3, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_3, s_4, s_2, s_1, s_2, s_3, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_3, s_4, s_3, s_1, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_3, s_4, and s_5. ■

To summarize, the number of action executions that Edge Counting needs in the worst case to reach a goal state is at least exponential in the number of states for reset state spaces (Figure 2.28), quicksand state spaces (Figure 2.29), and "complex state spaces" (Figure 2.30). Similarly, the number of action executions that random walks need on average to reach a goal state is at least exponential in the number of states for these domains. Still, we expect some improvement in experimental average-case performance when switching from random walks to Edge Counting, since Edge Counting remembers something about where it has already searched. It is possible, however, that the improvement is just a constant factor. Sections 2.5.3.4 and 2.5.3.5 contain experimental results that show that, indeed, the advantage of Edge Counting over random walks can be small.

Initially, the q-values $q(s,a)$ are approximations of $-1 - gd(succ(s,a))$ for all $s \in S \setminus G$ and $a \in A(s)$ and zero for $s \in G$ and $a \in A(s)$. At every point in time, $u(s) = \max_{a \in A(s)} q(s,a)$ for all $s \in S$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. $a :=$ one-of $\arg\max_{a' \in A(s)} q(s,a')$.
4. $q(s,a) := \min(q(s,a), -1 + u(succ(s,a)))$.
5. Execute action a, that is, change the current state to $succ(s,a)$.
6. $s :=$ the current state.
7. Go to 2.

Figure 2.31: Min-LRTA*

2.5.2.2 Efficient LRTA*-Type Real-Time Search Methods: Min-LRTA*

No LRTA*-type real-time search method that fits our skeleton can distinguish between actions in nongoal states before it has executed them it least once because all q-values are identical initially. This implies the following lower bound on their complexity:

Theorem 11 *The complexity of every LRTA*-type real-time search method that fits our skeleton is at least $O(ed)$ action executions over all deterministic domains. Furthermore, its complexity is at least $O(n^3)$ action executions over all reasonable, deterministic domains.*

Proof: (Lower Bound) "Complex state spaces" (Figure 2.30) are examples of reasonable domains for which the number of action executions that every LRTA*-type real-time search method that fits our skeleton needs in the worst case to reach a goal state is at least $O(ed)$ or, alternatively, $O(n^3)$. It has to execute each of the $O(n^2)$ actions in nongoal states that lead away from the goal state at least once in the worst case. In each of these cases, it has to execute $O(n)$ actions on average to recover from the action, for a total of $O(n^3)$ actions. In particular, it can traverse either the state sequence that is printed by the following program in pseudo code or a super sequence thereof if ties are broken in favor of successor states with smaller indices:

```
for i := 1 to n-1
  print i
  for j := 1 to i-1
    for k := j to i
      print k
print n
```

In this case, it executes at least $n^3/6 - n/6$ actions before it reaches the goal state (for $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_1, s_2, s_1, s_2, s_3, s_1, s_2, s_3, s_2, s_3, s_4, s_1, s_2, s_3, s_4, s_2, s_3, s_4, s_3, s_4, and s_5. Notice that $e = n^2/2 + n/2 - 1$ (for $n \geq 1$) and $d = n - 1$ (for $n \geq 1$) for the domain in Figure 2.30. Thus, the complexity is both at least $O(ed)$ and $O(n^3)$ action executions. ■

Thus, every LRTA*-type real-time search method that fits our skeleton has a complexity of at least $O(ed)$ action executions or, over all reasonable domains, $O(n^3)$ action executions. There are indeed LRTA*-type real-time search methods that fit our skeleton and have at most this complexity. An example is Min-LRTA*, a variant of Min-Max LRTA* that is related to Q-Learning (Section 2.5.6). In the following, we first discuss Min-LRTA* in general. Later, we study zero-initialized Min-LRTA*, that fits our skeleton.

Min-LRTA*: *LRTA* with minimal look-ahead* (Min-LRTA*) [Koenig and Simmons, 1996a] is a memoryless LRTA*-type real-time search method with minimal look-ahead (Figure 2.31). Its q-values approximate, for nongoal states, the negative value of the sum of one and the goal distance of the successor states of the actions. The action-selection step always greedily chooses the action with the maximal q-value in the current state.

Min-LRTA* is similar to Min-Max LRTA* with look-ahead one, and thus also to LRTA* with look-ahead one since we restrict our analysis to deterministic domains and both methods behave identically in deterministic

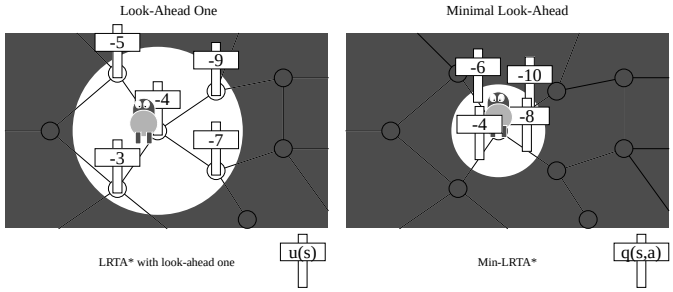


Figure 2.32: Difference between Min-Max LRTA* and Min-LRTA* in Deterministic Domains

domains. The only difference between Min-Max LRTA* with look-ahead one and Min-LRTA* is the following (Figure 2.32): Min-Max LRTA* associates u-values with states and looks at the u-values of the successor states of the current state to choose which action to execute. This is a look-ahead of one action execution. Min-LRTA*, on the other hand, associates q-values with actions and looks only at the q-values of the actions of the current state to choose which action to execute. This look-ahead is larger than zero, but smaller than one since Min-Max LRTA* does not even project one action execution ahead. The q-value of an action, however, only changes when the action is executed and can therefore be outdated: The q-value is a "local copy" of the u-value of the successor state since the q-value $q(s,a)$ is set to $-1+u(succ(s,a))$ when action a is executed in state s, assuming that $q(s,a) \geq -1+u(succ(s,a))$. However, the u-value of the successor state can change later if actions are executed in it. When Min-Max LRTA* has to decide which action to execute in state s it uses the new u-value of the successor state to evaluate action a. Min-LRTA*, on the other hand, uses the q-value of action a, that does not yet reflect the new u-value of the successor state. We therefore expect the performance of Min-LRTA* to be worse than that of Min-Max LRTA*.

In the following, we show how to transfer the complexity results about Min-Max LRTA* with look-ahead one to Min-LRTA*. Assume that Min-LRTA* operates in a domain with states S, start state s_{start}, goal states G, actions $A(s)$ for $s \in S$, transition function $succ$, and initial q-values $q(s,a)$ for $s \in S$ and $a \in A(s)$. We refer to this domain in the following as the *original domain*. We transform the domain so that Min-Max LRTA* with look-ahead one in the *transformed domain* and Min-LRTA* in the original domain behave identically. The transformed domain is larger than the original domain. Thus, we do not contradict our earlier statement that, in the same domain, we expect the performance of Min-LRTA* to be worse than that of Min-Max LRTA*.

The transformation is the following:

$$
\begin{array}{rcll}
\overline{S} & := & \{s_{s,a} | s \in S, a \in A(s)\} & \\
\overline{s}_{start} & := & s_{s_{start},a} & \text{where } a = \text{one-of arg max}_{a' \in A(s_{start})}\, q(s_{start}, a') \\
\overline{G} & := & \{s_{s,a} | s \in G, a \in A(s)\} & \\
\overline{A}(\overline{s}) & := & A(succ(s,a)) & \text{for all } \overline{s} = s_{s,a} \in \overline{S} \\
\overline{succ}(\overline{s}, \overline{a}) & := & s_{succ(s,a),\overline{a}} & \text{for all } \overline{s} = s_{s,a} \in \overline{S} \text{ and } \overline{a} \in \overline{A}(\overline{s}),
\end{array}
$$

where the $s_{s,a}$ are new states. Basically, the actions of the original domain become the states of the transformed domain. The possible successor states of such an *action state* $s_{s,a}$ are the action states that correspond to the actions that can be executed immediately after the execution of action a in state s. Figure 2.33 shows an example. Notice that every action execution in a nongoal state of the transformed domain necessarily results in a state change if this is true for the original domain. Theorem 12 shows that the maximal goal distance $\overline{d}$ of the transformed domain is at most one larger than the maximal goal distance d of the original domain. Thus, if the original domain is safely explorable ($d < \infty$), the transformed domain is safely explorable as well. Theorems 12 and 13 together show that the maximal goal distance $\overline{d}$ of the transformed domain equals

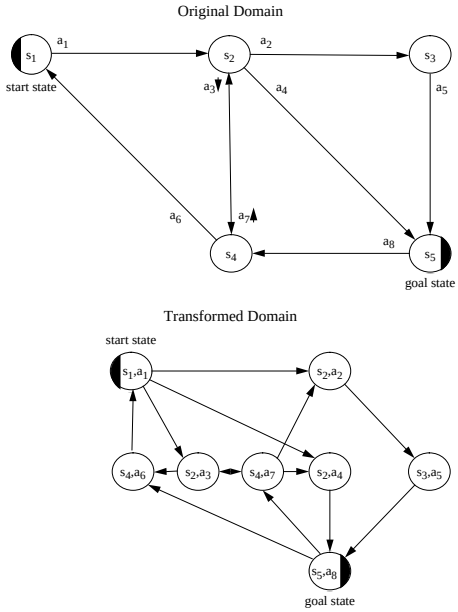


Figure 2.33: Domain and its Transformation

either the maximal goal distance d of the original domain or $d+1$. Figures 2.34 and 2.35 show that indeed both $\overline{d} = d$ and $\overline{d} = d+1$ are possible.

Theorem 12 *Consider any domain and its transformation. Then, $\overline{d} \leq d+1$, where d is the maximal goal distance of the original domain and $\overline{d}$ is the maximal goal distance of its transformation.*

Proof: The theorem holds for $d = \infty$. Now assume that $d < \infty$. We show that $\overline{gd}(s_{s,a}) \leq gd(succ(s,a)) + 1$ for all $s_{s,a} \in \overline{S}$. This trivially holds for $s_{s,a} \in \overline{G}$. Consider an arbitrary $s_{s,a} \in \overline{S} \setminus \overline{G}$ and a shortest path in the original domain from $succ(s,a)$ to a closest goal state $s' \in G$. Its length is $gd(succ(s,a)) < \infty$. Now consider the corresponding path in the transformed domain from $s_{s,a}$ to goal state $s_{s',a'} \in \overline{G}$, where $a' \in A(s')$ is arbitrary. Its length is $gd(succ(s,a)) + 1$, which is at least $\overline{gd}(s_{s,a})$. Then, $\overline{d} = \max_{s_{s,a} \in \overline{S}} \overline{gd}(s_{s,a}) \leq \max_{s_{s,a} \in \overline{S}} [gd(succ(s,a)) + 1] \leq \max_{s \in S} gd(s) + 1 = d + 1$. ■

Theorem 13 *Consider any domain and its transformation. Then, $d \leq \overline{d}$, where d is the maximal goal distance of the original domain and $\overline{d}$ is the maximal goal distance of its transformation.*

Proof: We show that $gd(s) \leq \overline{gd}(s_{s,a})$ for all $s_{s,a} \in \overline{S}$. Consider an arbitrary $s_{s,a} \in \overline{S}$ and a shortest path in the transformed domain from $s_{s,a}$ to a closest goal state $s_{s',a'} \in \overline{G}$. Its length is $\overline{gd}(s_{s,a}) \leq \infty$. Now consider the corresponding path in the original domain from s to goal state $s' \in G$. Its length is also $\overline{gd}(s_{s,a})$, which is at least $gd(s)$. Then, $d = \max_{s \in S} gd(s) = \max_{s \in S, a \in A(s)} gd(s) \leq \max_{s_{s,a} \in \overline{S}} \overline{gd}(s_{s,a}) = \overline{d}$, because we assume throughout this chapter that $A(s) \neq \emptyset$ for all $s \in S$. ■

Theorem 14 shows that Min-Max LRTA* with look-ahead one in the transformed domain and Min-LRTA* in the original domain behave identically if initially $u(s_{s,a}) = q(s,a)$ for all $s \in S$ and $a \in A(s)$ and ties are broken identically (provided that, as we assume throughout this chapter, every action execution in a nongoal state necessarily results in a state change). For all times $t = 0, 1, 2, \ldots$ (until termination), the states of Min-Max LRTA* correspond to the actions chosen by Min-LRTA* and the u-values of Min-Max LRTA*

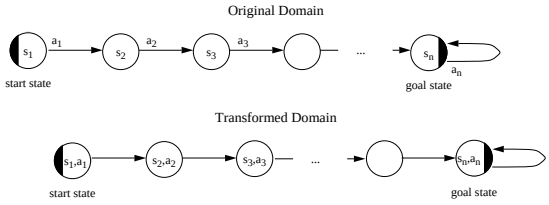


Figure 2.34: Domain with $d = n - 1$ and its Transformation with $\overline{d} = d$

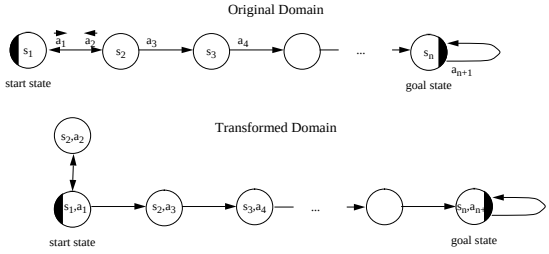


Figure 2.35: Domain with $d = n - 1$ and its Transformation with $\overline{d} = d + 1$

equal the q-values of Min-LRTA*. The time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update steps of Min-Max LRTA* (Line 4 in Figure 2.9) and Min-LRTA* (Line 4 in Figure 2.31).

Theorem 14 *For all times $t = 0, 1, 2, \ldots$ (until termination), $\overline{s}^t = s_{s^t,a^t}$ and $u^t(s_{s,a}) = q^t(s,a)$ for all $s \in S$ and $a \in A(s)$ provided that $u^0(s_{s,a}) = q^0(s,a)$ for all $s \in S$ and $a \in A(s)$ and ties are broken identically.*

Proof by induction: The theorem holds for $t = 0$. Now assume it also holds at time t. First, we prove that $\overline{a}^t = a^{t+1}$ if ties are broken identically.

$$
\begin{aligned}
\overline{a}^t &= \text{one-of arg} \max_{a \in \overline{A}(\overline{s}^t)} u^t(\overline{succ}(\overline{s}^t, a)) \\
&= \text{one-of arg} \max_{a \in \overline{A}(s_{s^t,a^t})} u^t(\overline{succ}(s_{s^t,a^t}, a)) \\
&= \text{one-of arg} \max_{a \in \overline{A}(s_{s^t,a^t})} u^t(s_{succ(s^t,a^t),a}) \\
&= \text{one-of arg} \max_{a \in A(succ(s^t,a^t))} q^t(succ(s^t,a^t),a) \\
&= \text{one-of arg} \max_{a \in A(s^{t+1})} q^t(s^{t+1},a) \\
&= \text{one-of arg} \max_{a \in A(s^{t+1})} q^{t+1}(s^{t+1},a) \qquad (2.2) \\
&= a^{t+1}.
\end{aligned}
$$

The transformation to Line 2.2 is valid since $q(s^t, a^t)$ is the only q-value of Min-LRTA* that changes between time t and $t+1$. Since all actions result in state changes, it holds that $s^t \neq s^{t+1}$ and therefore $q^t(s^{t+1}, a) = q^{t+1}(s^{t+1}, a)$ for all $a \in A(s^{t+1})$.

Next, we prove that $u^{t+1}(\overline{s}^t) = q^{t+1}(s^t, a^t)$. It holds that

$$\begin{aligned}
u^t(\overline{succ}(\overline{s}^t, \overline{a}^t)) &= u^t(\overline{succ}(s_{s^t,a^t}, a^{t+1})) \\
&= u^t(s_{succ(s^t,a^t),a^{t+1}}) \\
&= q^t(succ(s^t, a^t), a^{t+1}) \\
&= q^t(s^{t+1}, a^{t+1}) \\
&= q^{t+1}(s^{t+1}, a^{t+1}) \qquad (2.3) \\
&= \max_{a \in A(s^{t+1})} q^{t+1}(s^{t+1}, a) \\
&= u^{t+1}(s^{t+1}) \\
&= u^t(s^{t+1}) \qquad (2.4) \\
&= u^t(succ(s^t, a^t)). \qquad (2.5)
\end{aligned}$$

The transformations to Lines 2.3 and 2.4 are valid since all actions result in state changes and therefore $q^t(s^{t+1}, a^{t+1}) = q^{t+1}(s^{t+1}, a^{t+1})$ and $u^{t+1}(s^{t+1}) = u^t(s^{t+1})$. It also holds that

$$u^t(\overline{s}^t) = u^t(s_{s^t,a^t}) = q^t(s^t, a^t). \qquad (2.6)$$

Consequently,

$$\begin{aligned}
u^{t+1}(\overline{s}^t) &= \min(u^t(\overline{s}^t), -1 + u^t(\overline{succ}(\overline{s}^t, \overline{a}^t))) \\
&= \min(q^t(s^t, a^t), -1 + u^t(succ(s^t, a^t))) \\
&= q^{t+1}(s^t, a^t).
\end{aligned}$$

The only u-value of Min-Max LRTA* that changes between time t and $t+1$ is $u(\overline{s}^t)$ and the only q-value of Min-LRTA* that changes between time t and $t+1$ is $q(s^t, a^t)$. Since $u^{t+1}(\overline{s}^t) = q^{t+1}(s^t, a^t)$, it holds that $u^{t+1}(s_{s,a}) = q^{t+1}(s, a)$ for all $s \in S$ and $a \in A(s)$.

Finally, we prove that $\overline{s}^{t+1} = s_{s^{t+1},a^{t+1}}$.

$$\overline{s}^{t+1} = \overline{succ}(\overline{s}^t, \overline{a}^t) = \overline{succ}(s_{s^t,a^t}, a^{t+1}) = s_{succ(s^t,a^t),a^{t+1}} = s_{s^{t+1},a^{t+1}}. \blacksquare$$

Thus, both Min-LRTA* and Min-Max LRTA* need the same number of action executions to reach a goal state since $\overline{s}^t = s_{s^t,a^t} \in \overline{G}$ if and only if $s^t \in G$. This can be used to transfer the complexity results about Min-Max LRTA* with look-ahead one to Min-LRTA*. For example, the complexity of Min-Max LRTA* with initially admissible u-values is at most $O(\overline{n}\overline{d})$ action executions in the transformed domain (Theorem 4). Since $\overline{n}$ equals e and $\overline{d}$ equals either d or $d+1$, $O(\overline{n}\overline{d})$ equals $O(ed)$. Thus, the complexity of Min-LRTA* is at most $O(ed)$ action executions in the original domain.

We study zero-initialized Min-LRTA* in the following because it fits our skeleton. The value-update step of zero-initialized Min-LRTA* can be simplified. To understand why, consider Min-Max LRTA* with look-ahead one in the transformed domain. Its initial u-values are zero and thus consistent. The following argument shows that the value-update step of Min-LRTA* can be simplified whenever the corresponding u-values of Min-Max LRTA* are consistent: Consistent u-values of Min-Max LRTA* remain consistent according to

Theorem 2. Consequently, $u^t(\overline{s}^t) \geq -1 + \max_{a \in A(\overline{s}^t)} u^t(\overline{succ}(\overline{s}^t, a)) = -1 + u^t(\overline{succ}(\overline{s}^t, \overline{a}^t))$, since $\overline{a}^t = \text{one-of arg}\max_{a \in A(\overline{s}^t)} u^{t+1}(\overline{succ}(\overline{s}^t, a)) = \text{one-of arg}\max_{a \in A(\overline{s}^t)} u^t(\overline{succ}(\overline{s}^t, a))$. Then,

$$\begin{array}{rcl} -1 + u^t(succ(s^t, a^t)) & \stackrel{\text{Formula 2.5}}{=} & -1 + u^t(\overline{succ}(\overline{s}^t, \overline{a}^t)) \\ & \leq & u^t(\overline{s}^t) \\ & \stackrel{\text{Formula 2.6}}{=} & q^t(s^t, a^t). \end{array}$$

Consequently, the value-update step of Min-LRTA* can be simplified to $q(s, a) := -1 + u(succ(s, a))$. We refer to this value-update step as the *simplified value-update step* of Min-LRTA*. The following table presents the action-selection and simplified value-update steps of Min-LRTA*. Its q-values approximate, for nongoal states, the negative value of the sum of one and the goal distance of the successor states of the actions.

Min-LRTA*	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of arg $\max_{a' \in A(s)} q(s, a')$
value-update step (Line 4)	$q(s, a) := -1 + u(succ(s, a)) = -1 + \max_{a' \in A(succ(s,a))} q(succ(s, a), a')$

Theorem 15 follows from the above analysis of Min-LRTA*:

Theorem 15 *Zero-initialized Min-LRTA* has a tight complexity of $O(ed)$ action executions over all deterministic domains. Furthermore, it has a tight complexity of $O(n^3)$ action executions over all reasonable, deterministic domains.*

Proof: (Upper Bound) The complexity of Min-LRTA* is at most $O(ed)$ action executions, as argued above. Since $e \leq n^2$ and $d \leq n - 1$ for reasonable domains, it follows that its complexity is at most $O(n^3)$ action executions over all reasonable domains.

(Lower Bound) According to Theorem 11, the complexity of every LRTA*-type real-time search method that fits our skeleton is at least $O(ed)$ action executions or, over all reasonable domains, $O(n^3)$ action executions. This includes zero-initialized Min-LRTA*. ■

Thus, Min-LRTA* has a tight complexity of $O(ed)$ action executions or, over all reasonable domains, $O(n^3)$ action executions. Notice that e can grow faster than n. In reasonable domains, for example, e can grow as fast as n^2. Thus, as expected, the complexity of Min-LRTA*, namely, $O(ed)$, is, in general, worse than the complexity of Min-Max LRTA* with look-ahead one, namely, $O(nd)$. However, we have shown that no LRTA*-type real-time search method that fits our skeleton can have a smaller big-O complexity than Min-Max LRTA*. There exist variants of Min-LRTA* whose performance dominates the performance of Min-LRTA* [Koenig and Simmons, 1993b] but they also have a tight complexity of $O(ed)$ action executions or, over reasonable domains, $O(n^3)$ action executions.

2.5.3 Undirected Domains and Other Eulerian Domains

In this section, we study the complexity of LRTA*-type real-time search methods that fit our skeleton over all undirected domains and over all Eulerian domains. This is an interesting domain property because it affects the performance of LRTA*-type real-time search methods, but not the performance of traditional search methods. We first explain what Eulerian domains and undirected domains are, and then why every undirected domain is Eulerian but not every Eulerian domain is undirected.

A Eulerian tour (or, synonymously, Eulerian walk) on a graph is a path with the following property as defined by the Swiss mathematician Leonhard Euler when he considered whether the seven Königsberg bridges could

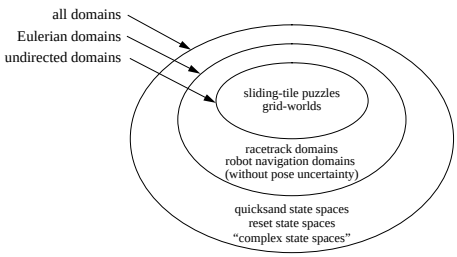


Figure 2.36: Subsets of Domains (including Example Domains)

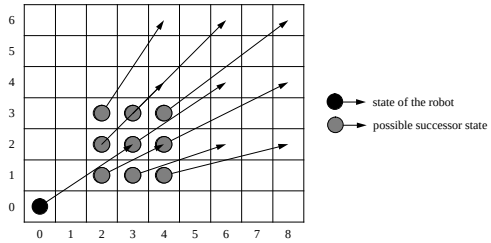


Figure 2.37: Actions in the Racetrack Domain

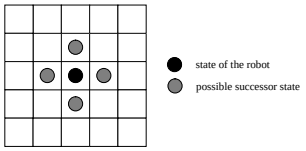


Figure 2.38: Actions in the Grid-World

be traversed without recrossing any of them [Newman, 1953]: its start vertex equals its end vertex and it contains all of the edges of the graph exactly once [Chartrand and Lesniak, 1986]. A Eulerian graph is a graph that contains a Eulerian tour. This implies for undirected graphs that the degree of every vertex is even. It implies for directed graphs that the in-degree of every vertex equals its out-degree. Since deterministic domains are directed graphs, a Eulerian domain is a deterministic domain where there are as many actions that leave a state as there are actions that enter it.

Definition 3 *A domain is* Eulerian *if and only if it is deterministic and* $|A(s)| = |\{(s', a')|s' \in S \wedge a' \in A(s') \wedge succ(s', a') = s\}|$ *for all* $s \in S$.

Undirected domains do not correspond to undirected graphs but rather to bi-directed graphs, that is, directed graphs that can be obtained from undirected graphs by replacing every undirected edge with a pair of directed edges, one for each direction. This explains why all undirected domains are directed Eulerian graphs and thus Eulerian domains. Many search domains from artificial intelligence are undirected and thus Eulerian (Figure 2.36). Examples include sliding-tile puzzles and grid-worlds.

There also exist domains that are Eulerian, but not undirected. Examples are *racetrack domains* [Gardner, 1973]. They correspond to grid-worlds, but a state of the domain is not only characterized by the (x,y) coordinates of the square that the agent currently occupies. Instead, it is described by two pairs of integers: the square that the agent occupies, and its speed in both the x and y direction. Actions correspond to adjusting both the x and y speed components by -1, 0, or 1. Given an action (speed change), the successor state is determined by computing the new speed components (one can impose a limit that the absolute speeds are not

allowed to exceed) and then determining the new location of the agent by adding each speed component to its corresponding location component. The new location of the agent has to be on the grid and cannot be contained in an obstacle. An example is shown in Figure 2.37. In this example, the current (x,y) location of the agent is $(0, 0)$ and its speed is $(3, 2)$. The agent now adjusts the speed to one of $(2, 1)$, $(2, 2)$, $(2, 3)$, $(3, 1)$, $(3, 2)$, $(3, 3)$, $(4, 1)$, $(4, 2)$, or $(4, 3)$. If it chooses speed $(2, 1)$, then its successor location is $(0, 0) + (2, 1) = (2, 1)$ and its successor speed is $(2, 1)$. This action is permissible because the successor location is indeed on the grid and not contained in an obstacle. Racetrack domains are robot navigation domains that are more realistic than grid-worlds (Figure 2.38). For instance, they model acceleration and take into account that the turn radius of the robot gets larger at higher speeds. Racetrack domains are Eulerian except around obstacles or at boundaries. In particular, obstacle-free racetrack domains on a torus are Eulerian, but not undirected. Racetrack domains have been used as test-beds for LRTA*-type real-time search methods in [Barto *et al.*, 1995]. Another example is our robot navigation domain (Section 2.4) without pose uncertainty. Again, it is Eulerian except around obstacles or at boundaries. In particular, an obstacle-free robot navigation domain without pose uncertainty on a torus is Eulerian, but not undirected.

We study Min-LRTA* again and compare its complexity to the complexities of both the most efficient and less efficient LRTA*-type real-time search methods that fit our skeleton. As examples of less efficient LRTA*-type real-time search methods we use again random walks and Edge Counting. The complexity of LRTA*-type real-time search methods over a subset of all domains can potentially be smaller than their complexity over all domains. This is the case if the subset contains only domains on which the LRTA*-type real-time search methods perform well. Thus, Eulerian domains could be easier to search than domains in general, and undirected domains could be even easier to search. In the following, we show that Eulerian domains are indeed easier to search with the studied LRTA*-type real-time search methods than domains in general. In fact, even LRTA*-type real-time search methods that are intractable in general can be tractable in undirected domains and other Eulerian domains. Undirected domains do not simplify the search any further. This explains why we regard "being undirected" and "being Eulerian" as one property and not two different properties.

2.5.3.1 Inefficient LRTA*-Type Real-Time Search Methods

First, we study again the LRTA*-type real-time search methods that can be inefficient, namely, random walks and Edge Counting.

Random Walks: The complexity of random walks remains infinite over all undirected domains and over all Eulerian domains, but their average-case performance decreases over all reasonable, undirected domains and over all reasonable, Eulerian domains, from being at least exponential in n to being a small polynomial in n.

Theorem 16 *Random walks have infinite complexity over all undirected domains and over all Eulerian domains. Their average performance is tight at $O(ed)$ action executions over these domains. Furthermore, it is tight at $O(n^3)$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains.*

Proof: It is easy to see that random walks have infinite complexity over all undirected domains and over all Eulerian domains (just consider an undirected domain with a cycle).

(Upper Bound) It is known that e is an upper bound on the number of action executions that random walks need on average to reach a specified successor state of the current state in Eulerian domains [Aleliunas *et al.*, 1979]. We give a complete proof of this fact in [Koenig and Simmons, 1992]. Now consider a shortest path from the start state to a closest goal state. Since its length is at most d, $O(ed)$ is an upper bound on the number of action executions that random walks need on average to reach a goal state in Eulerian domains. Since $e \leq n^2$ and $d \leq n - 1$ for reasonable domains, the average-case performance of random walks is at most $O(n^3)$ action executions over all reasonable, Eulerian domains and thus also over all reasonable, undirected domains.

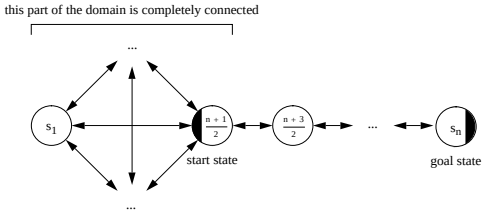


Figure 2.39: Undirected Domain

(Lower Bound) Figure 2.39 [Motwani and Raghavan, 1995] shows an example of a reasonable, undirected (and thus Eulerian) domain for which the number of action executions that random walks need on average to reach a goal state is at least $O(ed)$ or, alternatively, $O(n^3)$. As in the proof of Theorem 9, we can solve a set of linear equations to calculate the average performance of random walks in this domain. The result is that random walks execute $x_{s_{start}} = n^3/8 + n^2/8 - 5n/8 + 3/8$ actions on average before they reach the goal state (for odd $n \geq 1$). Notice that $e = n^2/4 + n - 5/4$ (for odd $n \geq 1$) and $d = n/2 + 1/2$ (for odd $n \geq 3$) for the domain in Figure 2.39. Thus, the complexity is both at least $O(ed)$ and $O(n^3)$ action executions. ■

Edge Counting: The average-case performance of random walks decreases over all reasonable, undirected domains and over all reasonable, Eulerian domains. The complexity of Edge Counting decreases as well, from being at least exponential in n to being a small polynomial in n.

Theorem 17 *Edge Counting has a tight complexity of $O(ed)$ action executions over all undirected domains and over all Eulerian domains. Furthermore, it has a tight complexity of $O(n^3)$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains.*

Proof: (Upper Bound) The complexity of Edge Counting is at most $e \times gd(s_{start}) - gd(s_{start})^2$ action executions over all safely explorable, Eulerian domains (Theorem 30 in Appendix 6.1). Thus, its complexity is at most $O(ed)$ action executions over all Eulerian domains and thus also over all undirected domains. Since $e \leq n^2$ and $d \leq n - 1$ for reasonable domains, its complexity is at most $O(n^3)$ action executions over all reasonable, Eulerian domains and thus also over all reasonable, undirected domains.

(Lower Bound) Figure 2.39 shows an example of a reasonable, undirected (and thus Eulerian) domain for which the number of action executions that Edge Counting needs in the worst case to reach a goal state is at least $O(n^3)$. In particular, it traverses the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller indices:

```
print (n+1)/2
for i := (n+3)/2 to n
  for j := i-2 downto (n+1)/2
    print j
  for j := 1 to (n-1)/2
    for k := j+1 to (n+1)/2
      print j
      print k
  for j := (n+3)/2 to i
    print j
```

In this case, Edge Counting executes $e \times gd(s_{start}) - gd(s_{start})^2 = n^3/8 + n^2/8 - 5/8 \times n + 3/8$ actions before it reaches the goal state (for odd $n \geq 1$). For example, for $n = 5$, it traverses the state sequence s_3, s_1, s_2, s_1, s_3, s_2, s_3, s_4, s_3, s_1, s_2, s_1, s_3, s_2, s_3, s_4, and s_5. Notice that $e = n^2/4 + n - 5/4$ (for odd $n \geq 1$) and $d = n/2 + 1/2$ (for odd $n \geq 3$) for the domain from Figure 2.39. Thus, the complexity is both at least $O(ed)$ and $O(n^3)$ action executions. ■

2.5.3.2 Efficient LRTA*-Type Real-Time Search Methods: The BETA Method

We showed that, in general, the complexity of every LRTA*-type real-time search method that fits our skeleton is at least cubic in the number of states over all reasonable domains. In Eulerian domains, there exist LRTA*-type real-time search methods that fit our skeleton and can do better than that. One example is the *Building a Eulerian Tour Algorithm* (BETA). BETA works only for Eulerian domains. Its exact origin is unclear. Deng and Papadimitriou [Deng and Papadimitriou, 1990] and Korach *et al.* [Korach *et al.*, 1990] stated it explicitly as a search method, but it has been used much earlier as part of proofs about properties of Eulerian graphs [Hierholzer, 1873].

To describe BETA, we need the following definition: We say that BETA is stuck when all edges leaving the current state have already been traversed at least once. Then, BETA can be described recursively as follows: BETA takes untraversed edges whenever possible. If it is stuck, it retraces the closed walk of previously untraversed edges that it has just completed and applies itself recursively from each state visited. Thus, BETA is similar to depth-first search, with the following difference: Since chronological backtracking is not always possible in directed graphs, BETA repeats its first edge traversals when it is stuck instead of backtracking its latest edge traversals.

BETA fits our skeleton, as we show in the following. Its q-values $q(s,a)$ are triples of integers. (Such a triple is then encoded as one integer, which we don't show here.) The first component, the *cycle number*, has index one and corresponds to the level of recursion of BETA. The second component has index two and counts the number of times action a has already been executed in state s, and the third component remembers when action a was executed first in state s (using a counter that is incremented after every action execution). The variable $memory$ is also a triple: its first two components remember the first two components of the previously executed action and its third component is the counter. All q-values are initialized with $(0,0,0)$ instead of 0.

BETA	(see Figure 2.27)
action-selection step (Line 3)	a := one-of $\arg\min_{a' \in X} q(s,a')[3]$ where $X = \arg\max_{a' \in Y} q(s,a')[1]$ and $Y = \arg\min_{a' \in A(s)} q(s,a')[2]$

value-update step (Line 4)	if q(s,a)[2] = 0 then
	  q(s,a)[3] := memory[3]+1
	  if memory[2] = 1 then
	  q(s,a)[1] := memory[1]
	  else
	  q(s,a)[1] := memory[1]+1
	q(s,a)[2] := q(s,a)[2] + 1
	memory[1] := q(s,a)[1]
	memory[2] := q(s,a)[2]
	memory[3] := memory[3] + 1

The action-selection step always chooses an action that has never been executed before. If no such action exists in the current state, it considers all actions that have been executed exactly once (such an action always exists) and chooses the action that belongs to the latest level of recursion. If there is more than one such action, ties are broken in favor of the action whose first execution preceded the executions of the other actions. When an action is executed for the first time, the value-update step remembers when it was executed and decides which cycle number it should get. If the action executed previously was executed for the first time too, then the new action inherits the cycle number of the previously executed action, otherwise a new level of recursion starts and the cycle number of the new action is one larger than the cycle number of the previously executed action. The value-update step also increments the number of times the current action has been executed. Finally, the value-update step remembers the first two components of the current action (so that it has them available after the action has been executed) and increments the counter.

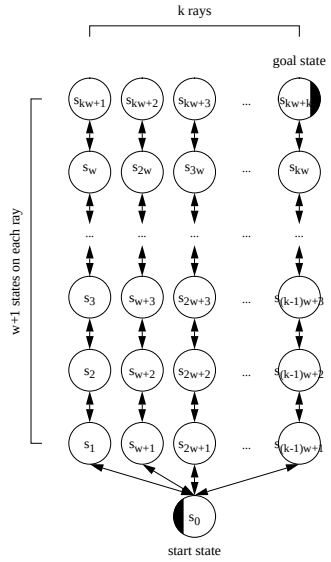


Figure 2.40: Undirected Star

BETA always reaches a goal state with a finite number of action executions in safely explorable, Eulerian domains and, moreover, executes every action at most twice [Deng and Papadimitriou, 1990]. Furthermore, the complexity of BETA is asymptotically tight at $2e$ action executions [Deng and Papadimitriou, 1990]. The complexity of BETA over a subset of domains can potentially be smaller than its complexity over all domains. However, its complexity remains asymptotically tight at $2e$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains [Koenig and Smirnov, 1996]. Figure 2.40 gives an example of a reasonable, undirected (and thus Eulerian) domain [Baeza-Yates *et al.*, 1993] for which the number of action executions that BETA needs in the worst case to reach a goal state is asymptotically $2e$. In particular, BETA can traverse the state sequence that is printed by the following program in pseudo code if it visits the states for the first time in the order s_0, s_1, s_2, ..., s_{kw+k}:

```
print 0
for i := 0 to k-1
  for j := w*i+1 to w*(i+1)
    print j
  for j := w*(i+1)-1 downto w*i+1
    print j
  print 0
for i := 0 to k-2
  for j := w*i+1 to w*(i+1)
    print j
  print w*k+1+i
  print w*(i+1)
  print w*k+1+i
  for j := w*(i+1) downto w*i+1
    print j
  print 0
for i := w*(k-1)+1 to w*k
  print i
print w*k+k
```

When BETA reaches the goal state, one action has not been executed, $w+1$ actions have been executed once, and $2kw+2k-w-2$ actions have been executed twice (for $k \geq 1$ and $w \geq 1$). Thus, in this case, BETA executes $4kw+4k-w-4$ actions before it reaches the goal state. Thus, there are $4kw+4k-w-4$ action executions and $2kw+2k$ actions, and the ratio of the two quantities approaches two for large $k = w$. Consequently, the complexity of BETA is asymptotically tight at $2e$ action executions. For example, for

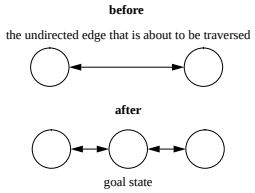


Figure 2.41: Domain Change

$k = 3$ and $w = 1$, it traverses the state sequence s_0, s_1, s_0, s_2, s_0, s_3, s_0, s_1, s_4, s_1, s_0, s_2, s_5, s_2, s_0, s_3, and s_6. Notice that BETA can be made more efficient. A variant of BETA could, for example, visit the states for the first time in the same order as BETA but always take a shortest path of already traversed edges to the next unvisited state. The performance of this variant of BETA dominates the performance of BETA. However, the example in Figure 2.40 shows that the complexity of this optimization of BETA is also asymptotically tight at $2e$ action executions (even over all reasonable, undirected domains and over all reasonable, Eulerian domains).

Theorem 18 *The complexity of every LRTA*-type real-time search method that fits our skeleton is at least $O(e)$ action executions over all safely explorable, undirected domains and over all safely explorable, Eulerian domains, and BETA has a tight complexity of $O(e)$ action executions over these domains. Furthermore, the complexity of every LRTA*-type real-time search method that fits our skeleton is at least $O(n^2)$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains, and BETA has a tight complexity of $O(n^2)$ action executions over these domains.*

Proof: (Upper Bound) BETA executes every action at most twice before it reaches the goal state in safely explorable, Eulerian domains [Deng and Papadimitriou, 1990]. Thus, its complexity is at most $O(e)$ action executions over all safely explorable, undirected domains and over all safely explorable, Eulerian domains. Since $e \leq n^2$ for reasonable domains, its complexity is at most $O(n^2)$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains.

(Lower Bound) For BETA and every other LRTA*-type real-time search method that fits our skeleton, we construct a reasonable (and thus safely explorable), undirected (and thus Eulerian) domain with $O(n^2)$ actions, almost half of which the LRTA*-type real-time search method has to traverse at least once in the worst case before it reaches a goal state: Consider the domain from Figure 2.39 or any other reasonable, undirected domain with $O(n^2)$ actions. Pick an arbitrary start state, but assume for now that it does not have a goal state. Run the LRTA*-type real-time search method and count how often it traverses each undirected edge (that is, sum up the edge traversals in both directions). Stop the LRTA*-type real-time search method immediately before it has traversed each undirected edge at least once. Replace this undirected edge with two undirected edges that are connected with an intermediate state and make the intermediate state the only goal state (Figure 2.41). Then, the next action execution results in a goal state and the LRTA*-type real-time search method executes at least $e/2-1$ actions before it reaches the goal state (for even $e \geq 4$). Thus, the complexity is both at least $O(e)$ and $O(n^2)$, since $e = O(n^2)$ according to our assumptions. ■

2.5.3.3 Min-LRTA*

The complexity of zero-initialized Min-LRTA* does not decrease over all undirected domains or over all Eulerian domains. It remains a small polynomial in n.

Theorem 19 *Zero-initialized Min-LRTA* has a tight complexity of $O(ed)$ action executions over all undirected domains and over all Eulerian domains. Furthermore, it has a tight complexity of $O(n^3)$ action executions over all reasonable, undirected domains and over all reasonable, Eulerian domains.*

Proof: (Upper Bound) The complexity of zero-initialized Min-LRTA* is at most $O(ed)$ action executions or, in reasonable domains, $O(n^3)$ action executions (Theorem 15).

(Lower Bound) Figure 2.39 shows an example of a reasonable, undirected (and thus Eulerian) domain for which the number of action executions that Min-LRTA* needs in the worst case to reach a goal state is at least $O(ed)$ or, alternatively, $O(n^3)$. In particular, it traverses the state sequence that is printed by the following program in pseudo code:

```
for i := n-1 downto (3n+1)/4
  print (n+1)/2
  for j := 1 to (n-1)/2
    for k := j+1 to (n+1)/2
      print j
      print k
  for j := (n+3)/2 to i-1
    print j
  for j := i downto (n+3)/2
    print j
print (n+1)/2
for j := 1 to (n-1)/2
  for k := j+1 to (n+1)/2
    print j
    print k
for j := (n+3)/2 to n
  print j
```

In this case, Min-LRTA* executes $n^3/16 + 3n^2/8 - 3n/16 - 1/4$ actions before it reaches the goal state (for $n \geq 1$ with $n \bmod 4 = 1$). For example, for $n = 5$, it traverses the state sequence s_3, s_1, s_2, s_1, s_3, s_2, s_3, s_4, s_3, s_1, s_2, s_1, s_3, s_2, s_3, s_4, and s_5. Notice that $e = n^2/4 + n - 5/4$ (for odd $n \geq 1$) and $d = n/2 + 1/2$ (for odd $n \geq 3$) for the domain in Figure 2.39. Thus, the complexity is both at least $O(ed)$ and $O(n^3)$ action executions. ■

2.5.3.4 Experimental Average-Case Results

So far, we have been concerned only with the complexity of LRTA*-type real-time search methods that fit our skeleton. However, their experimental average-case performance is equally important for practical purposes. In this section, we present a simple case study that shows that their experimental average-case performance is similar to their complexities.

We study the following zero-initialized LRTA*-type real-time search methods that fit our skeleton: Edge Counting, Min-LRTA*, random walks, and BETA. We study these four LRTA*-type real-time search methods in the two blocks-world domains in Figures 2.42 (Domain 1) and 2.43 (Domain 2) [Koenig and Simmons, 1996a]. Domain 1 is a Eulerian domain, and Domain 2 is a non-Eulerian domain that is a variant of reset state spaces. Otherwise both domains are very similar. In both cases, there are x indistinguishable blocks, all of which are initially on the table. The task is to stack all of them on top of one another on a platform (Figure 2.44). Domain 1 has four operators: "pick-up block from table," "put block on stack," "pick-up block from stack," and "put block on table." A "pick-up block from table" is always followed by a "put block on stack," and a block picked up from the stack is always subsequently placed on the table. Domain 1 is similar to traditional blocks-world domains, that are undirected, except that traditional blocks-world domains usually merge a pair of "pick-up" and "put-down" operators into one atomic "move" operator. Domain 2 has the same two pick-up operators and the same "put block on stack" operator, but the "put block on table" operator (which always follows a "pick-up block from stack" operator) knocks down the whole stack onto the table. Both domains are reasonable and very similar: both have $3x + 1$ states, $4x$ actions, and maximal goal distance $2x + 1$ (for $x \geq 1$). Furthermore, corresponding states have the same small number of actions available, either one or two.

Figures 2.45 and 2.46 show how many actions the LRTA*-type real-time search methods execute in these

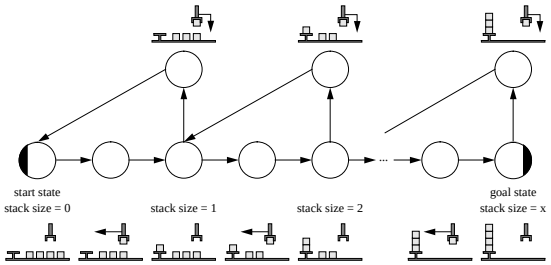


Figure 2.42: Domain 1

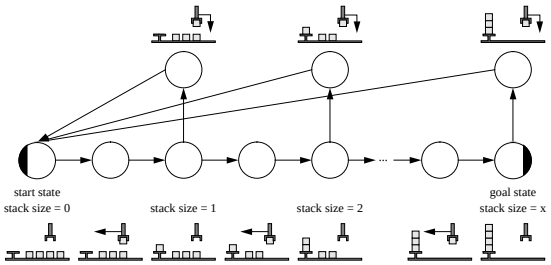


Figure 2.43: Domain 2

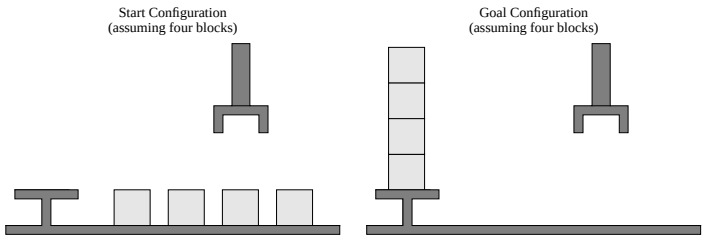


Figure 2.44: Simple Blocks-World Task

domains. Both figures are scaled in the same proportion. Their horizontal axes show the number of blocks x and their vertical axes the number of action executions needed to reach the goal state, averaged over 5,000 runs with randomly broken ties. The number of actions that random walks execute was calculated analytically. For example, it follows from the proof of Theorem 9 that random walks execute on average $3 \times 2^x - 4$ actions in Domain 2 before they reach the goal state (for $x \geq 1$). The legend of the figures always lists the graphs from top (in this case, maximal number of action executions) to bottom to make it easier to identify the graphs.

The experiments show that every studied LRTA*-type real-time search method does better in Domain 1 than in Domain 2. Random walks perform worst in both domains. This is to be expected since they do not remember any information. However, Edge Counting performs almost as poorly as random walks in Domain 2, and both search methods quickly become intractable. With 50 blocks, for example, random walks need $3 \times 2^{50} - 4$ action executions (that is, about 3.4×10^{15} action executions) on average to reach the goal state and perform about 500 billion times worse than Min-LRTA*, which needs only 6838.3 action executions on average. On the other hand, all search methods do quite well in Domain 1. Even the ones that perform poorly in Domain 2 perform almost as well as Min-LRTA*, the LRTA*-type real-time search method that performs well in both

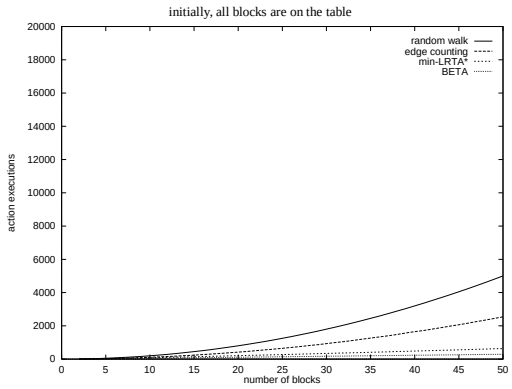


Figure 2.45: Experimental Average-Case Performance in Domain 1

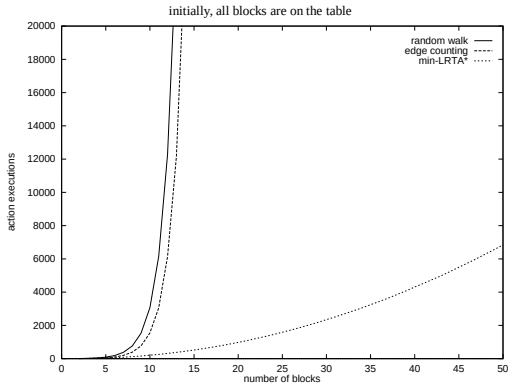


Figure 2.46: Experimental Average-Case Performance in Domain 2

Figure 2.47: One-Dimensional Grid-World (2)

domains. With 50 blocks, for example, Min-LRTA* performs 2.2 times worse than BETA (which needs 292.0 action executions on average), Edge Counting performs 8.7 times worse, and even random walks perform only 17.1 times worse. Thus, the interval spanned by the experimental average-case performance of the studied LRTA*-type real-time search methods is much smaller in Domain 1, a Eulerian domain, than in Domain 2, a non-Eulerian domain that is a variant of reset state spaces. All studied LRTA*-type real-time search methods are tractable in Domain 1 but some of them are intractable in Domain 2. This is similar to the theoretical complexity results.

2.5.3.5 Interpretation of the Results

We have studied selected LRTA*-type real-time search methods that fit our skeleton. In particular, we studied uninformed LRTA*-type real-time search methods with minimal look-ahead that solve one-shot planning tasks. We did this because these LRTA*-type real-time search methods make similar assumptions. The drawback is that one cannot compare LRTA*-type real-time search methods with each other solely on the

basis of the complexity results since some LRTA*-type real-time search methods are better than others in using heuristic knowledge to guide planning, allowing for larger look-aheads, and improving their performance over time as they solve similar planning tasks. Min-LRTA*, for example, has all three features, but other LRTA*-type real-time search methods are often as efficient as Min-LRTA* given our assumptions. For instance, we have already seen that Min-LRTA* and Edge Counting have the same tight complexity over all undirected domains and over all Eulerian domains, and that Min-LRTA* had only a moderate advantage in experimental average-case performance over Edge Counting in Domain 1, a Eulerian domain.

One can also construct domains in which Edge Counting performs better than Min-LRTA*. An example is given in Figure 2.47. (Another example is the domain in Figure 2.39 with start state $n-2$.) Min-LRTA* can traverse the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller indices except for the first action execution in which the tie is broken in the opposite way:

```
print n-2
for i := n-1 downto 1
  print i
for i := n-2 downto 2
  for j := 2 to i
    print j
  for j := i-1 downto 1
    print j
for i := 2 to n
  print i
```

In this case, Min-LRTA* executes n^2-3n+4 actions before it reaches the goal state (for $n \geq 3$). For example, for $n = 5$, it traverses the state sequence s_3, s_4, s_3, s_2, s_1, s_2, s_3, s_2, s_1, s_2, s_1, s_2, s_3, s_4, and s_5. On the other hand, we have shown that Edge Counting is guaranteed to need at most $e \times gd(s_{start}) - gd(s_{start})^2 = 4n-8$ action executions to reach a goal state (for $n \geq 3$). (This bound turns out to be tight for this particular domain if ties are broken in favor of successor states with smaller indices.) For example, for $n = 5$, it traverses the state sequence s_3, s_2, s_1, s_2, s_3, s_4, s_3, s_2, s_1, s_2, s_3, s_4, and s_5. Since $n^2 - 3n + 4 > 4n - 8$ for $n > 4$, the complexity of Edge Counting for this particular search task is guaranteed to be smaller than that of Min-LRTA* for $n \geq 5$. Experiments show that a similar relationship holds on average.

The same effect can also be obtained in the two blocks-world domains in Figures 2.42 (Domain 1) and 2.43 (Domain 2). If we change the start state in both domains so that all but four blocks are already stacked initially, then both domains become easier to search. However, Figure 2.49 shows that the performance relationships of the LRTA*-type real-time search methods studied in Section 2.5.3.4 remain similar in Domain 2 (With 50 blocks, for example, Min-LRTA* now needs 6414.1 action executions on average to reach the goal state compared to 6838.3 action executions before.) Figure 2.48, on the other hand, shows that the performance relationships in Domain 1 change. (Figures 2.48 and 2.49 are scaled differently than Figures 2.45 and 2.46.) With 50 blocks, for example, Min-LRTA* now performs 1.3 times worse than BETA (that needs 182.3 action executions on average) and random walks perform 4.2 times worse, but Edge Counting performs 3.8 times *better* than BETA.

2.5.3.6 Summary of Results on Undirected Domains and Other Eulerian Domains

Our study was a first step in the direction of understanding how domain properties influence the performance of LRTA*-type real-time search methods. In this section, we studied only one particular domain property, namely, undirected domains and other Eulerian domains, and considered their effect only on the performance of selected LRTA*-type real-time search methods, but not on a whole class of LRTA*-type real-time search methods. In the next section, we study a second domain property.

We compared Min-LRTA* to both efficient LRTA*-type real-time search methods, such as BETA, and – equally importantly – to LRTA*-type real-time search methods that can be inefficient, such as Edge Counting. The complexity results illustrate that one can learn not only from comparing search methods to the best known

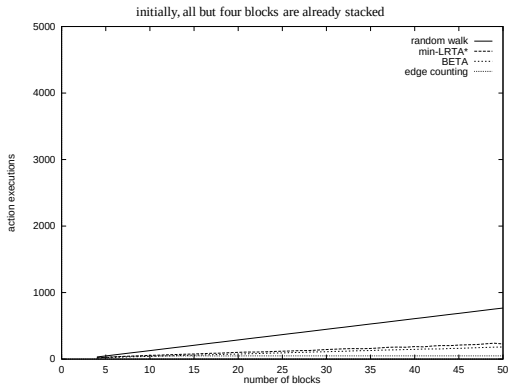


Figure 2.48: Experimental Average-Case Performance in Domain 1

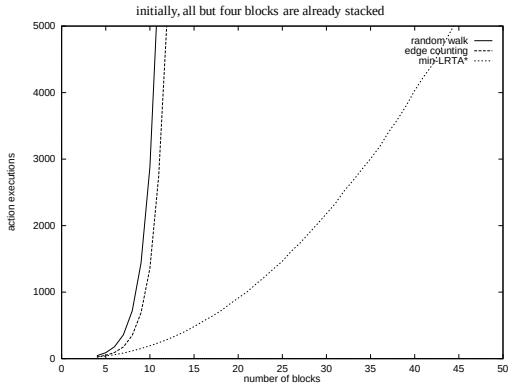


Figure 2.49: Experimental Average-Case Performance in Domain 2

ones, as is usually done, but also from comparing them to inefficient ones.

We illustrated, both theoretically and experimentally, that the performance of the studied LRTA*-type real-time search methods can differ significantly over all reasonable domains and over all reasonable, Eulerian domains: some non-Eulerian domains are hard to search for the studied LRTA*-type real-time search methods. LRTA*-type real-time search methods differ in this respect from traditional search methods. We derived the following complexity results about reasonable domains. Over all reasonable domains, no LRTA*-type real-time search method can beat the complexity of Min-LRTA*, which is a small polynomial in n. In contrast, the complexity of Edge Counting, the deterministic LRTA*-type real-time search method that we derived from random walks, is at least exponential in n. The picture changes over all reasonable, Eulerian domains. The complexity of Edge Counting decreases to a small polynomial in the number of states and equals now the complexity of Min-LRTA*, which remains unchanged. In addition, BETA (a dedicated LRTA*-type real-time search method for Eulerian domains) has a smaller complexity. This illustrates that LRTA*-type real-time search methods that make use of special domain properties can have a smaller complexity than Min-LRTA*. All complexities remain the same over all reasonable, undirected domains, a subset of reasonable, Eulerian domains.

To summarize, while Min-LRTA* does rather well over all reasonable domains when being compared to the other studied LRTA*-type real-time search methods, its advantage decreases over all reasonable, undirected domains and over all reasonable, Eulerian domains, over which the complexities of the studied LRTA*-type

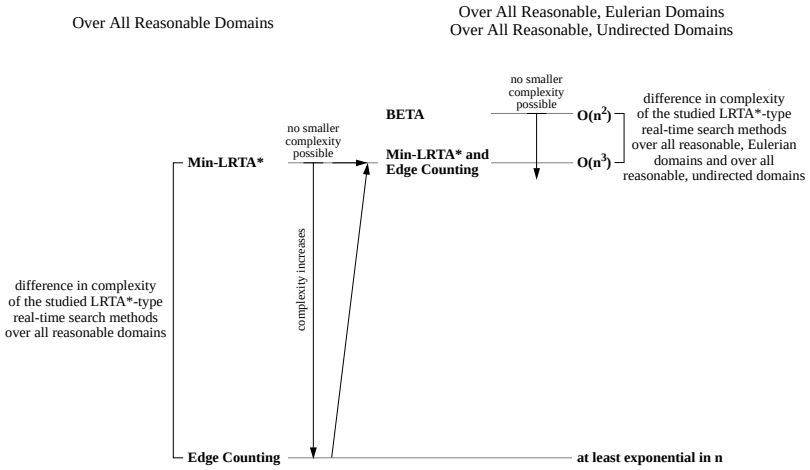


Figure 2.50: Complexities in Undirected Domains and other Eulerian Domains

real-time search methods span a much smaller interval than over all reasonable domains (Figure 2.50). In particular, the complexities of all studied LRTA*-type real-time search methods are small polynomials in the number of states over all reasonable, undirected domains and over all reasonable, Eulerian domains. This includes the complexities of LRTA*-type real-time search methods that can be intractable in general. Thus, undirected domains and other Eulerian domains are easier to search with the studied LRTA*-type real-time search methods than some non-Eulerian domains such as, for example, reset state spaces, quicksand state spaces, and "complex state spaces." Consequently, Min-LRTA* (an LRTA*-type real-time search method that is always efficient) and Edge Counting (an LRTA*-type real-time search method that can be intractable) have a small complexity on both sliding-tile puzzles and grid-worlds, but reset state spaces, quicksand state spaces, and "complex state spaces" are able to differentiate between them.

2.5.3.7 Extensions: Larger Look-Aheads

Although we have limited our study to LRTA*-type real-time search methods with minimal look-ahead, we would like to point out briefly how some of the complexity results transfer to LRTA*-type real-time search methods with larger look-aheads. Researchers have proposed various strategies that improve the performance of LRTA*-type real-time search methods, such as strategies for generating larger local search spaces or the combination of LRTA*-type real-time search methods and other search mechanisms. Examples include [Shekhar and Dutta, 1989, Sutton, 1990, Hamidzadeh and Shekhar, 1991, Hamidzadeh, 1992, Ishida, 1992, Shekhar and Hamidzadeh, 1992, Hamidzadeh and Shekhar, 1993, Ishida, 1995, Knight, 1993, Moore and Atkeson, 1993, Shakhar and Hamidzadeh, 1993, Dearden and Boutilier, 1994, Pemberton and Korf, 1994, Matsubara and Ishida, 1994, Ishida and Shimbo, 1996]. Methods that determine how much to plan between plan executions have been described in [Boddy and Dean, 1989, Russell and Wefald, 1991, Zilberstein, 1993, Goodwin, 1994]. In the following, we discuss both Node Counting, a variant of Edge Counting with larger look-ahead, and LRTA* with look-ahead one, the inspiration for Min-LRTA*.

Node Counting is a memoryless LRTA*-type real-time search method that differs from Edge Counting in that it looks at the successor states of the current state when choosing actions. The q-values approximate the negative number of times the actions have been executed.

Node Counting	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of $\arg\max_{a' \in A(s)} \sum_{a'' \in A(succ(s,a'))} q(succ(s,a'),a'')$
value-update step (Line 4)	$q(s,a) := -1 + q(s,a)$

The action-selection step always executes the action that leads to the successor state that has been visited the fewest number of times. In an actual implementation, one would maintain only one u-value $u(s)$ for each state s with $u(s) = \sum_{a \in A(s)} q(s, a)$. In this case, initially, $u(s) = 0$ for all $s \in S$. The u-values approximate the negative number of times the states have been visited. Since the initial u-values are consistent, we can use the simplified value-update step.

Node Counting	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of arg $\max_{a' \in A(s)} u(succ(s, a'))$
value-update step (Line 4)	$u(s) := -1 + u(s)$

We compare Node Counting to LRTA* with look-ahead one (Figure 2.6). LRTA* is similar to Node Counting in that it is a memoryless LRTA*-type real-time search method that looks at the successor states of the current state when choosing actions. It has the same action-selection step as Node Counting but a slightly different value-update step: it uses the u-value of the successor state instead of the u-value of the current state in the value-update step. Initially, $u(s) = 0$ for all $s \in S$. The u-values approximate the negative goal-distances of the states.

LRTA* with Look-ahead One	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of arg $\max_{a' \in A(s)} u(succ(s, a'))$
value-update step (Line 4)	$u(s) := -1 + u(succ(s, a))$

We have shown in Section 2.3.3.3 that zero-initialized LRTA* with look-ahead one has a tight complexity of $O(nd)$ action executions, and that it has a tight complexity of $O(n^2)$ action executions over all safely explorable domains (Theorem 5). Figure 2.14 shows that it also has a tight complexity of $O(nd)$ action executions over all undirected domains and over all Eulerian domains, and that it has a tight complexity of $O(n^2)$ action executions over all safely explorable, undirected domains and over all safely explorable, Eulerian domains.

The experimental average-case performance of zero-initialized Node Counting and LRTA* with look-ahead one are comparable in many traditional search domains from artificial intelligence. For example, variants of Node Counting have been used independently in [Pirzadeh and Snyder, 1990] and [Thrun, 1992b] to explore unknown grid-worlds (either on their own or to accelerate reinforcement-learning methods), in both cases with great success. Two of our experiments confirm these results. In both experiments, the performance difference between Node Counting and LRTA* with look-ahead one is statistically insignificant for any reasonable level of significance (for both sign tests and t tests).

Sliding-Tile Puzzles: We compare zero-initialized Node Counting and LRTA* with look-ahead one on the eight puzzle with the American goal state (Figure 2.7). We count the number of action executions until the LRTA*-type real-time search methods reach the goal state, averaged over 25,000 runs with randomly broken ties. The same 25,000 randomly chosen start states are used in both cases. Node Counting needs, on average, 85,579 action executions to reach the goal state, compared to 85,746 action executions needed by LRTA*. Out of the 25,000 runs, Node Counting outperforms LRTA* 12,512 times and is beaten 12,488 times.

Grid-Worlds: We perform the same experiment on an obstacle-free square grid-world of size 50×50 whose start and goal states are in opposite corners. Node Counting needs, on average, 2,874 action executions to reach the goal state, compared to 2,830 action executions needed by LRTA*. Out of the 25,000 runs, Node Counting outperforms LRTA* 12,345 times, is beaten 12,621 times, and ties 34 times.

According to a recent, still unpublished result by Szymanski, the complexity of Node Counting is at least exponential in n even over all reasonable, undirected domains [Szymanski, private communication]. This is interesting because it implies that there must be some other (yet-to-be-discovered) property of sliding-tile puzzles or grid-worlds that makes them easy to solve with node counting. For our purposes, it is sufficient to show that the complexity of Node Counting is exponential in n for *some* reasonable domains (we do not show this for undirected domains): Consider any Domain X and a Domain Y that is derived from Domain

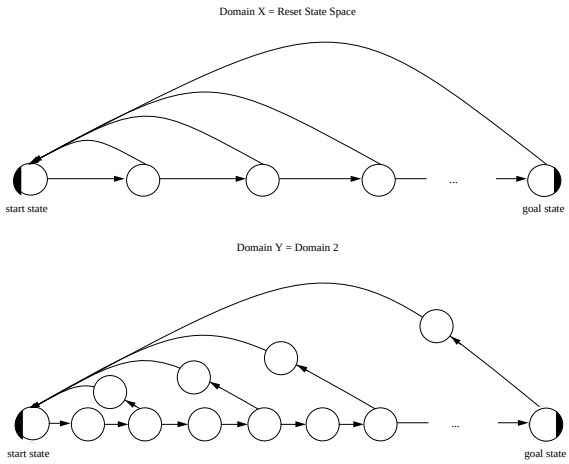


Figure 2.51: Domain with Intermediate States

X by replacing each of its directed edges with two directed edges that are connected with an intermediate state. An example is shown in Figure 2.51. Then, Node Counting in Domain Y, Edge Counting in Domain Y, and Edge Counting in Domain X behave identically (if ties are broken identically). This means that Node Counting in Domain Y and Edge Counting in Domain Y execute the same number of actions, which is twice the number of actions that Edge Counting executes in Domain X (since they have to execute two actions for every action that Edge Counting executes in Domain X). As examples, consider the blocks-world domains in Figures 2.42 (Domain 1) and 2.43 (Domain 2). Domain 1 is derived from a one-dimensional grid-world (Figure 2.15), and thus the experimental average-case performance of Node Counting and Edge Counting (as shown in Figure 2.45) are identical in Domain 1. Similarly, Figure 2.51 shows that Domain 2 is derived from a reset state space (Figure 2.28), and thus the experimental average-case performance of Node Counting and Edge Counting (as shown in Figure 2.46) are identical in Domain 2, and so are their complexities. Thus, Domain 2 is an example of a domain for which the number of action executions that Node Counting needs in the worst case to reach a goal state is at least exponential in n if ties are broken in favor of actions that lead "upward:" The appearance of the intermediate "pick-up" operators makes it so that a look-ahead of one is insufficient to avoid reset actions. Variants of quicksand state spaces and "complex state spaces" with this property can be constructed in the same way. This shows that the complexity of Node Counting is at least exponential in n (even over all reasonable domains).

To summarize, LRTA* with look-ahead one (an LRTA*-type real-time search method that is always efficient) and Node Counting (an LRTA*-type real-time search method that can be intractable) have almost the same experimental average-case performance on both sliding-tile puzzles and grid-worlds, but variants of reset state spaces, quicksand state spaces, and "complex state spaces" are able to differentiate between them. Similar reset state spaces, quicksand state spaces, and "complex state spaces" can also be constructed for LRTA*-type real-time search methods with even larger look-aheads. This is similar to the complexity results about LRTA*-type real-time search methods with minimal look-ahead.

2.5.4 Domains with a Small Value of ed

So far, we were concerned with only one domain property, namely, undirected domains and other Eulerian domains. The complexity results, however, also uncovered another domain property that is relevant to the performance of LRTA*-type real-time search methods. In particular, the expressions ed and nd showed up repeatedly in the complexity results. In this section, we show experimental results that indicate that both ed and nd capture the performance of several LRTA*-type real-time search methods well in several domains:

The performance of the studied LRTA*-type real-time search methods can be predicted by measuring their performance in domains of the same topology but smaller values of ed and nd.

The measures of task size ed and nd were introduced in [Koenig and Simmons, 1992] and [Koenig and Simmons, 1993a] and, if scaled with n^2, later called "oblongness" in [Smirnov, 1997]. The complexity results that involve them are the following:

- Zero-initialized Min-Max LRTA* with look-ahead one and zero-initialized LRTA* with look-ahead one have a tight complexity of $O(nd)$ action executions over all deterministic domains (Theorem 5).
- Zero-initialized Min-LRTA* has a tight complexity of $O(ed)$ action executions over all deterministic domains (Theorem 15), and no search method that fits our skeleton can do better in the worst case (Theorem 11).
- Random walks have a tight average-case performance of $O(ed)$ action executions over all undirected domains and over all Eulerian domains (Theorem 16). Zero-initialized Edge Counting has a tight complexity of $O(ed)$ action executions over these domains (Theorem 17).

Similar complexity results also hold for other LRTA*-type real-time search methods: An example is the LRTA*-type real-time search method called "Vertex Ant Walk, " whose complexity is tight at $O(nd)$ action executions over deterministic domains [Wagner *et al.*, 1997]. Another example is zero-initialized Bi-Directional Q-Learning ("Version 2") [Koenig and Simmons, 1992], that is a variant of Min-LRTA* that finds optimal plans from every state. It terminates and does not require that the agent be reset to the start state when it reaches a goal state. Its complexity is tight at $O(ed')$ over deterministic domains, where d' is the maximal distance between any two states in the domain (its diameter) and $d' = d$ for the following test domains. Thus, Bi-Directional Q-Learning demonstrates that finding optimal plans with Min-LRTA* is not necessarily more complex than only reaching a goal state [Koenig and Simmons, 1992].

This suggests using ed and nd as measures of task size. Many of the above complexities are also tight at $O(n^3)$ or $O(n^2)$ action executions, respectively, over all reasonable domains. Thus, we could use n^3 and n^2 as measures of task size as well. We do not do that because ed and nd often grow more slowly than n^3 and n^2, respectively. This is the case for the following two reasons:

The number of actions e often grows more slowly than n^2, the square of the number of states. In fact, it often grows only linearly in n because often the number of actions that can be executed in every state is bounded from above by a (usually small) constant. For example, at most four actions can be executed in every state of both sliding-tile puzzles and grid-worlds.

The maximal goal distance d often grows more slowly than $n - 1$, one less than the number of states. For example, we have already discussed that the maximal goal distance of the eight puzzle is only 30 for the American goal state and 31 for the European goal state, but the eight puzzle has 181,440 states. Similarly, an obstacle-free square grid-world of size $x \times x$ has $n = x^2$ states and maximal goal distance $d = 2x - 2$. Thus, $d = 2\sqrt{n} - 2$.

In the following, we investigate the measures of task size experimentally. Both ed and nd lead to identical results in many domains since e is often linear in n, as argued above. This holds for all of the domains tested. We therefore investigate only ed. We study the following zero-initialized LRTA*-type real-time search methods: Bi-Directional Q-Learning, Edge Counting, LRTA* with look-ahead one, Min-LRTA*, and random walks. We study these five LRTA*-type real-time search methods in reset state spaces with $n = 2, 3, \ldots, 50$ states (Figure 2.28), one-dimensional grid-worlds with $n = 2, 3, \ldots, 50$ states whose start and goal states are at opposite ends (Figure 2.15), and obstacle-free square grid-worlds with $n = 4, 9, \ldots 196$ states whose start and goal states are in opposite corners (Figure 2.38).

Figures 2.52, 2.53, and 2.54 show how many actions the LRTA*-type real-time search methods execute in these domains. Since we know that the performance of random walks and Edge Counting can be exponential in the number of states in non-Eulerian domains, we do not show the performance graphs for these two

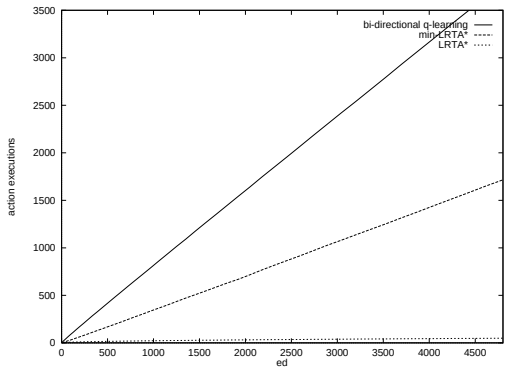


Figure 2.52: Experimental Average-Case Performance in Reset State Spaces

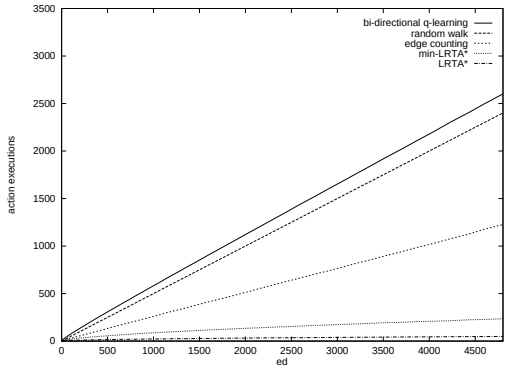


Figure 2.53: Experimental Average-Case Performance in One-Dimensional Grid-Worlds

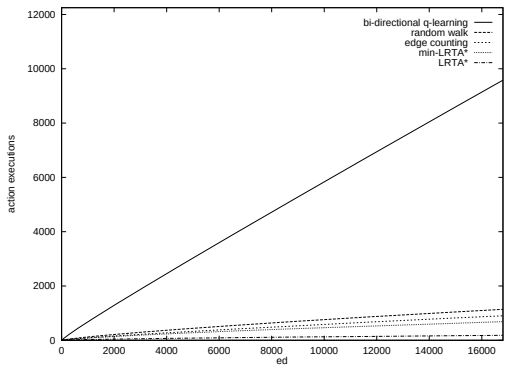


Figure 2.54: Experimental Average-Case Performance in Square Grid-Worlds

LRTA*-type real-time search methods in the reset state space. All figures are scaled in the same proportion. Their horizontal axes show the measure of task size ed and their vertical axes the number of action executions needed to complete the tasks, averaged over 5,000 runs with randomly broken ties. The number of actions that random walks execute in one-dimensional grid-worlds was calculated analytically as the behavior of symmetrical one-dimensional random walks with one reflecting and one absorbing barrier [Feller, 1966].

Notice that many of the graphs appear to be almost linear, especially in large domains. This is not a trivial property. For example, Figure 2.45 shows that the experimental average-case performance of LRTA*-type real-time search methods is often not linear in n (the figure uses the number of blocks, but the number of states is one larger than three times the number of blocks). This makes ed a better measure of task size for LRTA*-type real-time search methods than n, but the measure is not perfect: First, the slopes of the graphs are not the same for a given LRTA*-type real-time search method across domains. We already observed this when Min-LRTA* outperformed Edge Counting in Figure 2.45 and Edge Counting outperformed Min-LRTA* in Figure 2.48. Second, some of the graphs are sub-linear. Consider, for example, LRTA*. It needs exactly $n - 1$ action executions to reach the goal state in reset state spaces and one-dimensional grid-worlds. In both domains, $d = n - 1$ and $e = 2n - 2$. Thus, the number of action executions is $\sqrt{ed}/\sqrt{2}$, which is sub-linear in ed. Despite these two problems, the performance of the studied LRTA*-type real-time search methods can be predicted reasonably well in the studied domains by measuring their performance in smaller domains of the same topology. Future work will gather more experience with this measure of task size in more complex domains and with additional LRTA*-type real-time search methods (including those with larger look-aheads).

2.5.5 Selection of Test-Beds for LRTA*-Type Real-Time Search Methods

We have studied how domain properties influence the performance of LRTA*-type real-time search methods. The complexity results can help to distinguish easy LRTA*-type real-time search tasks from hard ones, which can help experimental researchers to decide when to use LRTA*-type real-time search methods. We showed, for instance, that sliding-tile puzzles and grid-worlds are well suited for LRTA*-type real-time search, for two reasons: First, they are undirected. Second, the product of their number of actions and their maximal goal distance is small because both factors are small (relative to the number of states). In the following, we show another application of the complexity results: the selection of test-beds for LRTA*-type real-time search methods.

Often, engineers have to choose methods that solve their problems but they do not have the time to implement (or analyze) all of the candidate methods. Instead, they rely on results reported in the literature, such as results that show how well a method operates in various *test-beds* (prototypical test domains). The engineers then interpret these results in the context of their problem. It is therefore important that the performance of LRTA*-type real-time search methods in the test-beds be representative of their performance in the domains of interest: test-beds should either directly reflect the domain properties of the domains of interest or, at least, be representative of a wide range of domains. Complexity results that suggest which domain properties make domains easy to search with LRTA*-type real-time search methods can help experimental researchers to choose appropriate test-beds for experimenting with LRTA*-type real-time search methods, reporting their results, and interpreting the results reported by others.

We distinguish discriminating and nondiscriminating test-beds. We consider a domain to be a *nondiscriminating test-bed* for LRTA*-type real-time search methods (*Type 1*) if no LRTA*-type real-time search method has a significant performance advantage over other (reasonable) LRTA*-type real-time search methods, otherwise the domain is a *discriminating test-bed* (*Type 2*). In Figure 2.55, for example, domains of Type 1a are nondiscriminating test-beds, because they are hard to search with LRTA*-type real-time search methods: even the most efficient LRTA*-type real-time search methods perform poorly in them. Likewise, domains of Type 1b are nondiscriminating, because they are easy to search: even inefficient "reasonable" LRTA*-type real-time search methods (Section 2.5.2.1) perform very well in them. Domains of Type 2, on the other hand, are discriminating test-beds, because they are better able to discriminate between efficient and inefficient LRTA*-type real-time search methods.

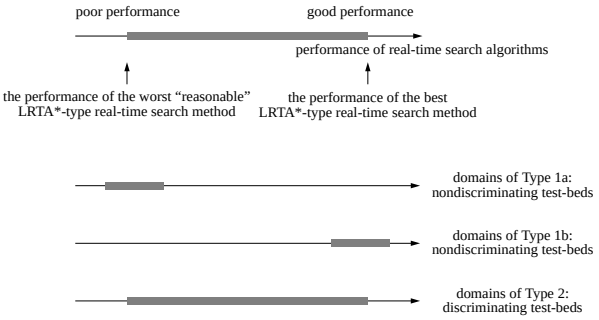


Figure 2.55: Discriminating and Nondiscriminating Test-Beds (1)

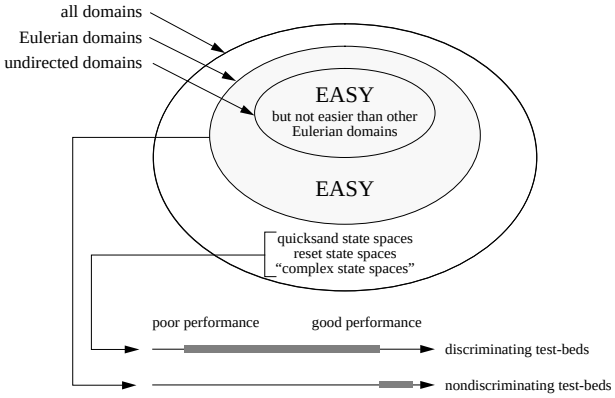


Figure 2.56: Discriminating and Nondiscriminating Test-Beds (2)

Comparing LRTA*-type real-time search methods using test-beds of Type 1 is fine as long as the test-beds are representative of the domains that are currently of interest to engineers. However, there is a problem with reporting performance results of LRTA*-type real-time search methods only for test-beds of Type 1. Some day, an engineer might become interested in domains with properties that have not been anticipated. Consequently, no performance results have been reported for this domain or similar domains, and the engineer is likely going to use the published performance results for other domains to decide which LRTA*-type real-time search method to use.

Which LRTA*-type real-time search method to use is not crucial if the domains of the engineer are of Type 1. Assume, however, that the domains are of Type 2 but the engineer has only experimental evaluations of LRTA*-type real-time search methods in domains of Type 1 available. In this case, the reported results appear to suggest that all LRTA*-type real-time search methods perform equally well and that it does not matter much which of them is used to solve the task. In reality, however, there is a huge difference in performance among the LRTA*-type real-time search methods in the domains of interest and the engineer should carefully decide which one to use. Consequently, when choosing appropriate LRTA*-type real-time search methods, one can easily be misled if one has available only evaluations of LRTA*-type real-time search methods in domains of Type 1. One might even choose an LRTA*-type real-time search method that is intractable in the domains of Type 2.

We identified domain properties that make domains easy to search with LRTA*-type real-time search methods. For example, the complexity results showed that undirected domains and other Eulerian domains are easy to search with a variety of LRTA*-type real-time search methods, even those LRTA*-type real-time search methods that can be intractable in general. Consequently, undirected and other Eulerian domains are of Type 1b and are not ideal test-beds for comparing LRTA*-type real-time search methods across a wide range

of domains (Figure 2.56). This includes most traditional search domains from artificial intelligence, such as sliding-tile puzzles and grid-worlds.

We therefore suggest reporting experimental results not only for undirected domains and other Eulerian domains, but also for non-Eulerian domains. We identified three classes of non-Eulerian test-beds that are of Type 2, namely, reset state spaces, quicksand state spaces, and "complex state spaces." They are able to separate some LRTA*-type real-time search methods with minimal look-ahead that can be intractable from LRTA*-type real-time search methods with minimal look-ahead that are always tractable and, thus, do not suffer as much from the problems of the traditional test-beds. Furthermore, minor variants of these domains can also differentiate among LRTA*-type real-time search methods with larger look-aheads. Thus, variants of these domains should be included in test suites that are used to evaluate LRTA*-type real-time search methods. We currently do not know of any real-world domains that resemble reset state spaces, quicksand state spaces, or "complex state spaces." This, however, should not prevent one from including them in test suites, since domains of future applications might resemble them. Furthermore, we do not expect any single domain to be sufficient for comparing LRTA*-type real-time search methods. In quicksand state spaces, for example, all actions have only local effects and some inefficient LRTA*-type real-time search methods might be able to perform efficiently in them.

To summarize, the complexity results can help experimental researchers to choose appropriate test-beds for experimenting with LRTA*-type real-time search methods, reporting their results, and interpreting the results reported by others.

2.5.6 Selection of Representations for Reinforcement-Learning Methods

In this section, we study another application of the complexity results: the selection of representations for reinforcement-learning tasks. *Reinforcement learning* [Barto *et al.*, 1989, Kaelbling *et al.*, 1986] is learning from positive and negative rewards. In this section, we relate reinforcement learning to LRTA*-type real-time search and use the complexity results to analyze the performance of reinforcement-learning methods for various representations of reinforcement-learning tasks. We show that the choice of representation can have a large impact on their performance and that reinforcement-learning tasks that were considered intractable can be made tractable with only a simple change in their representation. Thus, the complexity results can help experimental researchers to choose representations for reinforcement-learning methods that enable them to solve reinforcement-learning tasks efficiently.

Reinforcement learning is often done in the context of completely observable Markov decision process models (Section 3.2): An agent receives finite *immediate reward* $r(s, a)$ when it executes action a in state s. If the agent receives immediate reward r_t when it executes the $(t+1)$st action, then its *total reward* is $\sum_{t=0}^{\infty}[\gamma^t r_t]$, where $\gamma \in (0, 1]$ is the *discount factor*. The discount factor specifies the relative value of an immediate reward received after t action executions compared to the same reward received one action execution earlier. If $\gamma = 1$, the total reward is called *undiscounted* otherwise it is called *discounted*.

Reinforcement-learning methods express plans as mappings from states to actions (also known as "stationary, deterministic policy," short: policy). Although the notion "policy" originated in the field of stochastic dynamic programming, similar schemes have been proposed in the context of artificial intelligence, including universal plans [Schoppers, 1987]. Reinforcement-learning methods determine policies that maximize the (average) total reward of the agent (no matter which state it starts in) provided that the agent always executes the action that the policy assigns to its current state.

A *goal-directed reinforcement-learning task* can be stated as: the agent has to learn an optimal policy (shortest path) for reaching a goal state in an unknown domain. Goal-directed reinforcement-learning tasks have been studied in [Benson and Prieditis, 1992, Pemberton and Korf, 1992, Peng and Williams, 1992, Moore and Atkeson, 1993, Barto *et al.*, 1995], among others. One necessary step towards a solution of goal-directed reinforcement-learning tasks is to locate a goal state. We therefore study the complexity of reinforcement-learning methods until the agent reaches a goal state for the first time. If reinforcement-

learning methods terminate when the agent reaches a goal state, then they solve the following *goal-directed exploration task*: the agent has to reach a goal state in an unknown domain, but it does not need to learn an optimal policy. Studying goal-directed exploration tasks provides insight into the corresponding goal-directed reinforcement-learning tasks. Whitehead [Whitehead, 1991a], for example, proved that solving goal-directed exploration tasks with reinforcement-learning methods can be intractable, which illustrates that solving goal-directed reinforcement-learning tasks can be intractable as well. In particular, he showed that the behavior of uninformed reinforcement-learning methods can degenerate to a random walk until they reach a goal state for the first time, even in deterministic domains. In this case, the number of action executions that they need on average to reach a goal state can be exponential in the number of states. This complexity result seems to limit the usefulness of reinforcement-learning methods, but contrasts sharply with experimental observations in [Kaelbling, 1990, Whitehead, 1991b, Peng and Williams, 1992, Moore and Atkeson, 1993], that report good performance results.

These discrepancies motivate us to study how the performance of uninformed reinforcement-learning methods for goal-directed exploration tasks in deterministic domains is affected by different representations of the tasks. We do this in the context of Q-Learning [Watkins, 1989], probably the most popular reinforcement-learning method. In nondeterministic domains, Q-Learning assumes that nature always chooses the successor states with some time-invariant probability distribution that depends only on the current state and the executed action. Thus, nature chooses successor states randomly according to given transition probabilities, and Q-Learning attempts to maximize the average total reward. The transition probabilities do not need to be known. Q-Learning learns them implicitly. We study an uninformed, memoryless, on-line variant of Q-Learning [Whitehead, 1991a] with minimal look-ahead and a greedy action-selection step that always chooses the action with the maximal q-value in the current state. We study this reinforcement-learning method because it performs only a minimal amount of computation between action executions, choosing only which action to execute next, and basing this decision only on information local to the current state. If this inefficient reinforcement-learning method has a small complexity, then we can expect other reinforcement-learning methods to have a small complexity as well.

Q-Learning	(see Figure 2.27)
action-selection step (Line 3)	a := one-of $\arg\max_{a' \in A(s)} q(s,a')$
value-update step (Line 4)	$q(s,a) := (1-\alpha)q(s,a) + \alpha(r(s,a) + \gamma\, u(s'))$ $= (1-\alpha)q(s,a) + \alpha(r(s,a) + \gamma \max_{a' \in A(s')} q(s',a'))$, where s' is the successor state that results from the execution of action a in state s

For now, we leave the initial q-values unspecified. The value-update step uses a *learning rate* $\alpha \in (0,1]$ that determines how much $q(s,a)$ changes with every update. In probabilistic domains, the learning rate enables Q-Learning to average $r(s,a) + \gamma\, u(s')$ for all successor states $s' \in succ(s,a)$. In order for the q-values to converge to the desired values, the learning rate has to approach zero asymptotically, in a manner described in [Watkins and Dayan, 1992]. Since we restrict the analysis of Q-Learning to deterministic domains, averaging is not needed and consequently we set the learning rate to one throughout this section. (We discuss nondeterministic domains briefly in Sections 2.5.6.4 and 2.5.6.5.)

The complexity results in [Whitehead, 1991a] apply to Q-Learning as well: Uninformed Q-Learning can degenerate to a random walk until it reaches a goal state for the first time, even in deterministic domains. In this case, the number of action executions that it needs on average to reach a goal state can be exponential in the number of states. We, however, relate Q-Learning to Min-LRTA* and use the earlier complexity results to derive representations of goal-directed exploration tasks for which the complexity of uninformed Q-Learning in deterministic domains is polynomial. This can be achieved with only a simple change in the reward structure ("penalizing the agent for action executions") or the initialization of the q-values ("initializing optimistically"). Consequently, Q-Learning can be tractable without any need for augmentation of the method. These complexity results can help experimental researchers to choose representations for reinforcement-learning methods that enable them to solve reinforcement-learning tasks efficiently. They also provide the theoretical underpinnings for the experimental observations mentioned above that reported good performance results for some representations of reinforcement-learning tasks.

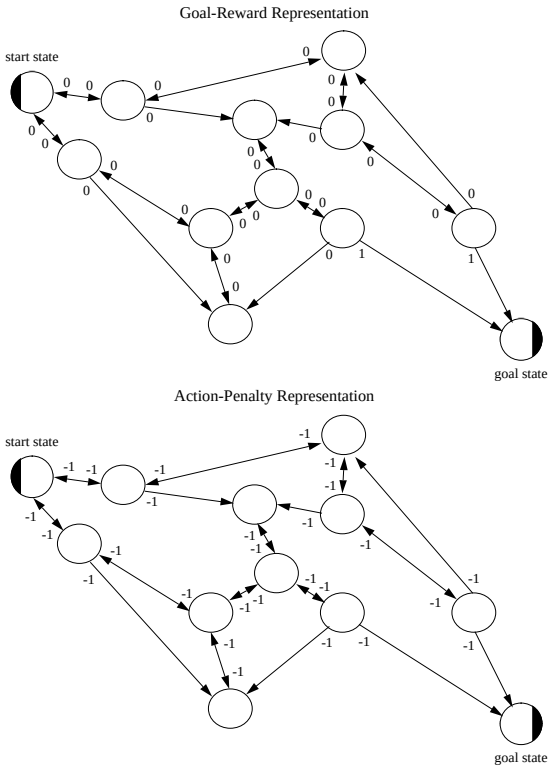


Figure 2.57: Reward Structures

2.5.6.1 Representations for Reinforcement-Learning Methods

When modeling goal-directed exploration tasks, one has to decide on their representation, where the *representation* determines both the immediate rewards that the reinforcement-learning method receives and the initialization of its q-values. In this section, we introduce possible reward structures and initializations, and discuss their properties. All of these representations have been used in the experimental reinforcement-learning literature to represent goal-directed reinforcement-learning tasks, that is, for learning shortest paths to a goal state. In later sections, we show that the choice of representation can have a large impact on the performance of Q-Learning.

For the reward structure, we have to decide on the immediate rewards $r(s, a)$ for $s \in S \setminus G$ and $a \in A(s)$. For the initial q-values, we have to decide on the q-values $q(s, a)$ for $s \in S \setminus G$ and $a \in A(s)$. Notice that, for goal-directed exploration tasks, the immediate rewards $r(s, a)$ and the q-values $q(s, a)$ for $s \in G$ and $a \in A(s)$ do not matter, because execution terminates when the agent reaches a goal state. We leave the immediate rewards $r(s, a)$ for $s \in G$ and $a \in A(s)$ unspecified. We use the q-values $q(s, a) = 0$ for $s \in G$ and $a \in A(s)$. Then, $u(s) = 0$ for $s \in G$, which reflects that execution terminates when the agent reaches a goal state and thus its total future reward is zero. In the following, we discuss possible reward structures and initializations of the q-values.

Reward Structures: The complexity of Q-Learning depends on properties of the reward structure. The reward structure should be suitable for goal-directed reinforcement-learning tasks, and thus it has to prefer shorter paths to a goal state over longer paths. Since reinforcement-learning methods determine policies that maximize the total reward of the agent, the reward structure must guarantee that states with smaller goal distances lead to larger optimal total rewards when the agent starts in them. Two different reward structures with this property have been used in the literature: the goal-reward representation and the action-penalty

representation. We define these reward structures in the next paragraphs. Both reward structures assign the same immediate reward to almost all of the actions. This immediate reward is zero for the goal-reward representation and minus one for the action-penalty representation, see Figure 2.57. We use this figure later to show how the two reward structures influence the behavior of Q-learning. (We discuss additional properties of the two reward structures in Section 4.6.4.)

The **goal-reward representation** rewards the agent for stopping in a goal state, but does not reward or penalize it for executing actions. This reward structure has been used in [Sutton, 1990, Whitehead, 1991a, Peng and Williams, 1992, Thrun, 1992b, Lin, 1993], among others. Formally,

$$r(s,a) = \begin{cases} 1 & \text{if } succ(s,a) \in G \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } s \in S \setminus G \text{ and } a \in A(s).$$

The optimal total undiscounted reward of every nongoal state s is one in safely explorable domains, and its optimal total discounted reward is $\gamma^{gd(s)-1} \leq 1$. Thus, discounting is necessary for learning shortest paths with this reward structure. If no discounting were used, the agent would always receive a total reward of one if it reaches a goal state and Q-Learning could not distinguish among paths of different lengths. If discounting is used, then the goal reward gets discounted with every action execution, and the agent tries to reach a goal state with as few action executions as possible to maximize the portion of the goal reward that it receives.

The **action-penalty representation** penalizes the agent for every action that it executes, but does not reward or penalize it for stopping in a goal state. This reward structure is *denser* than the goal-reward representation if goal states are relatively sparse (the agent receives nonzero immediate rewards more often). It has been used in [Barto *et al.*, 1989, Koenig, 1991, Barto *et al.*, 1995] among others. Formally,

$$r(s,a) = -1 \qquad \text{for } s \in S \setminus G \text{ and } a \in A(s).$$

The action-penalty representation can also be generalized to nonuniform immediate rewards in case the actions have different immediate costs (for example, different execution times).

The optimal total undiscounted reward of every nongoal state s is $-gd(s) \leq -1$, and its optimal total discounted reward is $(\gamma^{gd(s)} - 1)/(1-\gamma) \leq -1$. Thus, discounting can be used for learning shortest paths with this reward structure, but is not necessary. In both cases, the agent tries to reach a goal state with as few action executions as possible to minimize the amount of penalty that it receives. Therefore, the action-penalty representation provides additional freedom for choosing the parameters of Q-Learning. The q- and u-values are integers if no discounting is used, otherwise they are reals. Integers have the advantage over reals that they need less memory space and can be stored without a loss in precision.

Initial Q-Values: The complexity of Q-Learning also depends on properties of the initial q-values. We study uninformed Q-Learning. Thus, the q-values cannot convey any information about the domain beyond the information that is already available to Q-Learning by other means. Uninformed Q-Learning is able to detect whether the current state is a goal state. However, initially it is unable to distinguish among different nongoal states or among different actions in the same state. Thus, all q-values of actions in nongoal states have to be initialized identically for uninformed Q-Learning.

Definition 4 *Q-values are* uniformly initialized *with x (or, synonymously, x-initialized), if and only if initially*

$$q(s,a) = \begin{cases} 0 & \text{if } s \in G \\ x & \text{otherwise} \end{cases} \qquad \text{for all } s \in S \text{ and } a \in A(s).$$

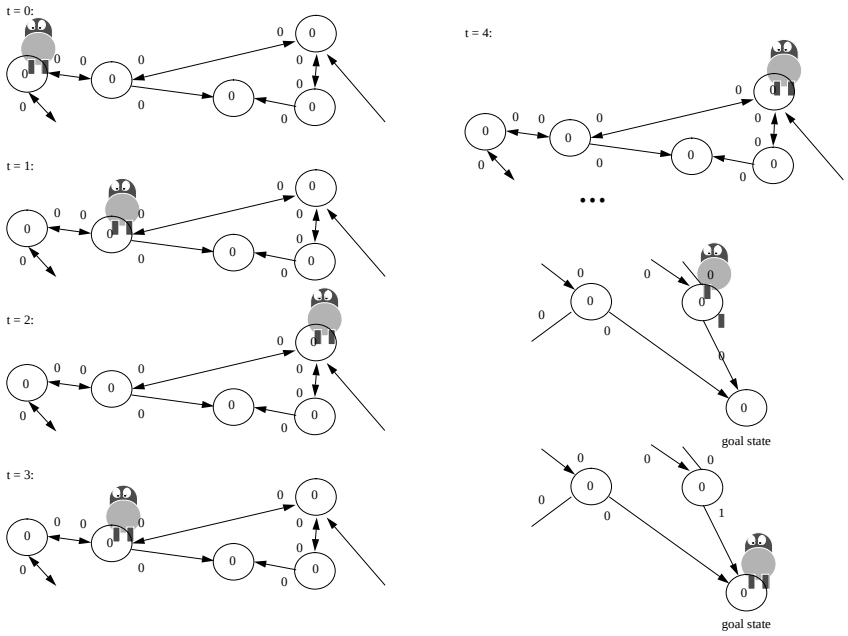


Figure 2.58: Discounted Zero-Initialized Q-Learning with the Goal-Reward Representation

Remember that we use the q-values $q(s, a) = 0$ for $s \in G$ and $a \in A(s)$, because then $u(s) = 0$ for $s \in G$, which reflects that execution terminates for goal-directed exploration tasks when the agent reaches a goal state and thus its total future reward is zero.

Until now, we have assumed that all uninformed methods are zero-initialized. If Q-Learning is x-initialized with $x \neq 0$, the q-values of actions in goal states and nongoal states are initialized differently. It is important to understand that this particular initialization does not convey any information about the domain, since Q-Learning is able to distinguish whether its current state is a goal state or not. Consequently, uniformly initialized Q-Learning is uninformed.

2.5.6.2 An Intractable Representation

In this section, we study Q-Learning that operates on the goal-reward representation (which implies that discounting has to be used) and is zero-initialized. We show that this representation makes the goal-directed exploration tasks intractable.

Discounted zero-initialized Q-Learning with the goal-reward representation receives a nonzero immediate reward only if an action execution results in a goal state. Figure 2.58 shows an example. The u-values are shown in the states, and the q-values label the actions. Thus, during the search for a goal state, all q-values remain zero. Q-Learning does not even remember which actions it has already executed, and the action-selection step has no information on which to base its decision about which action to execute next. If Q-Learning had a systematic bias for actions (for example, if it broke ties systematically according to a predetermined ordering on $A(s)$ for all states s) then it could cycle in the domain forever. We therefore assume that it chooses randomly from the actions available in the current state, in which case it performs a random walk. The average-case performance of random walks can be exponential in the number of states n (Theorem 9). The following corollary follows:

Corollary 20 *Discounted zero-initialized Q-Learning with the goal-reward representation has an average-case performance for reaching a goal state that is at least exponential in n over all reasonable, deterministic*

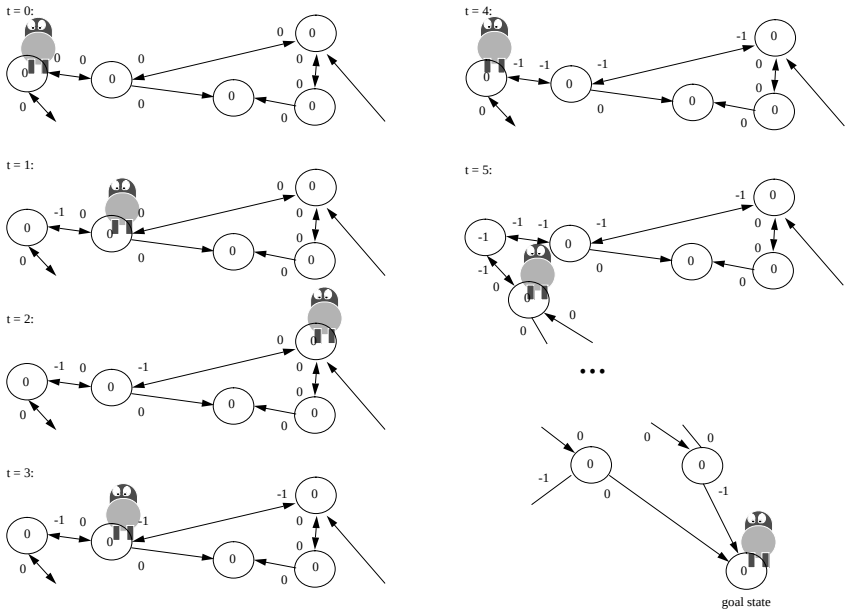


Figure 2.59: Undiscounted Zero-Initialized Q-Learning with the Action-Penalty Representation

domains.

This observation was made by Whitehead [Whitehead, 1991a], who then explored cooperative reinforcement-learning methods to decrease the complexity. Subsequently, Thrun [Thrun, 1992a] showed that even non-cooperative reinforcement-learning methods can have polynomial complexity if they are extended with a mechanism that he calls *counterbased exploration*. Counterbased Q-Learning maintains a second kind of state-action values, called *counters*, in addition to the q-values. Thrun was able to specify action-selection and value-update steps that use these counters and achieve polynomial complexity. There are other speed-up methods that can improve the performance of reinforcement-learning methods, such as the action-replay method [Lin, 1992]. However, this method does not change the q-values before a goal state has been reached for the first time when it is applied to discounted zero-initialized Q-Learning with the goal-reward representation and, thus, cannot be used to reduce its complexity for the reinforcement-learning tasks studied here.

2.5.6.3 Tractable Representations

In the following, we show that one does not need to augment Q-Learning to reduce its complexity. Uninformed Q-Learning is tractable if one uses either the action-penalty representation or nonzero initial q-values. The intuitive explanation is that the q-values change immediately in both cases, starting with the first action execution. This way, Q-Learning remembers something about the previous action executions. The complexity results show formally by how much the complexity is reduced.

Zero-Initialized Q-Values with the Action-Penalty Representation: In the following, we study Q-Learning that operates on the action-penalty representation and is zero-initialized. We first show that undiscounted Q-Learning with this representation is tractable and then show how the analysis can be applied to discounted Q-Learning. In both cases, the q-values change immediately since Q-Learning receives a nonzero immediate reward after every action execution. Figure 2.59 shows an example. The u-values are shown in the states, and the q-values label the actions. Notice that, different from Figure 2.58, the q-values now prevent the agent from repeating a previous action execution at $t = 3$.

Some reinforcement-learning methods and LRTA*-type real-time search methods have in common that they are asynchronous incremental dynamic programming methods [Barto *et al.*, 1989]. In fact, in deterministic domains, undiscounted zero-initialized Q-Learning with the action-penalty representation and learning rate one is identical to zero-initialized Min-LRTA* with the simplified value-update step. Thus, there is a strong connection between reinforcement learning and LRTA*-type real-time search. We exploit this connection to transfer the complexity results about LRTA*-type real-time search to reinforcement learning. Since the complexity of zero-initialized Min-LRTA* is tight at $O(n^3)$ action executions over all reasonable domains (Theorem 15), the following corollary follows:

Corollary 21 *Undiscounted zero-initialized Q-Learning with the action-penalty representation and learning rate one has a tight complexity of $O(n^3)$ action executions for reaching a goal state over all reasonable, deterministic domains.*

We can now reduce the complexity analysis of discounted zero-initialized Q-Learning with the action-penalty representation to the one of undiscounted zero-initialized Q-Learning with the same reward structure. Consider the following strictly monotonically increasing bijection from the q-values of discounted Q-Learning to the q-values of undiscounted Q-Learning. In the following, the time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update step of Q-Learning (Line 4 in Figure 2.27). For every $s \in S$ and $a \in A(s)$, map the q-value $q_1^t(s,a)$ of discounted Q-Learning to the q-value $q_2^t(s,a) = -\log_\gamma[1+(1-\gamma)q_1^t(s,a)]$ of undiscounted Q-Learning. This relationship continues to hold if it holds for the initial q-values: Assume that the relationship holds at time t and both discounted and undiscounted Q-Learning are in the same state. Since the mapping between q-values is strictly monotonically increasing, and both discounted and undiscounted Q-Learning always execute the action with the maximal q-value in the current state, they always choose the same action for execution and thus reach the same successor state (if ties are broken identically). If they execute action a^t in state $s^t \in S \setminus G$, then $q_1(s^t,a^t)$ and $q_2(s^t,a^t)$ are the only q-values that change between time t and $t+1$, and

$$
\begin{aligned}
q_2^{t+1}(s^t,a^t) &= -1 + u_2^t(succ(s^t,a^t)) \\
&= -1 + \max_{a' \in A(succ(s^t,a^t))} q_2^t(succ(s^t,a^t),a') \\
&= -1 + \max_{a' \in A(succ(s^t,a^t))} [-\log_\gamma[1+(1-\gamma)q_1^t(succ(s^t,a^t),a')]] \\
&= \max_{a' \in A(succ(s^t,a^t))} [-\log_\gamma[\gamma(1+(1-\gamma)q_1^t(succ(s^t,a^t),a'))]] \\
&= \max_{a' \in A(succ(s^t,a^t))} [-\log_\gamma[1+(1-\gamma)(-1+\gamma q_1^t(succ(s^t,a^t),a'))]] \\
&= -\log_\gamma[1+(1-\gamma)(-1+\gamma \max_{a' \in A(succ(s^t,a^t))} q_1^t(succ(s^t,a^t),a'))] \\
&= -\log_\gamma[1+(1-\gamma)(-1+\gamma u_1^t(succ(s^t,a^t)))] \\
&= -\log_\gamma[1+(1-\gamma)q_1^{t+1}(s^t,a^t)].
\end{aligned}
$$

Consequently, the relationship between the q-values continues to hold, and discounted and undiscounted Q-Learning behave identically. Thus, the complexity analysis of undiscounted Q-Learning also applies to discounted Q-Learning. If discounted Q-Learning is zero-initialized, then undiscounted Q-Learning is zero-initialized as well, since $q_2^0(s,a) = -\log_\gamma[1+(1-\gamma)q_1^0(s,a)] = -\log_\gamma[1+(1-\gamma)\times 0] = 0$ for all $s \in S \setminus G$ and $a \in A(s)$. This implies that it has a tight complexity of $O(n^3)$ action executions over all reasonable, deterministic domains (Corollary 21). The following corollary follows:

Corollary 22 *Discounted zero-initialized Q-Learning with the action-penalty representation and learning rate one has a tight complexity of $O(n^3)$ action executions for reaching a goal state over all reasonable, deterministic domains.*

One-Initialized Q-Values with the Goal-Reward Representation: Zero-initialized Q-Learning with the action-penalty representation is tractable because the q-values change immediately. Since the value-update step of Q-Learning updates the q-values using both the immediate reward and the q-values of the successor state, this suggests that one can achieve tractability not only by changing the immediate rewards, but also by initializing the q-values differently. In the following, we show that, indeed, Q-Learning is also tractable if the goal-reward representation is used (which implies that discounting has to be used as well) and the q-values are $1/\gamma$- or one-initialized. The latter initialization has the advantage that it does not depend on the discount factor.

The complexity analysis of discounted one-initialized (or $1/\gamma$-initialized) Q-Learning with the goal-reward representation can be reduced to the one of undiscounted (minus one)-initialized (or zero-initialized) Q-Learning with the action-penalty representation. We proceed as we proceeded for discounted zero-initialized Q-Learning with the action-penalty representation. This time, we consider the following strictly monotonically increasing bijection from the q-values of discounted Q-Learning with the goal-reward representation to the q-values of undiscounted Q-Learning with the action-penalty representation. For every $s \in S \setminus G$ and $a \in A(s)$, map the q-value $q_1^t(s,a)$ of discounted Q-Learning with the goal-reward representation to the q-value $q_2^t(s,a) = -1 - \log_\gamma q_1^t(s,a)$ of undiscounted Q-Learning with the action-penalty representation. This relationship continues to hold if it holds for the initial q-values: Assume that the relationship holds at time t and both discounted Q-Learning with the goal-reward representation and undiscounted Q-Learning with the action-penalty representation are in the same state. Since the mapping between q-values is strictly monotonically increasing and both discounted Q-Learning with the goal-reward representation and undiscounted Q-Learning with the action-penalty representation always execute the action with the maximal q-value in the current state, they always choose the same action for execution and thus reach the same successor state (if ties are broken identically). If they execute action a^t in state $s^t \in S \setminus G$, then $q_1(s^t, a^t)$ and $q_2(s^t, a^t)$ are the only q-values that change between time t and $t+1$, and

- if $succ(s^t, a^t) \in S \setminus G$:

$$\begin{aligned}
q_2^{t+1}(s^t, a^t) &= -1 + u_2^t(succ(s^t, a^t)) \\
&= -1 + \max_{a' \in A(succ(s^t, a^t))} q_2^t(succ(s^t, a^t), a') \\
&= -1 + \max_{a' \in A(succ(s^t, a^t))} [-1 - \log_\gamma q_1^t(succ(s^t, a^t), a')] \\
&= -1 + \max_{a' \in A(succ(s^t, a^t))} [-\log_\gamma [\gamma q_1^t(succ(s^t, a^t), a')]] \\
&= -1 - \log_\gamma [\gamma \max_{a' \in A(succ(s^t, a^t))} q_1^t(succ(s^t, a^t), a')]] \\
&= -1 - \log_\gamma [\gamma u_1^t(succ(s^t, a^t), a')] \\
&= -1 - \log_\gamma q_1^{t+1}(s^t),
\end{aligned}$$

- if $succ(s^t, a^t) \in G$:

$$\begin{aligned}
q_2^{t+1}(s^t, a^t) &= -1 + u_2^t(succ(s^t, a^t)) \\
&= -1 + \max_{a' \in A(succ(s^t, a^t))} q_2^t(succ(s^t, a^t), a') \\
&= -1 + \max_{a' \in A(succ(s^t, a^t))} 0 \\
&= -1 \\
&= -1 - \log_\gamma 1 \\
&= -1 - \log_\gamma [1 + \gamma \max_{a' \in A(succ(s^t, a^t))} 0]
\end{aligned}$$

$$\begin{aligned} &= -1 - \log_\gamma [1 + \gamma \max_{a' \in A(succ(s^t, a^t))} q_1^t(succ(s^t, a^t), a')] \\ &= -1 - \log_\gamma [1 + \gamma u_1^t(succ(s^t, a^t), a')] \\ &= -1 - \log_\gamma q_1^{t+1}(s^t). \end{aligned}$$

Consequently, the relationship between the q-values continues to hold, and discounted Q-Learning with the goal-reward representation and undiscounted Q-Learning with the action-penalty representation behave identically. Thus, the complexity analysis of undiscounted Q-Learning with the action-penalty representation also applies to discounted Q-Learning with the goal-reward representation. If discounted Q-Learning with the goal-reward representation is $1/\gamma$-initialized, then undiscounted Q-Learning with the action-penalty representation is zero-initialized, since $q_2^0(s, a) = -1 - \log_\gamma q_1^0(s, a) = -1 - \log_\gamma [1/\gamma] = 0$ for all $s \in S \setminus G$ and $a \in A(s)$. This implies that it has a tight complexity of $O(n^3)$ action executions over all reasonable, deterministic domains (Corollary 21).

Similarly, if discounted Q-Learning with the goal-reward representation is one-initialized, then undiscounted Q-Learning with the action-penalty representation is (minus one)-initialized, since $q_2^0(s, a) = -1 - \log_\gamma q_1^0(s, a) = -1 - \log_\gamma 1 = -1$ for all $s \in S \setminus G$ and $a \in A(s)$. It is easy to show that undiscounted (minus one)-initialized Q-Learning with the action-penalty representation has a tight complexity of $O(n^3)$ action executions for reaching a goal state over all reasonable, deterministic domains, since it and undiscounted zero-initialized Q-Learning with the action-penalty representation behave identically until they reach a goal state (if ties are broken identically). The following corollary follows:

Corollary 23 *Discounted one-initialized Q-Learning with the goal-reward representation and learning rate one has a tight complexity of $O(n^3)$ action executions for reaching a goal state over all reasonable, deterministic domains.*

2.5.6.4 Other Reinforcement-Learning Methods

We have related Q-Learning to Min-LRTA* and then used the complexity results about Min-LRTA* to analyze the performance of Q-Learning. We would like to point out briefly (but not prove) that one can proceed in a similar way for other on-line reinforcement-learning methods as well, such as Q̂-Learning and Value-Iteration.

Q̂-Learning: Q̂-Learning ("Q-hat-learning") [Heger, 1996] is similar to Q-Learning. In nondeterministic domains, Q-Learning assumes that nature always chooses the successor states with some time-invariant (but unknown) probability distribution that depends only on the current state and the executed action. Thus, nature chooses successor states randomly according to given transition probabilities, and Q-learning attempts to maximize the average total reward. Q̂-Learning, on the other hand, assumes that nature always chooses the worst successor state. Thus, nature is an opponent, and Q̂-Learning attempts to maximize the worst-case total reward. Which reinforcement-learning method to use for learning an optimal policy depends on the risk attitude of the agent (Chapter 4). Consider an uninformed, memoryless, on-line variant of Q̂-Learning with minimal look-ahead and a greedy action-selection step that always chooses the action with the maximal q-value in the current state.

Q̂-Learning	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of $\arg\max_{a' \in A(s)} q(s, a')$
value-update step (Line 4)	$q(s, a) := \min(q(s, a), r(s, a) + \gamma u(s')) = \min(q(s, a), r(s, a) + \gamma \max_{a' \in A(s)} q(s', a'))$, where s' is the successor state that results from the execution of action a in state s

Q̂-Learning requires the q-values to be initialized optimistically. It needs less information about the domain than Q-Learning (since it does not need to know how nature behaves) and converges faster. In particular, it does not need to execute each action in a state often enough to get a representative distribution over the

successor states and has no need for a learning rate, even in probabilistic domains [Heger, 1996]. Correctness and convergence results for $\hat{Q}$-Learning are given in [Heger, 1994].

In deterministic domains, undiscounted zero-initialized $\hat{Q}$-Learning with the action-penalty representation is identical to zero-initialized Min-LRTA* with the unsimplified value-update step. In nondeterministic domains, undiscounted zero-initialized $\hat{Q}$-Learning with the action-penalty representation is identical to a combination of zero-initialized Min-Max LRTA* (minimax principle) and zero-initialized Min-LRTA* (minimal look-ahead). This makes it possible to derive the following complexity result for $\hat{Q}$-Learning [Koenig and Simmons, 1995a, Koenig and Simmons, 1996b]:

Undiscounted or discounted zero-initialized $\hat{Q}$-Learning with the action-penalty representation and discounted one-initialized $\hat{Q}$-Learning with the goal-reward representation have a tight complexity of $O(n^3)$ action executions for reaching a goal state over all reasonable, nondeterministic domains and over all reasonable, deterministic domains.

Value-Iteration: Value-Iteration [Bellman, 1957] is similar to LRTA*. It does not use the q-values, but rather the u-values directly, and thus has a larger look-ahead than Q-Learning. In nondeterministic domains, Value-Iteration assumes (like Q-Learning) that nature always chooses the successor states with some time-invariant probability distribution that depends only on the current state and the executed action. Thus, nature chooses successor states randomly according to given transition probabilities, and Value-Iteration attempts to maximize the average total reward. $p(s'|s,a)$ denotes the transition probability that the system transitions from state s to state s' when action a is executed. If the transition probabilities are unknown, they have to be learned explicitly. Consider an uninformed, memoryless, on-line variant of value-iteration with look-ahead one and a greedy action-selection step that always chooses the action that maximizes the sum of the immediate reward and the discounted average u-value of the successor state.

Value-Iteration	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of $\arg\max_{a' \in A(s)} [r(s,a) + \gamma \sum_{s' \in S} [p(s'\|s,a) u(s')]]$
value-update step (Line 4)	$u(s) := r(s,a) + \gamma \sum_{s' \in S} [p(s'\|s,a) u(s')]$

Correctness and convergence results for value-iteration are given in [Barto *et al.*, 1989].

In deterministic domains, undiscounted zero-initialized Value-Iteration with the action-penalty representation is identical to zero-initialized LRTA* with the simplified value-update step. This makes it possible to derive the following complexity result for Value-Iteration [Koenig and Simmons, 1995b]:

Undiscounted or discounted zero-initialized Value-Iteration with the action-penalty representation and discounted one-initialized Value-Iteration with the goal-reward representation have a tight complexity of $O(n^2)$ action executions for reaching a goal state over all safely explorable, deterministic domains.

2.5.6.5 Reinforcement Learning in Probabilistic Domains

Reinforcement-learning methods cannot only be used in deterministic domains but also in nondeterministic domains. We have mentioned complexity results about $\hat{Q}$-Learning in nondeterministic domains, but have not analyzed Q-Learning and Value-Iteration in such domains. Whereas $\hat{Q}$-Learning attempts to maximize the worst-case total reward, Q-learning and Value-Iteration attempt to maximize the average total reward – this makes their analysis more difficult. Such an analysis would be interesting, however, since reinforcement-learning methods are often applied to probabilistic domains. Although one can hope that the performance of Q-Learning and Value-Iteration in deterministic domains is indicative of their performance in probabilistic domains, extending our complexity results to probabilistic domains is future work. In the following, we describe three problems that arise for Q-Learning and Value-Iteration in probabilistic domains that a formal analysis needs to address:

First, in probabilistic domains, Q-Learning always has to learn the transition probabilities. It learns them implicitly. Value-Iteration has to learn the transition probabilities if they are not given. It learns them

explicitly. For Q-Learning, the problem arises that the learning rate has to be smaller than one and each action has to be executed a large number of times for its q-value to converge even if the u-values of all successor states are already correct. Furthermore, the number of action executions required depends on the size of the initial q-values. For Value-Iteration, the problem arises that each action has to be sampled a large number of times to estimate the probabilities reliably.

Second, in probabilistic domains, admissible u-values do not necessarily remain admissible for Q-Learning. Neither are they guaranteed to remain admissible for Value-Iteration if the transition probabilities have to be learned. This is due to the fact that the transition probabilities are unknown and only (implicit or explicit) estimates are available. Although the difference between a transition probability and its estimate approaches zero with probability one as the sample size gets larger (provided that a reasonable estimation method is used), it can be large initially. As a consequence, it can happen that the u-value of a state s drops below its negative average goal distance $gd(s)$ even if the u-values of all of its successor states are admissible. As an example, consider undiscounted zero-initialized Value-Iteration with the action-penalty representation and assume that the transition frequencies are used to estimate the transition probabilities (maximum-likelihood estimation). Assume further that there is a state s in which only one action a can be executed. Executing this action results in state s_1 with probability 0.5 and in state s_2 with the complementary probability, where $gd(s_1) > gd(s_2)$. If $u(s_1) = -gd(s_1)$, $u(s_2) = -gd(s_2)$, and executing action a in state s has resulted n times in successor state s_1 and never in successor state s_2, then $u(s) = -1 + u(s_1) = -1 - gd(s_1) < -1 - 1/2\,gd(s_1) - 1/2gd(s_2) = -gd(s)$. Thus, $u(s) < -gd(s)$, and state s looks worse than it actually is. A similar problem occurs for Q-Learning, where the q-value $q(s, a)$ approaches $-1 - gd(s_1)$ and can get smaller than $-1-1/2\,gd(s_1)-1/2gd(s_2)$. In this case, again, $u(s) = q(s, a) < -1-1/2\,gd(s_1)-1/2gd(s_2) = -gd(s)$. One consequence of this problem is that the action-selection step of most reinforcement-learning methods can no longer be greedy in probabilistic domains. Greedy action selection for Q-Learning, for example, always chooses the action with the maximal q-value in the current state. In deterministic domains, this behavior always exploits but at the same time explores sufficiently often, where *exploitation* means to behave optimally according to the current knowledge and *exploration* means to acquire new knowledge. In probabilistic domains, however, greedy action selection might force Q-Learning to avoid executing actions whose q-values make them look worse than they actually are. This prevents Q-Learning from updating (and thus correcting) these q-values. Kaelbling [Kaelbling, 1990] calls this phenomenon "sticking." Consequently, there is a potential trade-off between exploitation and exploration in probabilistic domains. Exploration consumes time, but ensures that all q-values get updated, which allows reinforcement-learning methods to find an optimal policy and then solve the tasks faster. A traditional strategy, then, is to exploit most of the time and to explore only from time to time. Thrun [Thrun, 1992b] summarizes methods for dealing with the exploration-exploitation conflict in probabilistic domains.

Third, in probabilistic domains, optimal policies can have cycles (that is, the same state is visited more than once with positive probability). This has two consequences if the optimal policy is cyclic, both for Q-Learning and Value-Iteration, even if Value-Iteration knows the transition probabilities: First, the q-values or u-values never converge completely. Second, the number of action executions that are needed on average to reach a goal state can be exponential in n even for an agent that acts optimally [Thrun, 1992b]. In this case, the average-case performance of any reinforcement-learning method is at least exponential in n and one has to investigate ways to separate the inherent complexity of a given reinforcement-learning task from the performance of individual reinforcement-learning methods.

2.5.6.6 Summary of Results on Reinforcement Learning

We studied the behavior of uninformed reinforcement-learning methods that solve goal-directed reinforcement-learning tasks until they reach a goal state for the first time. When modeling goal-directed reinforcement-learning tasks, one has to decide on an appropriate representation, which consists of both the immediate rewards and the initial q-values. We showed that the choice of representation can have a large impact on the performance of Q-Learning and related reinforcement-learning methods.

We considered two reward structures that have been used in the experimental literature to learn shortest paths to a goal state, the goal-reward representation and the action-penalty representation. The goal-reward representation rewards an agent for stopping in a goal state, but does not reward or penalize it for executing actions. The action-penalty representation, on the other hand, penalizes the agent for every action that it executes, but does not reward or penalize it for stopping in a goal state. Zero-initialized Q-Learning with the goal-reward representation provides only sparse rewards. Even in deterministic domains, it performs a random walk at best. Although random walks always reach a goal state with probability one and finite average-case performance over all reasonable domains, the number of action executions that they need on average to reach a goal state can be exponential in the number of states. Furthermore, speed-up methods such as the action-replay method do not improve the performance of Q-Learning in this case. This provides motivation for making the reward structure dense. Since the value-update step of Q-Learning updates the q-values using both the immediate rewards and the q-values of the successor states, this can be achieved by changing the reward structure or initializing the q-values differently. We showed that both undiscounted and discounted zero-initialized Q-Learning with the action-penalty representation and discounted one-initialized Q-Learning with the goal-reward representation are tractable. For these representations, uninformed Q-Learning has a tight complexity that is cubic in the number of states over all reasonable, deterministic domains. Different from the intractable case, these two representations initialize the q-values optimistically, that is, the initial q-value $q(s, a)$ of every nongoal state is at least as large as the total (discounted or undiscounted) reward that Q-Learning achieves if it starts in state s, executes action a, and then acts optimally.

To summarize, reinforcement-learning methods are tractable for suitable representations. The complexity results can help experimental researchers to choose representations for reinforcement-learning methods that enable them to solve reinforcement-learning tasks efficiently. Even for reinforcement-learning tasks that cannot be reformulated as goal-directed reinforcement-learning tasks in deterministic domains, they suggest that the performance can be improved by making the reward structure dense, for example by increasing the initial q-values sufficiently. Some of our results have also been used to study extensions of reinforcement-learning methods. For example, they are used by Lin [Lin, 1993] to study "Hierarchical Q-Learning," that he applied to learning robot control.

2.6 Extensions

Throughout this chapter, we measured distances and thus the performance of LRTA*-type real-time search methods in action executions, which is reasonable if every action has the same immediate cost, for example, can be executed in about the same amount of time. Sometimes, however, actions have different cost. We presented complexity results for the special case where all of the immediate rewards are uniformly minus one, but the LRTA*-type real-time search methods and their complexity results can be generalized to arbitrary strictly negative reward structures, including those with nonuniform immediate rewards [Koenig and Simmons, 1992]. For example, the following table presents the action-selection and simplified value-update steps of Min-LRTA* if the immediate rewards are nonuniform. Notice the similarity to the reinforcement-learning methods in Section 2.5.6, which we stated already for nonuniform immediate rewards.

Min-LRTA*	(see Figure 2.27)
action-selection step (Line 3)	$a :=$ one-of $\arg\max_{a' \in A(s)} q(s, a')$
value-update step (Line 4)	$q(s, a) := r(s, a) + u(succ(s, a))$
	$= r(s, a) + \max_{a' \in A(succ(s,a))} q(succ(s, a), a')$

Throughout this chapter, we also assumed that every action execution in a nongoal state necessarily results in a state change. If we drop this assumption, then, different from Section 2.3.2, Min-Max LRTA* with arbitrary look-ahead (Figure 2.10) and local search space $S_{lss} = \{s\}$ no longer behaves identically to Min-Max LRTA* with look-ahead one (Figure 2.9). Furthermore, different from Section 2.5.2.2, Min-Max LRTA* with look-ahead one in the transformed domain no longer behaves identically to Min-LRTA* in the original domain.

This affects the complexity results about Min-Max LRTA* with look-ahead one and Min-LRTA*, but not the big-O versions of their complexity results. For example, the complexity of zero-initialized Min-Max LRTA* with look-ahead one doubles over all safely explorable domains if the requirement is dropped that every action execution in a nongoal state necessarily results in a state change [Koenig and Simmons, 1995b]. Further complexity results about Min-Max LRTA* with look-ahead one and Min-LRTA* without the assumption that every action execution in a nongoal state necessarily results in a state change are presented in [Koenig and Simmons, 1992] and [Koenig and Simmons, 1993a]. The complexity results about Min-Max LRTA* with arbitrary look-ahead (including $S_{lss} = \{s\}$) remain unaffected because, different from Min-Max LRTA* with look-ahead one, Min-Max LRTA* with arbitrary look-ahead and initially admissible u-values executes only actions in safely explorable domains whose execution necessarily results in a state change.

2.7 Future Work

In this chapter, we provided first steps towards understanding why LRTA*-type real-time search methods work, when they work, and how well they work. More work is required in this direction, especially since we showed that LRTA*-type real-time search methods and traditional search methods differ in their properties. Both more theoretical work and thorough experimental work are needed.

First, it is important to study the performance of LRTA*-type real-time search methods not only for traditional search domains from artificial intelligence (such as sliding-tile puzzles and grid-worlds) but also for more realistic applications. This avoids the problem that the domain properties of the test-beds might not be representative of the domain properties of the domains of interest. We illustrated the performance of LRTA*-type real-time search methods on robot-navigation tasks in mazes. Other interesting applications of LRTA*-type real-time search methods include the exploration of unknown environments with robots [Koenig and Smirnov, 1996], package routing in computer networks [Littman and Boyan, 1993], and on-line scheduling [Pemberton, 1995], such as air traffic control and factory scheduling.

Second, it is important to study in more detail how the look-ahead, the heuristic knowledge of a domain, and domain properties influence the performance of LRTA*-type real-time search methods.

Third, it is important to study the properties of other existing LRTA*-type real-time search methods, such as RTA* [Korf, 1987], a variant of LRTA*, and eventually characterize the properties of whole classes of LRTA*-type real-time search methods.

Fourth, it is important to develop new LRTA*-type real-time search methods, including LRTA*-type real-time search methods that can guarantee that their memory requirements are smaller than the number of states and LRTA*-type real-time search methods that are able to use their problem-solving experience to improve their parameters. For example, while Min-Max LRTA* works with arbitrary look-aheads, it is important to investigate how it can adapt its look-ahead dynamically to different domains so that it minimizes the sum of planning and plan execution time.

Fifth, it is important to study how LRTA*-type real-time search methods relate to other search methods that interleave planning and plan execution. This would allow the transfer of ideas and results across different areas of artificial intelligence, robotics, and theoretical computer science. We investigated the relationship of LRTA*-type real-time search methods and reinforcement-learning methods. Other methods that are similar to LRTA*-type real-time search methods include assumptive planning [Stentz, 1995, Nourbakhsh, 1996], deliberation scheduling (including any-time methods [Boddy and Dean, 1989]), sensor-based planning [Choset and Burdick, 1994], plan-envelope methods [Dean *et al.*, 1995], and various robot exploration and localization methods.

2.8 Conclusions

This chapter studied LRTA*-type real-time search methods. These are search methods that interleave planning and plan execution by searching forward from the current state of the agent and associate information with the states. They had been used as an alternative to traditional search methods and shown to speed up problem solving in deterministic domains. We illustrated that they can also be used to resolve uncertainty caused by nondeterminism and speed up problem solving in nondeterministic domains. To this end, we developed Min-Max LRTA*, an efficient LRTA*-type real-time search method for nondeterministic domains. Min-Max LRTA* allows for fine-grained control over how much planning to do between plan executions, uses heuristic knowledge to guide planning, and improves its performance over time as it solves similar planning tasks, until its performance is at least worst-case optimal. Since Min-Max LRTA* partially relies on observing the actual successor states of action executions, it does not plan for all possible successor states and thus can still have computational advantages even over several search episodes compared to a complete minimax search if nature is not as malicious as a minimax search assumes and some successor states do not occur in practice.

We also studied when Min-Max LRTA* and related real-time search methods work, why they work, and how well they work. We derived an upper bound on the complexity of Min-Max LRTA* and showed that it is tight for uninformed and fully informed Min-Max LRTA* with look-ahead one. We also showed that the performance of Min-Max LRTA* can decline as Min-Max LRTA* becomes better informed, although it improves eventually. We then studied uninformed LRTA*-type real-time search methods with minimal look-ahead in deterministic domains. These LRTA*-type real-time search methods do not even project one action execution ahead. We showed that Eulerian domains (a superset of undirected domains) are easy to search with a variety of these LRTA*-type real-time search methods, even those that can be intractable, and introduced reset state spaces, quicksand state spaces, and "complex state spaces" that are not as easy to search. We also introduced the product of the number of actions (or states) and the maximal goal distance as a good (but not perfect) measure of task size that determines how easy it is to search domains with a variety of LRTA*-type real-time search methods. These results can help to distinguish easy LRTA*-type real-time search tasks from hard ones, which can help experimental researchers to decide when to use LRTA*-type real-time search methods. For example, sliding-tile puzzles are undirected (and thus Eulerian). They also have a small number of actions and a small maximal goal distance (relative to the number of states). This makes them easy to solve with a variety of LRTA*-type real-time search methods. The results can also be used to choose test-beds for experimenting with LRTA*-type real-time search methods, reporting their results, and interpreting the results reported by others, and to choose representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods.

Chapter 3

Acting with POMDPs

Office-delivery robots have to navigate reliably despite a substantial amount of uncertainty. In Chapter 2, we studied goal-directed robot navigation in mazes. The robot had perfect actuators, perfect sensors, and perfect knowledge of the maze, but was uncertain about its start pose. However, office environments are more complex than mazes and thus office-delivery robots have to deal with more uncertainty, including uncertainty in actuation, uncertainty in sensing and sensor data interpretation, uncertainty in the start pose of the robot, and uncertainty about distances in the environment. A robot has to reach its destinations despite this uncertainty. This problem could be solved by either adapting the environment to the robot or the robot to the environment. Adapting the environment to the robot can, for example, be accomplished by placing beacons or bar codes in the environment. Adapting the robot to the environment can be done by placing a large number of very sophisticated sensors on-board the robot. Our goal is, however, to achieve reliable robot navigation in an inexpensive way without changing the environment.

Our approach to this problem is to use probabilities to model the uncertainty explicitly. Thus, while the method of Chapter 2 (Min-Max LRTA*) assumed that probabilities were not available, used sets of poses to maintain a belief in the current robot pose, and attempted to minimize the worst-case plan-execution cost, the method in this chapter assumes that probabilities are available (or can be learned), uses probability distributions over poses to maintain a belief in the current robot pose, and attempts to minimize the average plan-execution cost.

We use *partially observable Markov decision process models* (POMDPs) from operations research. POMDPs are finite state automata with transition and state uncertainty that can account for all of the navigation uncertainty mentioned above. We show how to use them to build a whole architecture for robot navigation that provides a uniform framework with an established theoretical foundation for pose estimation, path planning (including planning when, where, and what to sense), control during navigation, and learning. In addition, both the POMDP and the generated information (such as the pose information and the plans) can be utilized by higher-level planning modules, such as task planners.

Our POMDP-based navigation architecture uses a compiler that automatically produces POMDPs from *topological maps* (graphs of nodes and edges that specify landmarks, such as corridor junctions, and how they connect), actuator and sensor models, and uncertain knowledge of the office environment. The resulting POMDPs seamlessly integrate topological and distance information, and enable the robot to utilize as much, or as little, distance information as is available. Pose estimation, that is, the process of determining the position and orientation of the robot, can utilize all available sensor information, including landmarks sensed and distance traveled, deals easily with sensor noise and distance uncertainty, and deteriorates gracefully with the quality of the models. Finally, learning methods can be used to improve the models on-line, that is, while the robot is carrying out its office-navigation tasks.

We use the POMDP-based navigation architecture on a daily basis on Xavier [Simmons *et al.*, 1997]. Our experiments show that the architecture leads to robust long-term autonomous navigation in office environments

(with corridors, foyers, and rooms), significantly outperforming the landmark-based navigation technique that was used previously [Simmons, 1994a].

To summarize, we study how to act robustly despite uncertainty. Our main contribution is the following: We show how to model the problem as a POMDP from operations research. We develop a POMDP-based navigation architecture that performs pose estimation, planning, acting, and learning. This includes developing efficient methods for POMDP planning and learning that have small memory, running-time, and training-data requirements. Our work does not only result in a new robot navigation architecture but also a novel application area for POMDPs.

We proceed as follows: Section 3.1 contrasts our POMDP-based navigation architecture with more traditional navigation approaches. Section 3.2 discusses POMDPs and methods for pose estimation, planning, and learning in the abstract. We then apply and extend these models and methods in Section 3.3. In particular, Section 3.3.3 discusses planning and acting for the office-navigation task, and Section 3.3.4 discusses learning the corridor lengths and fine-tuning the actuator and sensor models. These two sections also contain experimental results. Finally, Section 3.4 lists related work, Section 3.5 describes possible extensions of the POMDP-based navigation architecture, and Section 3.6 summarizes our conclusions.

3.1 Traditional Approaches

How to get a robot from its current pose to a given goal pose is a central problem in robotics that has, of course, been studied before [Borenstein *et al.*, 1996]. Two common approaches for solving this problem are metric-based and landmark-based navigation.

Metric-based navigation relies on metric maps of the environment, resulting in navigation plans such as: move forward ten meters, turn right ninety degrees, move forward another ten meters, and stop. Metric-based navigation approaches take advantage of *motion reports* (information about the motion of the robot), such as the translation and rotation derived from the odometer (wheel encoders). They are, however, vulnerable to inaccuracies in both the map making (all distances have to be known precisely) and dead-reckoning abilities of the robot. Such approaches are often used where the robot has good absolute pose estimates, such as for outdoor robots using the Global Positioning System (GPS).

Landmark-based navigation, on the other hand, relies on topological maps whose nodes correspond to *landmarks* (locally distinctive places), such as junctions between corridors, doorways, or foyers. Map edges indicate how the landmarks connect and how the robot should navigate between them. A typical landmark-based navigation plan might be to move forward to the second corridor on the right, turn into that corridor, move to its end, and stop. Landmark-based approaches are attractive because they take advantage of *sensor reports* (information about the sensed features of the environment), such as data from ultrasonic sensors that indicate whether the robot is in a corridor junction. Thus, they do not depend on geometric accuracy. They suffer, however, from problems of *noisy sensors* (sensors occasionally not detecting landmarks) as well as problems of *sensor aliasing* (sensors not being able to distinguish between similar landmarks, such as different doorways of the same size) [Kuipers and Byun, 1988, Kortenkamp and Weymouth, 1994].

To maximize reliability in office navigation, it makes sense to utilize all information that is available to the robot (that is, both motion and sensor reports). While some landmark-based navigation methods use motion reports, mostly to resolve topological ambiguities, and some metric-based navigation methods use sensor reports to continuously realign the robot with the map [Kuipers and Byun, 1988, Mataric, 1990], the two sources of information are treated differently. We want a method that seamlessly integrates both sources of information, and is amenable to adding new sources of information such as apriori knowledge about which doorways are likely to be open or closed. We also do not want to assume that the distances are known precisely.

Another problem with both metric-based and landmark-based navigation approaches is that they typically represent only a single pose that is believed to be the current pose of the robot. If this pose proves to be incorrect, the robot is lost and has to re-localize itself, an expensive operation. This can be avoided

Figure 3.1: Position Estimation with Gaussians

by maintaining a set of possible poses (Section 2.4.1). While the robot rarely knows exactly where it is, it always has some belief as to what its true pose is, and thus is never completely lost. If the robot can associate probabilities with the poses and explicitly represent the various forms of uncertainty present in the office-navigation task, then it can maintain a probability distribution over all possible poses that represents the belief in its current pose, called the *pose distribution*. It can use Bayes rule to update this pose distribution after each motion and sensor report.

There are two ways of representing the pose distribution. The robot can either discretize the pose distributions or not. Discretizing them allows the robot to represent arbitrary pose distributions but incurs a discretization error. Consequently, there is a tradeoff between the precision and expressiveness of the models. We contend that, for office navigation, the added expressiveness of being able to model arbitrary pose distributions outweighs the loss in precision from discretization, especially since a very fine-grained discretization can be used. Even if the robot uses a coarse-grained discretization (as our robot does), it can use low-level control routines to overcome the discretization problem. For example, our robot uses control routines to keep itself centered in corridors and aligned along the main corridor axis. Similarly, our robot uses vision and a neural network to align itself exactly with doorways when it has reached its destination.

Previously reported navigation methods that maintain pose distributions often use Kalman filters [Kalman, 1960], that model only restricted pose distributions in continuous pose space. Examples include [Smith and Cheeseman, 1986, Smith *et al.*, 1990, Kosaka and Kak, 1992, Leonard *et al.*, 1992, Rencken, 1995]. In the simplest case, the pose distributions are modeled with Gaussians. While Gaussians are efficient to encode and update, they are not ideally suited for office navigation. In particular, due to sensor aliasing, the robot often wants to encode the belief that it might be in one of a number of noncontiguous (but similar looking) locations. Figure 3.1 illustrates this problem. A robot does not know where it is but sees a doorway to its left. Thus, discounting sensor uncertainty, it has to be at either of the two doorways but does not know which one. Figure 3.1 (left) shows the resulting position distribution. If the robot were forced to fit a single (unimodal) Gaussian to this multi-modal probability distribution, however, the Gaussian would end up being between the two doorways, due to symmetry, as depicted in Figure 3.1 (right). Thus, most of the probability mass would be far removed from the doorways – precisely the wrong result. Although Kalman filters can be used to represent more complex probability distributions than Gaussians (such as mixtures of Gaussians, at the cost of having to estimate the number of Gaussians and increasing the complexity), they cannot be used to model arbitrary probability distributions [Cox, 1994]. This problem diminishes their utility for office navigation.

Like Kalman filters, partially observable Markov decision process models (POMDPs) utilize motion and sensor reports to update the pose distribution and seamlessly integrate topological information and uncertain distance information. Different from Kalman filters, however, POMDPs discretize the possible poses, and thus allow the robot to represent arbitrary pose distributions. POMDPs are similar to temporal Bayesian networks [Dean *et al.*, 1990] but do not suffer from the problem that the size of the models grows linearly with the amount of the temporal look-ahead. Thus, their use for planning is not limited to rather small look-aheads.

3.2 POMDPs

This section provides a general, high-level overview of partially observable Markov decision process models (POMDPs) and common methods that operate on them. These models and methods form the basis of our POMDP-based navigation architecture, which is described in Section 3.3.

POMDPs consist of a finite set of states S, a finite set of observations O, and an initial state distribution π (a probability distribution over S), where $\pi(s)$ denotes the probability that the initial state of the POMDP process is s. Each state $s \in S$ has a finite set of actions $A(s)$ that can be executed in s. The POMDP further consists of a transition function p (a function from $S \times A$ to probability distributions over S), where $p(s'|s,a)$ denotes the *transition probability* that the system transitions from state s to state s' when action a is executed, an observation function q (a function from S to probability distributions over O), where $q(o|s)$ denotes the *observation probability* of making observation o in state s, and an immediate reward function r (a function from $S \times A$ to the real numbers), where $r(s,a)$ denotes the finite immediate reward resulting from the execution of action a in state s. The immediate rewards accumulate over time. Notice that the transition probabilities depend only on the current state and the executed action but not, for example, on how the current state was reached. Similarly, the observation probabilities depend only on the current state.

A *POMDP process* is a stream of <state, observation, action, immediate reward> quadruples: The POMDP process is always in exactly one state and makes state transitions at discrete time steps. The initial state of the POMDP process is drawn according to the probabilities $\pi(s)$. Thus, $p(s_1 = s) = \pi(s)$ for $t = 1$. Assume that at time t, the POMDP process is in state $s_t \in S$. Then, an observation $o_t \in O$ is generated according to the probabilities $p(o_t = o) = q(o|s_t)$. Next, a decision maker chooses an action a_t from $A(s_t)$ for execution. This results in an immediate reward $r_t = r(s_t, a_t)$ and the POMDP process changing state. The successor state $s_{t+1} \in S$ is selected according to the probabilities $p(s_{t+1} = s) = p(s|s_t, a_t)$. This process repeats forever.

An *observer* of the POMDP process is someone who knows the specification of the POMDP (as stated above) and observes the actions $a_1 \ldots a_{T-1}$ (short: $a_{1\ldots T-1}$) and observations $o_1 \ldots o_T$ (short: $o_{1\ldots T}$), but not the current states $s_1 \ldots s_T$ (short: $s_{1\ldots T}$) or immediate rewards $r_1 \ldots r_{T-1}$ (short: $r_{1\ldots T-1}$). A *decision maker* is an observer who also determines which actions to execute. Consequently, observers and decision makers usually cannot be sure exactly which state the POMDP process is in. This is the main difference between a POMDP and an MDP (completely observable Markov decision process model). Observers and decision makers of MDPs always know exactly which state the Markov process is in. They are still not able to predict the states that result from action executions, since actions can have nondeterministic effects.

Notice that the immediate rewards accumulate over time but cannot be observed by the observers and decision makers. Thus, the observers and decision makers cannot deduce any information from the rewards about which state the POMDP process is in. This is to cleanly distinguish between observations and rewards. Observations provide information about the current state of the POMDP process, whereas rewards are used to evaluate the behavior of decision makers. Despite this fact, we say that the immediate rewards are "received" by the decision makers.

Properties of POMDPs have been studied extensively in operations research [Monahan, 1982, Lovejoy, 1991, White, 1991]. In artificial intelligence and robotics, POMDPs have been applied to speech recognition [Huang *et al.*, 1990], handwriting recognition [Bunke *et al.*, 1995], and the interpretation of tele-operation commands [Hannaford and Lee, 1991, Yang *et al.*, 1994]. POMDPs have also gained popularity in the artificial intelligence community as a formal model for planning under uncertainty [Koenig, 1991, Cassandra *et al.*, 1994]. Consequently, standard methods are available to solve tasks that are typically encountered by observers and decision makers. In the following, we describe some of these methods.

3.2.1 State Estimation: Determining the Current State

Assume that an observer wants to determine the current state of a POMDP process. This corresponds to estimating where the robot currently is. Observers do not have access to this information, but can maintain a belief in the form of a *state distribution* (a probability distribution α over S). We write $\alpha(s)$ to denote the probability that the current state is s. These values can be computed incrementally with Bayes rule, using the knowledge about which actions have been executed and which observations resulted. To begin, the probability of the initial state of the POMDP process is $\alpha(s) = \pi(s)$. Subsequently, if the current state distribution is

α_{prior}, the state distribution after the execution of action a is α_{post}:

$$\alpha_{post}(s) = \frac{1}{scale} \sum_{s' \in S | a \in A(s')} [p(s|s', a)\alpha_{prior}(s')], \tag{3.1}$$

where $scale$ is a normalization constant that ensures that $\sum_{s \in S} \alpha_{post}(s) = 1$. This normalization is necessary only if action a is not defined in every state because only then $\sum_{s \in S} \sum_{s' \in S | a \in A(s')} [p(s|s', a)\alpha_{prior}(s')] \neq 1$.

Updating the state distribution after an observation is even simpler. If the current state distribution is α_{prior}, then the state distribution after making observation o is α_{post}:

$$\alpha_{post}(s) = \frac{1}{scale} q(o|s)\alpha_{prior}(s), \tag{3.2}$$

where $scale$, again, is a normalization constant that ensures that $\sum_{s \in S} \alpha_{post}(s) = 1$. This normalization is always necessary.

3.2.2 POMDP Planning: Determining which Actions to Execute

Assume that a decision maker wants to select actions so as to maximize the average total reward over an infinite planning horizon, which is $E(\sum_{t=1}^{\infty} [\gamma^{t-1} r_t])$, where $\gamma \in (0, 1]$ is the discount factor. This corresponds to navigating the robot to its destination in ways that minimize its average travel time. The *discount factor* specifies the relative value of an immediate reward received after t action executions compared to the same reward received one action execution earlier. If $\gamma = 1$, the total reward is called *undiscounted* otherwise it is called *discounted*. To simplify mathematics, we assume here that $\gamma < 1$, because this ensures that the average total reward is finite, no matter which actions are chosen (see also Section 4.6.4).

Consider an MDP (completely observable Markov decision process model). A fundamental result of operations research is that, in this case, there always exists a mapping from states to actions (also known as "stationary, deterministic policy," short: *policy*) so that the decision maker maximizes the average total reward by always executing the action that the policy assigns to the current state of the Markov process, independent of the start state of the Markov process. Such an optimal policy can be determined by solving the following system of $|S|$ equations for the variables $v(s)$, that is known as *Bellman's Equation* [Bellman, 1957]. Its solution is finite and unique if $\gamma < 1$:

$$v(s) = \max_{a \in A(s)} [r(s, a) + \gamma \sum_{s' \in S} [p(s'|s, a)v(s')]] \qquad \text{for all } s \in S. \tag{3.3}$$

$v(s)$ is the average total reward if the Markov process starts in state s and the decision maker acts optimally. The optimal action to execute in state s is $a(s) =$ one-of $\arg\max_{a \in A(s)} [r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a)v(s')]$. The system of equations can be solved in polynomial time using dynamic programming methods [Littman *et al.*, 1995b] such as linear programming. Other popular dynamic programming techniques are value-iteration [Bellman, 1957], policy-iteration [Howard, 1964], and Q-learning (Section 2.5.6). As an example, we describe *value-iteration*:

1. Set $v_1(s) := 0$ for all $s \in S$. Set $t := 1$.
2. Set $v_{t+1}(s) := \max_{a \in A(s)} [r(s, a) + \gamma \sum_{s' \in S} [p(s'|s, a)v_t(s')]]$ for all $s \in S$. Set $t := t + 1$.
3. Go to 2.

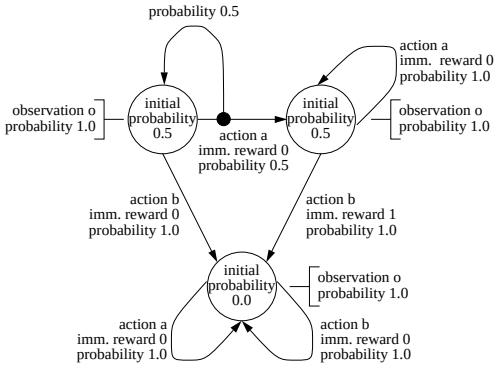


Figure 3.2: A POMDP Planning Problem

Here we leave the termination criterion unspecified. Then, for all $s \in S$, $v(s) = \lim_{t\to\infty} v_t(s)$. See [Bertsekas, 1987] for a good review of MDPs.

The POMDP planning problem can be transformed into a planning problem for an MDP. First, since the decision maker can never be sure which state the POMDP process is in, the set of executable actions A must be the same for every state s, that is, $A(s) = A$ for all $s \in S$. The corresponding MDP can then be constructed as follows: Its states are the state distributions α (*belief states*), with the initial state being α_o with probability $\sum_{s\in S}[q(o|s)\pi(s)]$ for all $o \in O$, where $\alpha_o(s) = q(o|s)\pi(s)/\sum_{s\in S}[q(o|s)\pi(s)]$ for all $s \in S$. Its actions are the same as the actions of the POMDP. The execution of action a in state α results in an immediate reward of $\sum_{s\in S}\alpha(s)r(s,a)$ and the Markov process changing state. There are at most $|O|$ possible successor states, one for each $o \in O$. The successor state α'_o is characterized by:

$$\alpha'_o(s) = \frac{q(o|s)\sum_{s'\in S}[p(s|s',a)\alpha(s')]}{\sum_{s\in S}[q(o|s)\sum_{s'\in S}[p(s|s',a)\alpha(s')]]} \qquad \text{for all } s \in S.$$

The transition probabilities are:

$$p(\alpha'_o|\alpha,a) = \sum_{s\in S}[q(o|s)\sum_{s'\in S}[p(s|s',a)\alpha(s')]].$$

Any policy that is optimal for this MDP is also optimal for the POMDP. This means that for POMDPs, there always exists a mapping from state distributions to actions (also known as stationary, deterministic POMDP policy, short: *POMDP policy*) that maximizes the average total reward (under reasonable assumptions) [Sondik, 1978].[1] The state distribution summarizes everything known about the current state – it is a sufficient statistic for the decision maker. In other words, future decisions cannot be improved by knowing how the current state distribution was reached or how much reward has already been received. This is called the *Markov property*.

The POMDP policy can be precomputed. During action selection, the decision maker only needs to calculate the current state distribution α (as shown in Section 3.2.1) and can then look up which action to execute. Unfortunately, the number of belief states α is infinite (even uncountable). This has two consequences. First,

[1] In general, this is not true even if the total reward is guaranteed to be finite. Consider, for example, the POMDP shown in Figure 3.2. The total reward of any POMDP process never exceeds one for this POMDP. However, the average total undiscounted reward of any action sequence can be increased by preceding its execution with the execution of either action a or b. Thus, there does not exist an optimal POMDP policy although it is possible to get arbitrarily close to the maximal total reward.

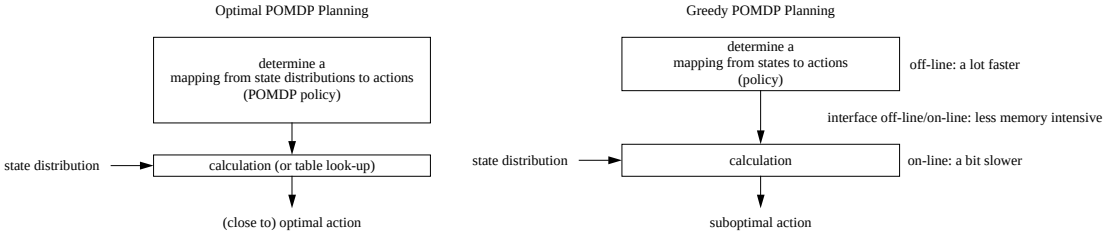


Figure 3.3: POMDP Planning and Acting: Comparing Optimal and Greedy Methods

the optimal policy cannot be stored in a table. In fact, most likely it is not possible to have an efficient on-line implementation of an optimal policy, even if an arbitrary amount of precomputation is allowed [Papadimitriou and Tsitsiklis, 1987]. Second, the MDP is infinite, and an optimal policy cannot be found efficiently for the MDP. In fact, the POMDP planning problem is PSPACE-complete in general [Papadimitriou and Tsitsiklis, 1987]. However, there are POMDP planning methods that trade off solution quality for speed although they usually do not provide quality guarantees [Littman *et al.*, 1995a, Boutilier and Poole, 1996, Wiering and Schmidhuber, 1996, Cassandra, 1997, Hauskrecht, 1997]. The SPOVA-RL method [Parr and Russell, 1995], for example, can determine approximate policies for POMDPs with about a hundred states in a reasonable amount of time by approximating the values $v(\alpha)$ with a simple function $f(\alpha)$. We anticipate further performance improvements since POMDP planning methods are the object of current research [Littman, 1996] and researchers are starting to investigate, for example, how to exploit the restricted structure of some POMDPs [Boutilier *et al.*, 1995a].

We describe here three *greedy POMDP planning methods* that can find policies for large POMDPs fast, but still yield reasonable office-navigation behavior. These three methods share the property that they pretend that the POMDP is completely observable and, under this assumption, use Formula (3.3) to determine an optimal policy. Then, they transform this policy into a POMDP policy (Figure 3.3). We call this transformation *completing* the policy because the mapping a from states to actions is completed to a mapping from state *distributions* to actions. Given the current state distribution α, the methods greedily (that is, without search) complete the policy as follows:

- The **"Most Likely State" Strategy** [Nourbakhsh *et al.*, 1995] executes the action that is assigned to the most likely state, that is, the action $a(\text{one-of arg max}_{s \in S}\, \alpha(s))$.
- The **"Voting" Strategy** [Simmons and Koenig, 1995] executes the action that accumulates the highest probability mass if each state votes for its action with a strength that corresponds to its probability, that is, the action one-of $\arg\max_{a \in A} \sum_{s \in S | a(s)=a} \alpha(s)$.
- The **"Completely Observable after the First Step" Strategy** [Chrisman, 1992, Tenenberg *et al.*, 1992] executes the action that would be optimal if the POMDP became completely observable after the action execution, that is, the action one-of $\arg\max_{a \in A} \sum_{s \in S} [\alpha(s)(r(s,a) + \gamma \sum_{s' \in S} [p(s'|s,a)v(s')])]$. This method allows one, for example, to choose the second best action if all states disagree on the best action but agree on the second best action.

How well each of these greedy POMDP planning methods performs depends on properties of the POMDP. Therefore, given an application, they should all be tried to see which one performs best.

3.2.3 POMDP Learning: Determining the POMDP from Observations

Assume that an observer wants to determine the POMDP that maximizes the probability of generating the observations made for a given sequence of actions, that is, $p(o_{1\ldots T}|a_{1\ldots T-1})$. This POMDP is the one that

best fits the data. In our application, this corresponds to fine-tuning the navigation model from experience, including the actuator, sensor, and distance models. While there is no known technique for efficiently finding a POMDP that best fits the data, there exist efficient methods that approximate the optimal POMDP. The *Baum-Welch method* [Rabiner, 1986] is one such method. This iterative EM (expectation-maximization) method for learning POMDPs from observations does not require control of the POMDP process, and thus can be used by an observer to learn the POMDP. It overcomes the problem that observers can never be sure about the current state of the POMDP process, because they cannot observe the current state directly and are not allowed to execute actions to reduce their uncertainty. Given an initial POMDP and an *execution trace* (a sequence of actions and observations), the Baum-Welch method constructs a POMDP that better fits the trace, in the sense that the generated POMDP has a larger probability $p(o_{1...T}|a_{1...T-1})$ of generating (or, synonymously, explaining) the observations for the given sequence of actions. This probability is also called the *fit*. By repeating this procedure with the same execution trace and the improved POMDP, we get a hill-climbing method that eventually converges to a POMDP which locally, but not necessarily globally, best fits the execution trace. The Baum-Welch method could be terminated when the probabilities of the POMDP have almost converged, for example, when the sum of the changes of all transition and observation probabilities or the largest change of any probability (the *span semi-norm*) drops below a given threshold. This termination criterion, however, can lead to *over-fitting*, that is, adapting to the noise in the execution trace, which would result in a poor generalization performance. Standard machine learning methods can be used to prevent over-fitting. One could, for example, split the execution trace into a training trace and a test trace. The test trace would be used for cross-validation, stopping the Baum-Welch method when the fit of the generated POMDPs starts to decrease on the test trace. This will usually happen before the probabilities have converged.

The Baum-Welch method operates as follows: It first uses the initial POMDP and all information contained in the training trace to calculate, for every point in time, a state distribution that represents the belief that the POMDP process was in a certain state at a certain point in time, using Bayes rule. It then estimates an improved POMDP from these probability distributions, using a maximum-likelihood method. The running time of such an iteration is linear in the product of the length of the training trace and the size of the POMDP. In the following, we describe a slight generalization of the Baum-Welch method that takes into account that the decision maker can choose from several actions in every state and not every action has to be defined in every state. In this case, the fit is the probability of generating the observations made for the executed sequence of actions times the probability that the sequence of actions can be executed, that is, $p(o_{1...T}|a_{1...T-1})p(\text{can execute } a_{1...T-1})$.[2] The Baum-Welch method estimates the improved POMDP in three steps:

First Step: It uses a dynamic programming method (the *forward-backward method*) [Devijver, 1985] that applies Bayes rule repeatedly. The forward phase calculates scaling factors $scale_t = p(o_t|o_{1...t-1}, a_{1...t-1})p(\text{can execute } a_t|o_{1...t-1}, a_{1...t-2})$ and alpha values $\alpha_t(s) = p(s_t = s|o_{1...t}, a_{1...t-1}) = (p(s_t = s, o_{1...t}|a_{1...t-1})p(\text{can execute } a_{1...t-1}))/\prod_{t'=1...t} scale_{t'}$ for all $s \in S$ and $t = 1 \ldots T$.

A1. Set $scale_1 := \sum_{s \in S}[q(o_1|s)\pi(s)]$.

A2. Set $\alpha_1(s) := q(o_1|s)\pi(s)/scale_1$ for all $s \in S$.

A3. For $t := 1$ to $T - 1$

(a) Set $scale_{t+1} := \sum_{s \in S}[q(o_{t+1}|s)\sum_{s' \in S|a_t \in A(s')}[p(s|s', a_t)\alpha_t(s')]]$.

(b) Set $\alpha_{t+1}(s) := (q(o_{t+1}|s)\sum_{s' \in S|a_t \in A(s')}[p(s|s', a_t)\alpha_t(s')])/scale_{t+1}$ for all $s \in S$.

The backward phase of the forward-backward method calculates beta values $\beta_t(s) = (p(o_{t+1...T}|s_t = s, a_{t...T-1})p(\text{can execute } a_{t...T-1}|s_t = s))/(p(o_{t...T}|o_{1...t-1}, a_{1...T-1})p(\text{can execute } a_{t-1...T-1}|o_{1...t-1}, a_{1...t-2})) = (p(o_{t+1...T}|s_t =$

[2] POMDPs in which not every action is defined in every state can be transformed into POMDPs in which every action is defined in every state. This is done by adding the undefined actions to the states. All of the added actions lead with probability one to a new state in which only one observation can be made. This observation is different from all existing observations. The Baum-Welch method can then be applied to either the original or the transformed POMDP, with identical results.

$s, a_{t\ldots T-1})p(\text{can execute} a_{t\ldots T-1}|s_t = s))/\prod_{t'=t\ldots T} scale_{t'}$ for all $s \in S$ and $t = 1\ldots T$. Unlike the other state distributions, $\sum_{s\in S} \beta_t(s)$ is not guaranteed to equal one:

A4. Set $\beta_T(s) := 1/scale_T$ for all $s \in S$.

A5. For $t := T-1$ downto 1

(a) Set $\beta_t(s) := \begin{cases} \sum_{s'\in S}[p(s'|s,a_t)q(o_{t+1}|s')\beta_{t+1}(s')]/scale_t & \text{if } a_t \in A(s) \\ 0 & \text{otherwise} \end{cases}$ for all $s \in S$.

Second Step: The Baum-Welch method calculates the gamma values $\gamma_t(s,s') = p(s_t = s, s_{t+1} = s'|o_{1\ldots T}, a_{1\ldots T-1})$ for all $t = 1\ldots T-1$ and $s, s' \in S$ with $a_t \in A(s)$, and $\gamma_t(s) = p(s_t = s|o_{1\ldots T}, a_{1\ldots T-1})$ for all $t = 1\ldots T$ and $s \in S$.

A6. Set $\gamma_t(s,s') := \alpha_t(s)p(s'|s,a_t)q(o_{t+1}|s')\beta_{t+1}(s')$ for all $t = 1\ldots T-1$ and $s, s' \in S$ with $a_t \in A(s)$.

A7. Set $\gamma_t(s) := scale_t\alpha_t(s)\beta_t(s)$ for all $t = 1\ldots T$ and $s \in S$.

The alpha values α_t are the same as the state distributions calculated in Section 3.2.1. They take all the information in a prefix of the training trace into account and are calculated forward from $t = 1$. They can be calculated on the fly, because they are conditioned only on the part of the training trace that is available at the current time t. The beta values β_t, in contrast, take all the information in a suffix of the training trace into account and are calculated backward from $t = T$. The gamma values γ_t combine the information from the alpha and beta values and take all the information of the training trace into account. Thus, the gamma values $\gamma_t(s)$ are more precise estimates of the same state distribution as the alpha values $\alpha_t(s)$, because they also utilize information (represented by the beta values) that became available only after time t. For example, going forward three meters and seeing a wall ahead at time t provides strong evidence that at time $t-3$ the robot was three meters away from the end of the corridor. The scaling factors are used to prevent the numerators of the alpha and beta values from underflowing. They are also convenient for calculating the fit, since $p(o_{1\ldots T}|a_{1\ldots T-1})p(\text{can execute } a_{1\ldots T-1}) = \prod_{t=1}^{T} scale_t$.

Third Step: The Baum-Welch method uses the following frequency-counting re-estimation formulae to calculate the improved initial state distribution, transition probabilities, and observation probabilities. The over-lined symbols represent the probabilities that constitute the improved POMDP:

A8. Set $\bar{\pi}(s) := \gamma(s)$.

A9. Set $\bar{p}(s|s,a) := \sum_{t=1\ldots T-1|a_t=a} \gamma_t(s,s')/\sum_{t=1\ldots T-1|a_t=a} \gamma_t(s)$ for all $s, s' \in S$ and $a \in A(s)$.

A10. Set $\bar{q}(o|s) := \sum_{t=1\ldots T|o_t=o} \gamma_t(s)/\sum_{t=1\ldots T} \gamma_t(s)$ for all $s \in S$ and all $o \in O$.

Figure 3.4 summarizes the calculations performed by the Baum-Welch method.

The Baum-Welch method can improve the transition and observation probabilities of a POMDP only for those probabilities that are not either zero or one. This means that transitions that were impossible (or certain) in the original POMDP remain so in the improved POMDP. Thus, the Baum-Welch method cannot change the structure of a POMDP, namely, its number of states and how they connect. This property has the following implications: First, if $p(s'|s,a) = 0$, then the values $\gamma_t(s,s')$ need not be computed for $s, s' \in S$ and $t = 1\ldots T-1$. This saves computation time. Second, the transition and observation probabilities of the initial POMDP have to be different from zero or one if one wants the Baum-Welch method to be able to adjust them. (Probabilities close to zero or one are acceptable, though.) Third, transition and observation probabilities originally different from zero or one can get indistinguishably close to these extremes during learning (a precision problem). After that the Baum-Welch method cannot change them any longer. To avoid this problem one often keeps the probabilities suggested by the re-estimation formulae away from the extremes by keeping them within the interval $[\epsilon, 1-\epsilon]$ for a small $\epsilon > 0$, provided that the initial probabilities were different from zero or one.

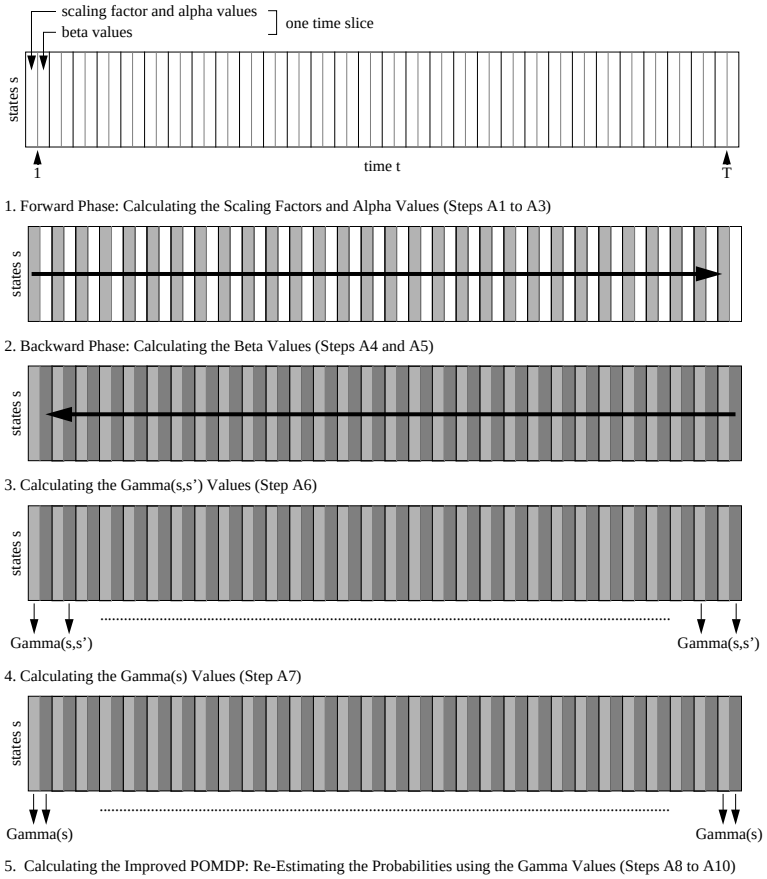


Figure 3.4: An Overview of the Baum-Welch Method

3.2.4 Most Likely Path: Determining the State Sequence from Observations

Assume that an observer wants to determine the most likely sequence of states that the POMDP process visited. This corresponds to determining the path that the robot most likely took to get to its destination. While the techniques from Section 3.2.3 can determine the most likely state of the POMDP process at each point in time, merely connecting these states might not result in a continuous path. The *Viterbi method* is a dynamic programming method that computes the most likely path efficiently [Viterbi, 1967]. Its first three steps are the same as Steps A1 to A3 (from page 90), except that summations on Lines A3(a) and A3(b) are replaced by maximizations:

B1. Set $scale'_1 := \sum_{s \in S}[q(o_1|s)\pi(s)]$.

B2. Set $\alpha'_1(s) := q(o_1|s)\pi(s)/scale'_1$ for all $s \in S$.

B3. For $t := 1$ to $T - 1$

(a) Set $scale'_{t+1} := \sum_{s \in S}[q(o_{t+1}|s) \max_{s' \in S | a_t \in A(s')}[p(s|s', a_t)\alpha'_t(s')]]$.

(b) Set $\alpha'_{t+1}(s) := [q(o_{t+1}|s) \max_{s' \in S | a_t \in A(s')}[p(s|s', a_t)\alpha'_t(s')]]/scale'_{t+1}$ for all $s \in S$.

B4. Set $\bar{s}_T := \max_{s \in S} \alpha'_T(s)$.

B5. For $t := T - 1$ to 1

(a) Set $\bar{s}_t :=$ one-of $\arg\max_{s' \in S | a_t \in A(s')}[p(\bar{s}_{t+1}|s', a_t)\alpha'_t(s')]$.

$\bar{s}_t$ is the state at time t that is part of the most likely state sequence.

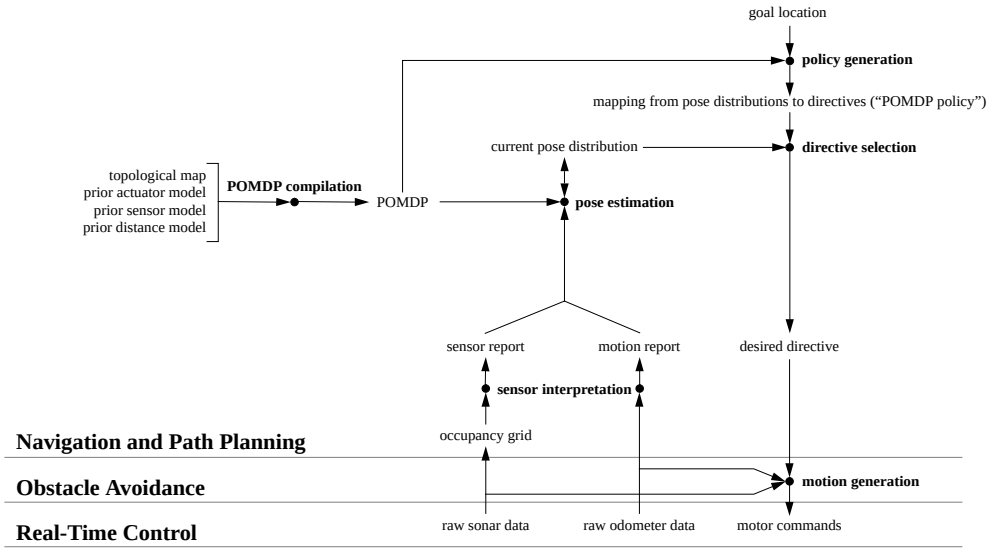


Figure 3.5: The POMDP-Based Navigation Architecture

3.3 The POMDP-Based Navigation Architecture

This section describes the architecture of an office-navigation system that applies the general models and methods presented in Section 3.2 to the office-navigation task. The POMDP-based navigation architecture implements the navigation layer of Xavier's autonomous mobile-robot system for office delivery. The role of the navigation layer is to send directives to the obstacle avoidance layer that move the robot towards its goal, where a *directive* is either a change in desired heading or a "stop" command. The obstacle avoidance layer then heads in that direction while using sensor data to avoid obstacles.

The POMDP-based navigation architecture consists of several components (Figure 3.5): The *sensor-interpretation* component converts the continual motion of the robot into discrete *motion reports* (heading changes and distance traveled) and produces *sensor reports* of high-level features observed in the environment, such as walls and openings of various sizes, observed in front of the robot and to its immediate left and right. The *pose-estimation* component uses the motion and sensor reports to maintain a pose distribution by updating the state distribution of the POMDP.

The *policy-generation* component generates a POMDP policy that maps pose distributions to directives. Whenever the pose-estimation component generates a new pose distribution, the *directive-selection* component uses the new pose distribution to index the pre-calculated POMDP policy and sends the resulting *directive* (heading) to the obstacle avoidance layer, which then generates the robot motion commands. Thus, directive selection is fast and very reactive to the motion and sensor reports. The directives are also fed back to the sensor-interpretation component (not shown in Figure 3.5), since the interpretation of the raw odometer and ultrasonic sensor data is heading dependent.

The initial POMDP is generated once for each environment by the *POMDP-compilation* component. It uses a topological map and initial approximate actuator, sensor, and distance models. In Section 3.3.4, we will also add an unsupervised, passive *model-learning* component to the POMDP-based navigation architecture. As the robot gains more experience while it performs its office-navigation tasks, the model-learning component uses extensions of the Baum-Welch method to automatically adapt the initial POMDP to the environment of the robot, which improves the accuracy of the actuator and sensor models and reduces the uncertainty about the distances in the environment. This increases the precision of the pose-estimation component, which ultimately improves the office-navigation performance of the robot.

In the following, we first describe the interface between the POMDP-based navigation architecture and the obstacle avoidance layer, including the sensor-interpretation component. Then, we describe the POMDP and the POMDP-compilation component in detail. Finally, we explain how the POMDP is used by the pose-estimation, policy-generation, and directive-selection components and report on experiments performed with the POMDP-based navigation architecture.

3.3.1 Interface to the Obstacle Avoidance Layer

An advantage of a layered robot system is that the navigation layer is insulated from many details of the actuators, sensors, and the environment (such as stationary and moving obstacles). The navigation layer itself provides further abstractions in the form of discrete motion and sensor reports that further insulate the POMDP from details of the robot control.

These abstractions have the advantage of enabling us to discretize the possible poses of the robot into a finite, relatively small number of states, and keep the number of possible motion and sensor reports small. In particular, we discretize the location of the robot with a precision of one meter, and discretize its orientation into the four compass directions. We have used other discretization granularities as well, but found this one to be sufficient in our building. While more fine-grained discretizations yield more precise models, they also result in larger POMDPs and thus in larger memory requirements and more time consuming computations.

3.3.1.1 Directives

The task of the navigation layer is to supply a series of changes in desired headings, which we call *directives*, to the obstacle avoidance layer to make the robot reach its goal pose. The main role of the obstacle avoidance layer is to make the robot head in the given direction while avoiding obstacles.

The directives issued by the navigation layer are: change the desired heading by ninety degrees (“turn right”), minus ninety degrees (“turn left”), zero degrees (“go forward”), and stop. Directives are cumulative, so that, for example, two successive “right turn” directives result in a smooth 180 degree turn. If the robot is already moving, a new directive does not cause it to stop and turn, but merely to change the desired heading, which the obstacle avoidance layer then tries to follow. This results in the robot making smooth turns in corridor junctions.

The robot uses Reid Simmons’ Curvature Velocity Method [Simmons, 1996] for local obstacle avoidance. The Curvature Velocity Method formulates the problem as one of constrained optimization in velocity space. Constraints are placed on the translational and rotational velocities of the robot that stem from physical limitations (velocities and accelerations) and the environment (the configuration of obstacles). The robot chooses velocity commands that satisfy all the constraints and maximize an objective function that trades off speed, safety, and goal directedness. These commands are then sent to the motors and, if necessary, change the actual heading of the robot.

The obstacle avoidance layer also keeps the robot centered along the main corridor axis by correcting for angular dead-reckoning error. It tries to fit lines to the ultrasonic sensor data and, if the fit is good enough, uses the angle between the line (for example, a wall) and the desired heading to compensate for angular drift.

3.3.1.2 Sensor Interpretation: Motion and Sensor Reports

The sensor-interpretation component asynchronously generates discrete *motion and sensor reports* that are abstractions of the continuous stream of data provided by the sensors on-board the robot (wheel encoders for the motion reports and ultrasonic sensors for the sensor reports). An *execution trace* is the chronological sequence of motion and sensor reports, preceded by the initial pose distribution (Figure 3.6). One can specify a uniform distribution over all poses (corresponding to no initial knowledge), the exact initial pose

```
no initial knowledge given
left sensor report:   saw wall (1.0)
right sensor report:  saw small_opening (1.0)
front sensor report:  saw unknown (1.0)
motion report:        went forward 1 meter
left sensor report:   saw unknown (1.0)
right sensor report:  saw wall (1.0)
front sensor report:  saw unknown (1.0)
motion report:        went forward 1 meter
left sensor report:   saw wall (1.0)
...
```

Figure 3.6: An Execution Trace

Sensor	Features that the Sensor Reports on
front	`unknown`, `wall`
left	`unknown`, `wall`, `small_opening`, `medium_opening`, `large_opening`
right	`unknown`, `wall`, `small_opening`, `medium_opening`, `large_opening`

Figure 3.7: The (Virtual) Sensors and their Features

(corresponding to complete initial knowledge), or a noisy version of the initial pose, such as, "the robot is approximately two meters West of corridor junction X facing North;" the initial pose distribution is then normally distributed around this pose with a specified variance. The execution trace is recorded automatically as the robot moves around. The pose-estimation component uses Bayes rule after each motion and sensor report to update the pose distribution, starting with the initial pose distribution.

Motion reports are derived by discretizing the smooth motion of the robot. The sensor-interpretation component periodically receives reports from the odometer on-board the robot. It combines this information with the robot's commanded heading to produce a virtual odometer that keeps track of the distance traveled along, and orthogonal to, that heading. This ensures that the distance the robot travels in avoiding obstacles is not counted in determining how far it has traveled along a corridor. The sensor-interpretation component integrates the odometer reports over time and generates a "forward" motion report after each meter of cumulative travel in the desired heading. Similarly, the sensor-interpretation component reports when the heading of the robot has changed relative to the desired heading, and reports this in units of ninety degree turns. This assumes that corridors are straight and perpendicular to each other. The sensor-interpretation component also generates motion reports when the robot moves orthogonally to the desired heading. Such slide motions often occur in open spaces (such as foyers) where the robot can move a significant distance orthogonally to the desired heading during obstacle avoidance.

Sensor reports are generated for three virtual (that is, high-level) sensors that report features in the immediate front, to the immediate left, and to the immediate right of the robot. We do not model a virtual sensor that reports features immediately behind the robot, partly because the POMDP-based navigation architecture can usually infer this information from past sensor reports and partly because this enables researchers to follow the robot undetected. However, new virtual sensors, such as a "back" sensor, or a "front" sensor based on vision instead of ultrasonic sensors, can easily be added by specifying their sensor probabilities (Section 3.3.2.2).

Figure 3.7 lists the virtual sensors that we currently use, together with the features that they report on. The features are currently predetermined although navigation performance can be improved by optimizing the set of features [Thrun, 1996]. The sensor-interpretation component derives them from the ultrasonic sensor data by using a small local *occupancy grid* (obstacle map) [Elfes, 1989] in the coordinates of the robot that is centered around the robot (Figure 3.8). The occupancy grid combines the raw data from all ultrasonic sensors and integrates them over the recent past. The sensor-interpretation component then processes the occupancy grid by projecting a sequence of rays perpendicular to the robot heading until they intersect an occupied grid cell. If the end points of the rays can be fit to a line with a small chi-squared statistic, a wall has been detected

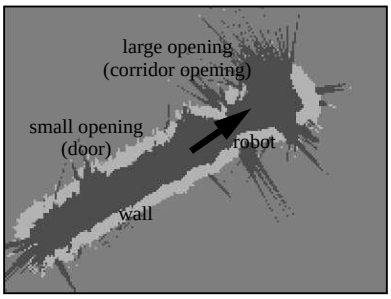


Figure 3.8: An Occupancy Grid with Features of the Environment

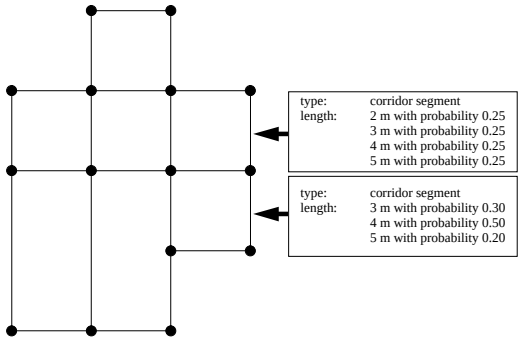


Figure 3.9: Topological Map Augmented with Distance Information

with high probability. Similarly, a contiguous sequence of long rays indicates an opening.

The occupancy grid filters some noise out of the ultrasonic sensor data by integrating them over time and over different ultrasonic sensors. This is important since raw data from ultrasonic sensors can be very noisy due to specular reflections. The virtual sensor reports also approximate the Markov property better than the ultrasonic sensor data. Two ultrasonic sensors, for example, that point in approximately the same direction produce highly correlated data. By bundling their raw data into one virtual sensor we attempt to make sensor reports more independent of each other.

3.3.2 POMDP Compilation

A POMDP is defined by its states and its initial state distribution, its observations and observation probabilities, its actions and transition probabilities, and its immediate rewards. In our POMDP-based navigation architecture, the states of the POMDP encode the pose of the robot. The initial state distribution encodes the available knowledge about the initial pose and thus coincides with the initial pose distribution. The observations are probability distributions over the features, one for each virtual sensor, and the observation probabilities encode the sensor models. The actions are the motion reports (for pose estimation) and directives (for policy generation). The transition probabilities encode the actuator model and the map, including the distance uncertainty. Finally, the immediate rewards (here, costs) express the average execution times of the actions, and the objective is to determine a POMDP policy that minimizes the average total cost. This minimizes the average travel time of the robot to the given goal pose (modulo discounting). A more general preference model will be discussed in Chapter 4.

The map information is initially encoded as a topological map whose nodes represent junctions between corridors, doorways, or foyers. The nodes are connected by undirected edges that indicate how the landmarks connect. The edges are augmented with uncertain distance information in the form of probability distributions

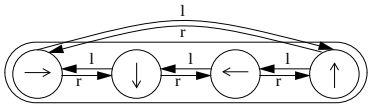


Figure 3.10: A Group of Four States Modeling One Location

over the possible edge lengths (Figure 3.9). We assume that the topological map of the environment can easily be obtained. Approximate distances can then be obtained from either rough measurements, general knowledge, or the model-learning component. The remainder of this section describes how the robot navigation problem maps to the states, observations, and actions of the POMDP and then how the topological map is used to create the POMDP automatically.

3.3.2.1 States and the Initial State Distribution

Since we discretize the orientation of the robot into the four compass directions, a group of four states together with "left turn" (l) and "right turn" (r) actions, is necessary to fully represent the possible robot poses at each spatial location (Figure 3.10). Since we discretize space with a resolution of one meter, each group of four nodes represents one square meter of free space. The initial state distribution of the POMDP process then encodes the possibly uncertain knowledge of the initial robot pose and thus coincides with the pose distribution. We do not treat it as part of the POMDP, but specify it as part of the execution trace (Section 3.3.1.2), since different office-navigation tasks in the same environment differ in their initial state distributions but share all other information.

3.3.2.2 Observations and Observation Probabilities

We denote the set of sensors by I and the set of features that sensor $i \in I$ reports on by $F(i)$. Sensor i reports a probability $r_i(f)$ for each feature $f \in F(i)$. This simplifies the interpretation of the raw sensor data, because it allows more than one interpretation as to which feature is present. By weighting the presence of features, the sensor-interpretation component is not forced to report the most likely interpretation only, but can report the other interpretations as well, with a lower degree of belief.

The sensor model is specified by the sensor probabilities $q_i(f|s)$ for all $i \in I$, $f \in F(i)$, and $s \in S$, which encode the sensor uncertainty. $q_i(f|s)$ is the probability with which sensor i reports feature f in state s. We do not represent the observations explicitly, but calculate only their probability: If sensor i reports feature f with probability $r_i(f)$, then we model this as an observation o with the observation probabilities $q(o|s) = \prod_{i \in I} \sum_{f \in F(i)} [q_i(f|s) r_i(f)]$ for all $s \in S$. This formula assumes that the sensor reports of different sensors are independent, given the state. It is only an approximate solution if the sensors report true probability distributions over their features, especially when the sensor probabilities have to be learned, but it seems to work well in practice.

A sensor that has not issued a report is assumed to have reported the feature unknown with probability one. This can happen because, for example, the sensor-interpretation component has made no determination which feature is present. It can also happen because the sensor-interpretation and pose-estimation components operate asynchronously and the sensor-interpretation component was not able to issue a report in time. The probabilities $q_i(\mathsf{unknown}|s)$ are chosen so that the state distribution remains unaffected in this case, that is, $q_i(\mathsf{unknown}|s) = q_i(\mathsf{unknown}|s')$ for all $i \in I$ and $s, s' \in S$. Learning can change these probabilities later because even a sensor report unknown can carry information (Section 3.3.4.4.1).

To simplify the specification of the sensor model, rather than characterizing $q_i(f|s)$ for each state $s \in S$ individually, we characterize it for a partition C of the state space S, where the classes (state sets) $c \in C$ have to be exhaustive and mutually exclusive. Figure 3.11 shows the classes that we use. New classes can

Class c	Explanation
`wall`	a wall about one meter away
`near-wall`	a wall about two meters away
`open`	a wall three or more meters away (for example, a corridor opening)
`closed-door`	a closed door
`open-door`	an open door
`door`	a door with unknown door state (open or closed)

Figure 3.11: Classes of States

easily be added, for example, for walls adjacent to corridor openings so that a sensor could pick up some of the corridor opening. An example is the wall marked X in Figure 3.13. The sensor model is then specified by probabilities $q_i(f|c)$, that is, probabilities that a sensor reports a given feature when the robot is in that particular class of states. We then define $q_i(f|s) := q_i(f|c(s))$ for all $s \in S$, where $c(s)$ is the unique class with $s \in c$. For example, the "left" sensor is partially characterized by:

$$
\begin{aligned}
&q_{\text{"left" sensor}}(\texttt{wall}|\texttt{open}) && = 0.05\\
&q_{\text{"left" sensor}}(\texttt{small\_opening}|\texttt{open}) && = 0.20\\
&q_{\text{"left" sensor}}(\texttt{medium\_opening}|\texttt{open}) && = 0.40\\
&q_{\text{"left" sensor}}(\texttt{large\_opening}|\texttt{open}) && = 0.30\\
&q_{\text{"left" sensor}}(\texttt{unknown}|\texttt{open}) && = 0.05
\end{aligned}
$$

These probabilities indicate that corridor openings to the left are most commonly detected as medium-sized openings, but can often be seen as either large or small openings although they are hardly ever confused for walls. Also, the left sensor usually provides some report when in the vicinity of openings; it hardly ever reports the feature `unknown`.

3.3.2.3 Actions and Transition Probabilities

We first discuss how to model actions in general, then how to encode corridors, and finally how to encode corridor junctions, doorways, rooms, and foyers.

3.3.2.3.1 Modeling Actions The actions of the POMDP encode the motion reports (for pose estimation) and directives (for policy generation). The transition probabilities of actions encode the actuator (and distance) uncertainty, and their immediate costs encode how long it takes to complete them. In general, all actions have their intended effect with high probability. However, there is a small chance that the robot ends up in an unintended pose, such as an unintended orientation (this is not shown in Figure 3.10 and subsequent figures because it makes them hard to comprehend). Exactly which poses these are is determined by the actuator model. For example, the robot might fail to turn, overshoot the turn, or move forward while turning. Likewise, dead-reckoning uncertainty usually results in the robot overestimating its travel distance in corridors (due to wheel slippage). This can be modeled with self transitions, that is, "forward" actions do not change the state with small probability.

There is a trade-off in which possible transitions to model and which ones to leave out. Modeling fewer transitions results in smaller POMDPs and thus in smaller memory requirements and less time consuming computations. One loses predictive power if one models nonexisting transitions since nonexisting transitions make the pose estimates less exact. On the other hand, if one does not model all possible action outcomes, the robot may do something unmodeled and the POMDP will lose track of the pose. In the worst case, the pose estimates can become inconsistent, that is, every possible pose is ruled out. This is undesirable, because the robot can only recover from an inconsistency by re-localizing itself. Therefore, while we found that most actions have fairly deterministic outcomes, we introduce probabilistic outcomes where appropriate. This also benefits the Baum-Welch method used by the model-learning component, because the Baum-Welch method

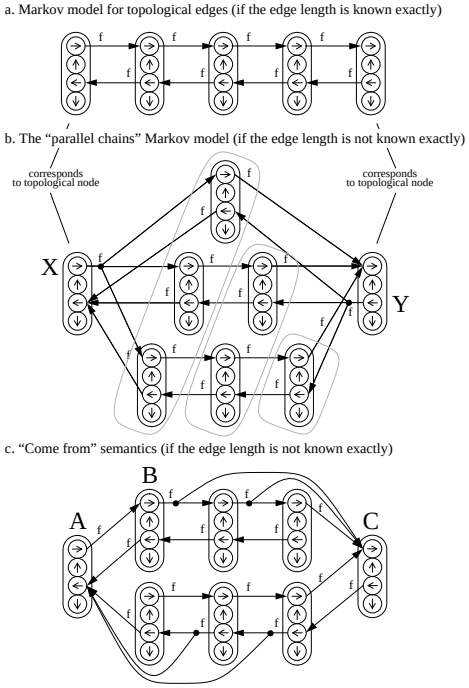


Figure 3.12: Representations of Topological Edges

is not able to change probabilities that are zero but is able to reduce nonzero transition probabilities to values close to zero. This means that it cannot "generate" necessary transitions that were not present initially, but it can "remove" unnecessary transitions. Consequently, when in doubt whether to model a transition, one should include it in the initial POMDP rather than leaving it out.

There is a slight difference between an action that encodes a motion report and the action that encodes the corresponding directive. For the most part, their transition probabilities are identical. A motion report "turned left" and a directive "turn left," for example, both lead with high probability to a state at the same location whose orientation is ninety degrees counterclockwise. Only the semantics of "forward" motion reports and "forward" directives differ slightly. Basically, "forward" actions are not defined in states that face walls when dealing with motion reports but they are defined and result in self-transitions (that is, not changing state) when dealing with directives: If the robot is able to move forward one meter, it is unlikely that it was facing a wall. Thus, for dealing with motion reports, the self-transition probability of "forward" actions are set very low in states that face walls. We actually define "forward" actions in these states to avoid eliminating the true state from the state distribution in the face of slippage and the effects of discretization. On the other hand, for planning purposes, the same self-transition probabilities are set high, since we know that low-level control routines prevent the robot from moving into walls.

3.3.2.3.2 Modeling Corridors The representation of topological edges is a key to our approach. Topological edges correspond, for example, to corridor segments, that is, the part of a corridor between two corridor junctions, doorways, or foyers. We will refer to topological edges as corridors, although corridors are usually composed of several corridor segments and corridor segments are just one example of topological edges. If the true length $l_{true}(c)$ of some corridor c is known, it is simple to model the ability to traverse the corridor with a state chain that has "forward" (f) actions between those states whose orientations are parallel to the main corridor axis (Figure 3.12(a)). Often, however, the length of a corridor will not be known precisely. In this case, one can estimate a probability distribution p_c over the possible lengths $l \in [l_{min}(c), l_{max}(c)]$, where $l_{min}(c)$ and $l_{max}(c)$ are a lower and upper bound, respectively, on $l_{true}(c)$. Then, $p_c(l)$ is the probability that

the length is l, and the corridor can be modeled as a set of parallel state chains that share their first and last states (Figure 3.12(b)). Each chain corresponds to one of the possible lengths $l \in [l_{min}(c), l_{max}(c)]$, and the "forward" actions in the shared states have probabilistic outcomes according to the probabilities $p_c(l)$. These transition probabilities thus model the distance uncertainty. Each "forward" action after that is deterministic (modulo actuator uncertainty). While this *parallel-chain semantics* best captures the actual structure of the environment, it is relatively inefficient, since the number of states is quadratic in $l_{max}(c) - l_{min}(c)$, the difference between the maximum and minimum length to consider for the corridor.

As a compromise between fidelity and efficiency, one can model corridors by collapsing the parallel chains in a way that we call the *come-from semantics* (Figure 3.12(c)). The groups of four states that we collapse into one group are framed in Figure 3.12(b). Each corridor is then represented using two chains, one for each of the corridor directions. In the come-from semantics, the spatial location of a state is known relative to the topological node (for example, corridor junction) from which the robot comes, but its location relative to the end of the chain is uncertain. For example, state B in Figure 3.12(c) is one meter away from A, but is between one and three meters away from C. An alternative representation is the *go-to semantics*, in which the location of a state is specified relative to the topological node towards which the robot is heading, but the distance from the start node is uncertain.

The come-from semantics seems more natural than the go-to semantics, since it models the intuition that one is typically more certain about where one just came from than about one's destination. For example, if the robot turns around at some state s within the corridor and heads back towards the start node, the come-from semantics will correctly predict that the robot will reach that node when it has traveled the same distance it took to reach state s in the first place (modulo actuator uncertainty). That is, even though the exact length of the corridor is uncertain, the robot remembers how long it has traveled in coming from a given topological node. This can be important when the robot misses a corridor opening and overshoots, then realizes that and turns around. In this case, the additional information about how far it went into the corridor helps to locate the corridor opening. However, the come-from semantics also has disadvantages. In particular, it is difficult to integrate sensors that detect features well ahead of the robot, such as sensors based on vision.

When the distance uncertainty is large, the come-from or go-to semantics can save significant memory space over the parallel-chains semantics, since the number of states is only linear in $l_{max}(c)$, the maximum length to consider for the corridor. For example, they need only 80 states (that is, 20 groups of four states) to encode a corridor that is between two and ten meters long, compared to 188 states for the parallel-chains semantics. Since the distance uncertainty in our maps is not that large, we actually switched from the come-from semantics to the parallel-chains semantics in our implementation of the POMDP-based navigation architecture. We did this because the parallel-chains semantics combines the advantages of the come-from and go-to semantics: it models both how far the robot went into a corridor *and* how far it is still away from the next topological node (landmark). Some of our experiments use the come-from semantics and others use the parallel-chains semantics.

3.3.2.3.3 Modeling Corridor Junctions, Doorways, Rooms, and Foyers While we could represent corridor junctions simply with a single group of four states, our experience with the real robot has shown this representation to be inadequate, since the spatial resolution of a state is one meter, but our corridors are two meters wide. To understand why this approach can lead to problems, consider the scenario shown in Figure 3.13: The robot is actually one state away from the center of a T-junction (facing the wall), but due to distance uncertainty still believes to be further away from the corridor junction. The "left" sensor picks up the corridor opening and reports it. This increases the belief that the robot is already in the corridor junction. Now assume that, due to communication delays among distributed processes, the robot continues to move forward and generates a "forward" motion report, which is possible because the corridor junction is two meters wide. According to the model, however, this rules out that the robot was in the corridor junction since the robot cannot move forward in a corridor junction that is one meter wide, except for a small probability of self-transitioning. Consequently, the most likely state jumps back into the corridor, and the resulting state distribution is approximately the same as if the "left" sensor had not issued its report. Thus, the sensor report contributes

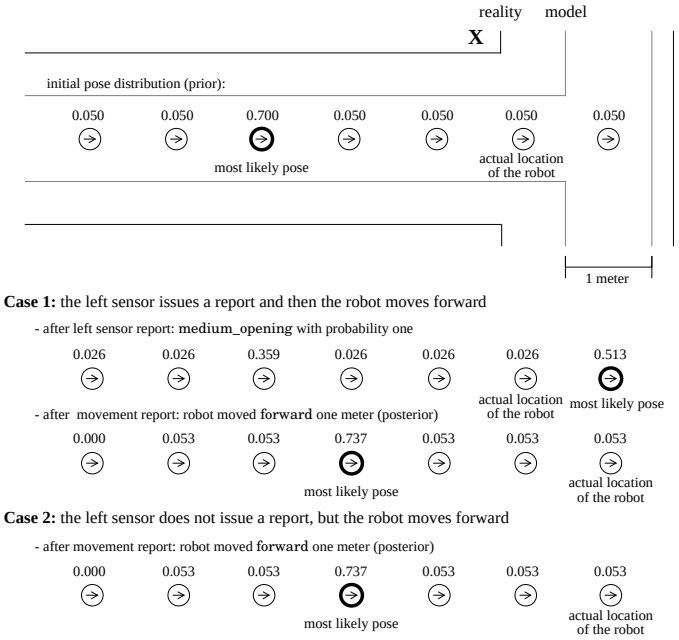


Figure 3.13: The Effect of Pretending that Corridors are One Meter Wide

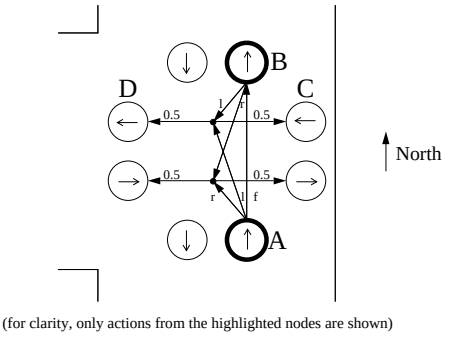


Figure 3.14: The Representation of Corridor Junctions

little to the current state distribution and its information is essentially lost. Figure 3.13 illustrates this effect under the simplifying assumption that there is no distance uncertainty, the actuator model is completely deterministic, and the "left" sensor is characterized by $q_{\text{"left" sensor}}(\texttt{medium\_opening}|\texttt{open}) = 0.40$ and $q_{\text{"left" sensor}}(\texttt{medium\_opening}|\texttt{wall}) = 0.02$. However, the general effect is independent of the sensor model or the initial state distribution.

While one remedy is to represent corridor junctions using four (that is, two by two) groups of four states each, we achieve nearly the same result with four groups of two states each, which both saves memory space and makes the model simpler (Figure 3.14). The basic idea is that turns within a corridor junction are nondeterministic and transition with equal probability to one of the two states of the appropriate orientation in the corridor junction. For example, in entering the corridor junction of Figure 3.14 from the South, the robot would first encounter state A, then state B if it continued to move forward. If it then turned left, it would be facing West, and would transition to either states C or D with equal probability. This model agrees with how the robot actually behaves in corridor junctions. In particular, it corresponds to the fact that it is very difficult to pin down the robot's location exactly while it is turning in the middle of a corridor junction.

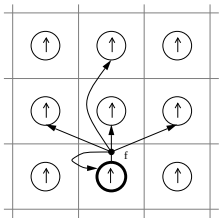


Figure 3.15: The Representation of Two-Dimensional Space

Doorways can be modeled more simply, since the width of our doors is approximately the resolution of the POMDP. A single exact-length state chain, as in Figure 3.12(a), leads through a door into a room. The state of a door (open or closed) can typically change and is thus often not known in advance. We therefore associate with doorways a probability p that the door is open. This probability encodes the uncertainty about the dynamic state of the environment. The observation probabilities associated with seeing a doorway are:

$$q(o|\texttt{door}) = p \times q(o|\texttt{open-door}) + (1-p) \times q(o|\texttt{closed-door}).$$

While we model corridors as one-dimensional chains of states, we represent foyers and rooms by tessellating two-dimensional space into a matrix of locations. From each location, the "forward" action has some probability of transitioning straight ahead, but also some probability of self-transitioning and moving to diagonally adjacent states which represents the robot drifting sideways without noticing it (Figure 3.15). Currently, we do not have a good approach for efficiently representing distance uncertainty in rooms and foyers.

3.3.3 Using the POMDP for Planning and Acting

In this section, we explain how the POMDP-based navigation architecture uses POMDP methods for estimating the robot's pose and for directing its behavior.

3.3.3.1 Pose Estimation

The pose-estimation component uses the motion and sensor reports to update the state distribution using Formulae (3.1) and (3.2) from Section 3.2.1. The updates are fast, since they require only one iteration over all states, plus an additional iteration for the subsequent normalization. While the amount of computation can be decreased by performing the normalization only once in a while (to prevent rounding errors from dominating), it is fast enough that we actually do it every time.

Reports by the same sensor in the same pose are clearly dependent since they depend on the same cells of the occupancy grid. Therefore, aggregating multiple reports without a motion report in between would not possess the Markov property. To avoid this problem, the pose-estimation component uses only the latest report from each sensor between motion reports to update the state distribution.

Motion reports tend to increase the pose uncertainty, due to nondeterministic transitions (actuator and distance uncertainty), while sensor reports tend to decrease it. An exception are "forward" motion reports that can decrease uncertainty if some of the states with nonzero probability are "wall facing," because knowing that a "forward" directive could be executed successfully carries information. In practice, this effect can be seen when the robot turns within a corridor junction: Before the turn, there is often some probability that the robot

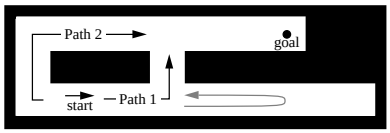


Figure 3.16: A Corridor Environment

has not yet reached the corridor junction. After the robot has turned and successfully moved forward a bit, the probability that it is still in the original corridor quickly drops. This is a major factor in keeping the pose uncertainty low, even when the robot travels long distances.

3.3.3.2 Policy Generation and Directive Selection

The policy-generation component has to compute a POMDP policy that minimizes the average travel time of the robot. The directive-selection component then simply indexes this POMDP policy repeatedly with the current state distribution to determine which directives to execute.

For policy generation, we need to handle "stop" directives. This is done by adding a "stop" action to the set $A(s)$ of each state s. The immediate cost of the "stop" action is zero in states that correspond to the goal pose and high otherwise, rapidly increasing with the distance from the goal pose. In all cases, executing a "stop" action stops the execution and the robot does not incur any future costs. The immediate costs of all actions reflect their execution times. The policy-generation component then has to determine a POMDP policy that minimizes the average total cost incurred. This minimizes the average travel time of the robot to the given goal pose (modulo discounting). This is similar to the action-penalty representation from Section 2.5.6.1.

If all actions have approximately the same cost, one can also use the goal-reward representation from Section 2.5.6.1. Executing a "stop" action again stops the execution and the robot does not incur any future rewards. This time, however, the immediate reward of the stop action is high in states that correspond to the goal pose and low otherwise. The immediate rewards of all other actions are zero. Thus, the only reward that the robot receives is for stopping. The policy-generation component then has to determine a POMDP policy that maximizes the average total reward. Discounting ensures that planning attempts to minimize the number of action executions. (Section 4.6.4 discusses the difference between using the action-penalty representation and the goal-reward representation.)

Optimal POMDP planning methods allow the robot to act optimally in the face of arbitrary initial pose uncertainty, including when and where to sense and how to trade-off between *exploitation* (acting to exploit existing knowledge) and *exploration* (acting to acquire further knowledge). Since our POMDPs typically have thousands of states, we need to use the greedy POMDP planning methods from Section 3.2.2. These methods pretend that the POMDP is completely observable. Under this assumption, they determine an optimal mapping from poses to directives (a policy) and then transform it into a mapping from pose *distributions* to directives (a POMDP policy). This, however, can lead to very suboptimal results. In Figure 3.16, for example, Path 1 is shorter than Path 2 and thus requires less travel time if the states are completely observable. Because of sensing uncertainty, however, a robot can miss the first turn on Path 1 and overshoot. It then has to turn around and look for the corridor opening again, which increases its travel time along Path 1. On the other hand, when the robot follows Path 2 it cannot miss turns. Thus, it might actually be faster to follow Path 2 than Path 1, but the greedy POMDP planning methods would always recommend Path 1.

We address this problem by using a different method for generating the policy, but continue to use the methods from Section 3.2.2 to complete it. Figure 3.17 depicts the resulting POMDP-based navigation architecture. To generate the policy, its uses Richard Goodwin's decision-theoretic path planner [Goodwin, 1997], that takes into account that the robot can miss turns and corridors can be blocked. The planner uses a generate, evaluate, and refine strategy to determine a path in the topological map that minimizes the average travel

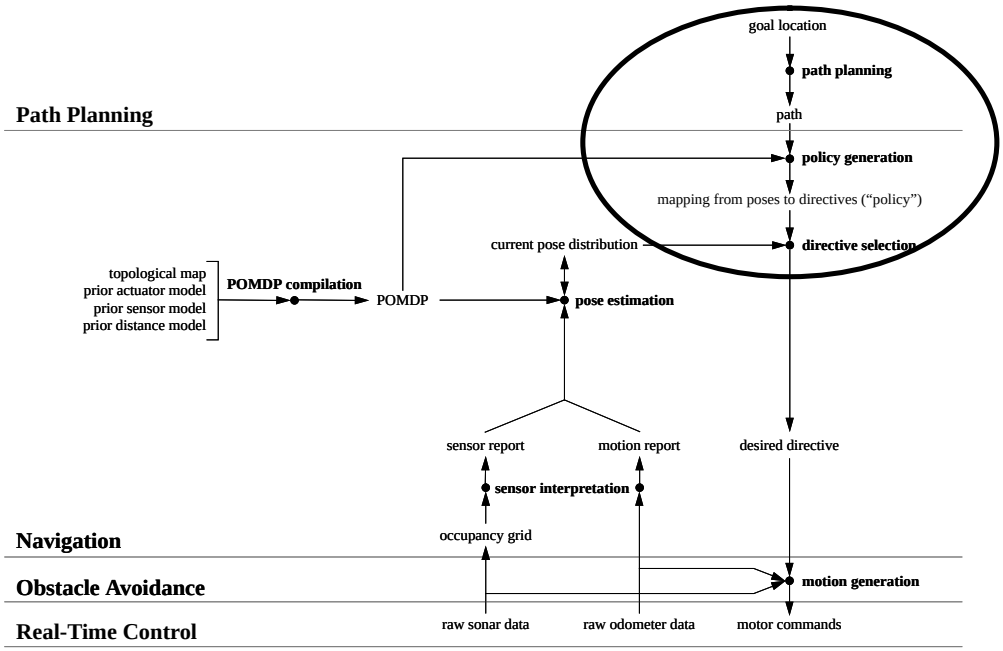


Figure 3.17: The Greedy POMDP-Based Navigation Architecture

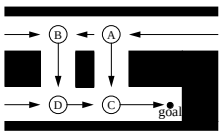


Figure 3.18: Two Parallel Corridors

time of the robot [Koenig *et al.*, 1996]. This also makes planning more efficient, since POMDP planners can plan for such contingencies only by augmenting the state information with the status of each blockage, which increases the number of states exponentially in the number of possible blockages. The POMDP-based navigation architecture then converts this path to a policy. For states corresponding to a topological node on the path, directives are assigned to head the robot towards the next node, except for the states corresponding to the last node on the path, which are assigned "stop" directives. Similarly, for all states corresponding to the topological edges between two nodes on the path, directives are assigned to head the robot towards the next node. Finally, for states not on the path, directives are assigned that move the robot back towards the planned path, again taking the travel times into account. This way, if the robot strays from the nominal (optimal) path, it will automatically execute corrective directives once it realizes its mistake. Thus, re-planning is only necessary when the robot detects that the nominal path is blocked. Figure 3.18 shows the resulting preferred headings for a simple corridor environment. Notice that the preferred heading between A and B is towards B because this way the robot does not need to turn around if it overshoots A. This minimizes its travel time, even though the goal distance is a bit longer. While this planning method is still suboptimal, it is a reasonable compromise between planning efficiency and plan quality. If suboptimal planning or changes to the building topology (for example, corridors being blocked) lead to limit cycles (similar to those described in [Simmons, 1994a]), such as spinning in place or repeatedly traversing the same area of the building, we have added execution monitors that detect the problem and re-plan.

As described in Section 3.2.2, there are several greedy POMDP planning methods for choosing which directive to issue, given a policy and the current state distribution. It turns out that the "Most Likely State" strategy

Figure 3.19: Xavier

does not work well with our models, because topological entities are encoded in multiple states. For example, since corridor junctions are modeled using several states for each orientation, it is reasonable to consider all of their recommendations when deciding which directive to issue. The "Voting" and "Completely Observable after the First Step" strategies both work well in our environment if the initial pose uncertainty is not overly large. Both strategies are relatively immune to the pose uncertainty that arises during navigation.

These greedy POMDP planning methods also have disadvantages. Since they operate on a policy and account for pose uncertainty only greedily, they make the assumption that the robot collects sensor data on the fly as it moves closer to its destination. As opposed to other POMDP planning methods, they do not plan when, where, and what to sense. This property fits virtual sensors based on ultrasonic sensors well, since ultrasonic sensors produce a continuous stream of data and do not need to be pointed. A disadvantage of this property is, however, that the robot does not handle well situations in which localization is necessary (including those with large initial pose uncertainty), and we thus recommend using more sophisticated POMDP planning techniques as they become more efficient. For example, it is often more effective to actively gather information that helps the robot to reduce its pose uncertainty, even if this requires the robot to move away from the goal temporarily.

On the other hand, the greedy POMDP planning methods often lead to an optimal behavior of the robot. For example, even if the robot does not know for certain which of two parallel corridors it is traversing, it does not need to stop and re-plan, as long as the directives associated with both corridors are the same. This way the robot can continue making progress towards its desired goal, while at the same time collecting evidence, in the form of sensor readings, that can help to disambiguate its true location. If, for example, the robot is trying to reach the goal pose in Figure 3.18 but is unsure as to whether it is at A facing West or B facing West, it does not need to resolve its uncertainty. Instead, it can turn left into the corridor, move to its end, turn left again, move to the end of that corridor and stop. It is likely that the robot will localize itself depending on whether it senses the corridor opening corresponding to C when it traverses the final corridor and localization is certain at the end of that corridor. Behaviors like this one take advantage of the fact that floors of buildings are usually constructed in ways that allow people to obtain sufficient clues about their current location from past experience and the local part of the environment only – otherwise they would easily become confused. Of course, ultrasonic sensors are much more noisy than human vision and cannot detect some landmarks that can be observed by people, such as signs or door labels, which makes the office-navigation task harder for robots than for people.

3.3.3.3 Experiments

The POMDP-based navigation architecture has to be evaluated experimentally because the world cannot be expected to completely satisfy the independence properties (Markov property) that POMDPs assume. In the end, one has to test experimentally whether they satisfy the Markov property well enough for the

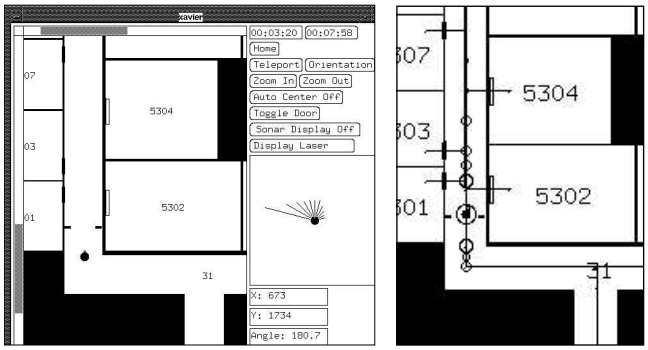


Figure 3.20: The Simulator (left) and Graphical Interface (right)

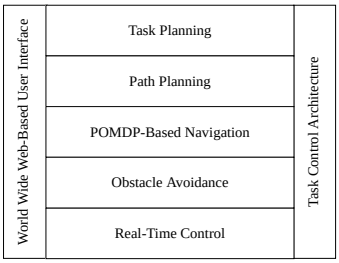


Figure 3.21: A Layered Architecture for Office-Delivery Robots

POMDP-based navigation architecture to yield reliable office-navigation behavior. In this section, we perform experiments with the POMDP-based navigation architecture in two environments for which the Markov property is only an approximation: Xavier, an actual mobile robot navigating in our building (Figure 3.19), and the real-time Xavier simulator (Figure 3.20), a realistic simulation of Xavier that allows us to perform repeatable experiments. The Xavier simulator is not based on the POMDP used for office navigation and consequently does not possess the Markov property assumed by that POMDP (just like reality). We use the same code for both sets of experiments, since the robot and its simulator have the exact same interfaces, down to the level of the real-time control layer.

The POMDP-based navigation architecture is implemented in C, and is one layer of the autonomous, mobile-robot system for office delivery (Figure 3.21) developed by the Xavier group at Carnegie Mellon University [Simmons *et al.*, 1997]. Besides the navigation layer described here, the layers of the system include a *real-time control* layer that provides the raw sensor data and controls the motors of the robot, Reid Simmons' *obstacle avoidance* layer that keeps the robot moving smoothly in a goal direction while avoiding static and dynamic obstacles [Simmons, 1996], Richard Goodwin's *path-planning* layer that uses a generate, evaluate, and refine strategy to find paths with minimal average travel time between two given locations on the topological map [Goodwin, 1997], and Karen Haigh's *task-planning* layer that uses PRODIGY [Veloso *et al.*, 1995] (a symbolic nonlinear planner) to schedule multiple office-navigation requests that arrive asynchronously [Haigh and Veloso, 1996]. The layers, which are implemented as a number of distributed, concurrent processes operating on several processors, are integrated using Reid Simmons' Task-Control Architecture which provides facilities for interprocess communication, task decomposition and sequencing, execution monitoring and exception handling, and resource management [Simmons, 1994b]. Finally, interaction with the robot is via the World Wide Web, which provides pages for both commanding the robot and monitoring its progress (Figure 3.22). Users worldwide can specify goal locations for Xavier on one floor of our building, and then monitor the execution of the office-navigation request by viewing frequent updates of both the current image taken by the camera on-board the robot and Xavier's most likely pose, as determined by the POMDP-based navigation architecture.

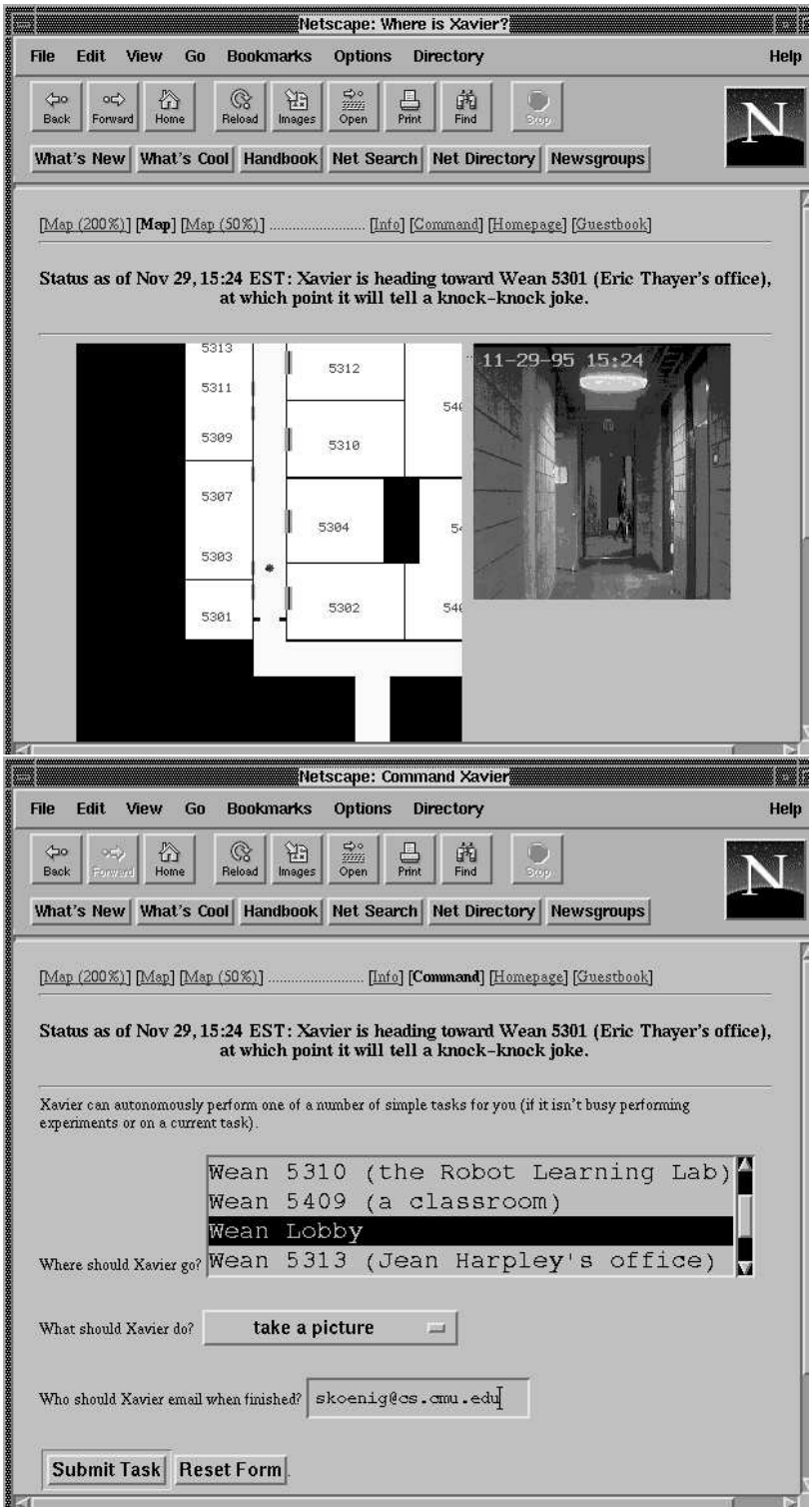


Figure 3.22: The World Wide Web Interface

In all experiments, we model the uncertainty about the length of each topological edge as a uniform distribution over the interval ranging from 80 to 150 percent of its true length and keep the initial pose uncertainty minimal: The initial probability for the robot's actual pose is about 70 percent. The remaining probability mass is distributed in the vicinity of the actual pose according to a normal distribution.

We report results for the "Voting" Strategy of policy generation and directive selection. The experiments illustrate that this efficient strategy performs well for the office-navigation tasks considered here and the Markov property is satisfied well enough for the POMDP-based navigation architecture to yield a reliable office-navigation performance. Cassandra *et al.* [Cassandra *et al.*, 1996] contains an experimental comparison of several greedy policy generation and directive selection strategies in more complex environments, but using simpler POMDPs than we use here.

3.3.3.3.1 Experiments with the Robot Xavier was designed and built by the Xavier group at Carnegie Mellon University. It is built on top of a 24 inch diameter RWI B24 base, which is a four-wheeled synchrodrive mechanism that allows for independent control of the translational and rotational velocities. The sensors on Xavier include bump panels, wheel encoders, a Denning ultrasonic sensor ring with 24 ultrasonic sensors, a front-pointing Nomadics laser light striper with a 30 degree field of view, and a Sony color camera that is mounted on a pan-tilt head from Directed Perception. The experiments do not use the laser light striper, and the camera is used only for fine positioning at the destination.

Control, perception, and planning are carried out on two on-board 66 Megahertz Intel 486 computers under the Linux operating system. An on-board color Intel 486 lap-top computer is used to monitor Xavier's status with a graphical interface, a screen shot of which is shown in Figure 3.23. The computers are connected to each other via Ethernet and to the outside world via a Wavelan radio connection.

A partial map of our environment is shown in Figure 3.23. This part corresponds to half of one floor of our building, which is half of the environment used in the experiments. The part shown has 95 topological nodes and 180 edges, and the corresponding POMDP has 3,348 states. In the period from December 1, 1995 to August 31, 1997, Xavier attempted 3,417 office-navigation requests and reached its intended destination in 3,227 cases, where each job required it to move 43 meters on average (Figure 3.24). Most failures are due to problems with Xavier's hardware (boards shaking loose) and the wireless communication between the on-board robot system and the off-board user interface (which includes the statistic-gathering software), and thus are unrelated to the POMDP-based navigation architecture. Failures that are attributable to the POMDP-based navigation architecture are often caused by its insufficient ways of modeling rooms. Rooms are modeled with only one group of four states and self-transitions. The self-transitions lead to the problem that decreasing the probability of being in a room takes a long time.

The completion rate is 94 percent. Its fluctuation is due to other researchers experimenting with their new code for other layers of the office-delivery system. The completion rate before using the POMDP-based navigation architecture was only 80 percent [Simmons, 1994a]. It was obtained on the same robot with an otherwise unchanged robot system except that a traditional landmark-based navigation technique was used in place of the POMDP-based navigation architecture. Thus, the difference in performance can be directly attributed to the different office-navigation techniques.

3.3.3.3.2 Experiments with the Simulator To show the performance of the POMDP-based navigation architecture in an environment that has a more complex topology than what we have available in our building, we also perform two experiments with the Xavier simulator. The topological map of the corridor environment shown in Figure 3.25 has seventeen topological nodes and eighteen edges, and the corresponding POMDP has 1,184 states.

First Experiment: In the first experiment, the task is to navigate from $start_1$ to $goal_1$. The preferred headings are shown with solid arrows in Figure 3.25. We ran a total of fifteen trials (Figure 3.26), all of which were completed successfully. The robot has to travel a rather long distance from $start_1$ before its first turn. Since this distance is uncertain and corridor openings are occasionally missed, the robot occasionally overshoots

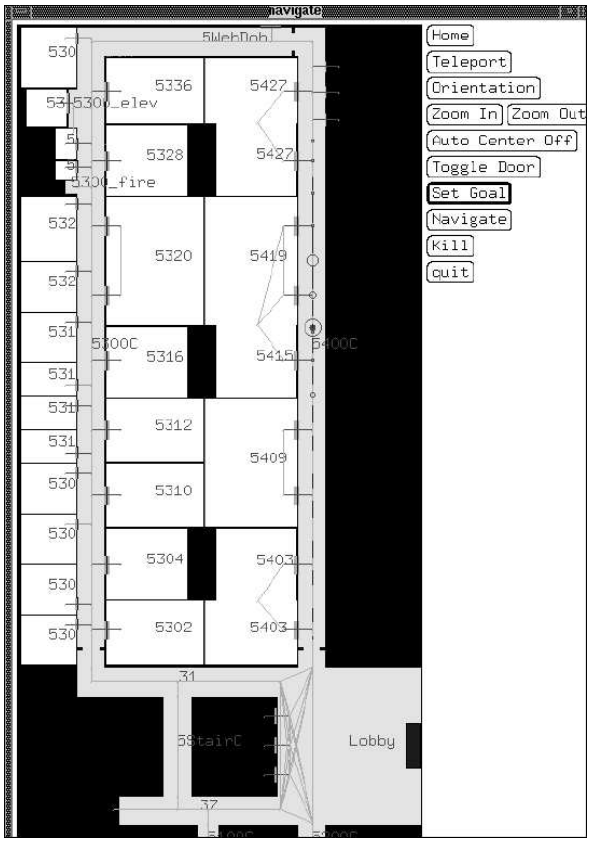


Figure 3.23: The Graphical Interface

B, and then becomes uncertain whether it is really at C or B. However, as discussed earlier in the context of Figure 3.18, this ambiguity does not need to be resolved since the same directive is assigned to both nodes. The robot turns left in both cases and then goes straight. The same thing happens when it gets to D, since it thinks it might be at either D or E. The robot eventually corrects its beliefs when, after turning left and traveling forward, it detects a corridor opening to its left. At this point, the robot becomes fairly certain that it is at E. A purely landmark-based navigation technique can easily get confused in this situation, since it has no expectations about seeing this corridor opening, and can only attribute it to sensor error, which, in this case, is incorrect.

Second Experiment: In the second experiment, the robot has to navigate from $start_2$ to $goal_2$. The preferred headings for this office-navigation task are shown with dashed arrows in Figure 3.25. Again, we ran fifteen trials (Figure 3.27). For reasons that are similar to those in the first experiment, the robot can confuse G with F. If it is at G but thinks it is probably at F, it turns right and goes forward. However, when it detects the end of the corridor but does not detect a right corridor opening, it realizes that it is at H rather than I. Since the probability mass has now shifted, it turns around and goes over G, F, and I to the goal. This shows that the POMDP-based navigation architecture can gracefully recover from misjudgments based on wrong sensor reports – even if it takes some time to correct its beliefs. It is important to realize that this behavior is not triggered by any explicit exception mechanism, but results automatically from the way the pose estimation and directive selection interact.

3.3.4 Using the POMDP for Learning

Month	Days in Use	Jobs Attempted	Jobs Completed	Completion Rate	Distance Traveled
December 1995	13	262	250	95 %	7.7 km
January 1996	16	344	310	90 %	11.4 km
February 1996	15	245	229	93 %	11.6 km
March 1996	13	209	194	93 %	10.0 km
April 1996	18	319	304	95 %	14.1 km
May 1996	12	192	180	94 %	7.9 km
June 1996	7	179	170	95 %	8.2 km
July 1996	7	122	121	99 %	5.4 km
August 1996	—	—	—	— %	— km
September 1996	9	178	165	93 %	8.2 km
October 1996	11	168	151	90 %	7.8 km
November 1996	11	228	219	96 %	10.5 km
December 1996	9	172	167	97 %	8.8 km
January 1997	9	157	154	98 %	7.5 km
February 1997	4	87	77	89 %	4.2 km
March 1997	1	13	13	100 %	0.6 km
April 1997	4	51	46	90 %	2.7 km
May 1997	6	60	58	97 %	2.9 km
June 1997	11	148	145	98 %	7.1 km
July 1997	10	131	128	98 %	6.0 km
August 1997	13	152	146	96 %	5.8 km
Total	199	3,417	3,227	94 %	148.4 km

Figure 3.24: Performance Data (all numbers are approximate)

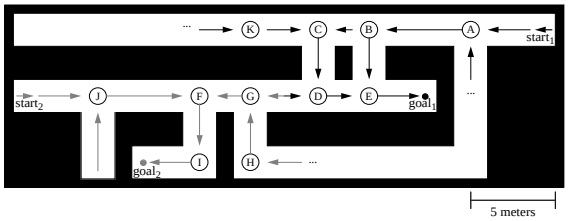


Figure 3.25: A Corridor Environment

Path	Frequency	Time
ABE	12	68.2 s
ABCDE	3	79.7 s

Figure 3.26: The First Experiment

Path	Frequency	Time
JFI	11	60.6 s
JFGFI	2	91.5 s
JFGHGFI	1	116.0 s
JFGFGFI	1	133.0 s

Figure 3.27: The Second Experiment

So far, we have assumed that the POMDP is compiled from a topological map and given actuator, sensor, and distance models. Two common approaches are to provide the robot with the models or let it learn the models autonomously by exploring its environment. Both approaches have disadvantages:

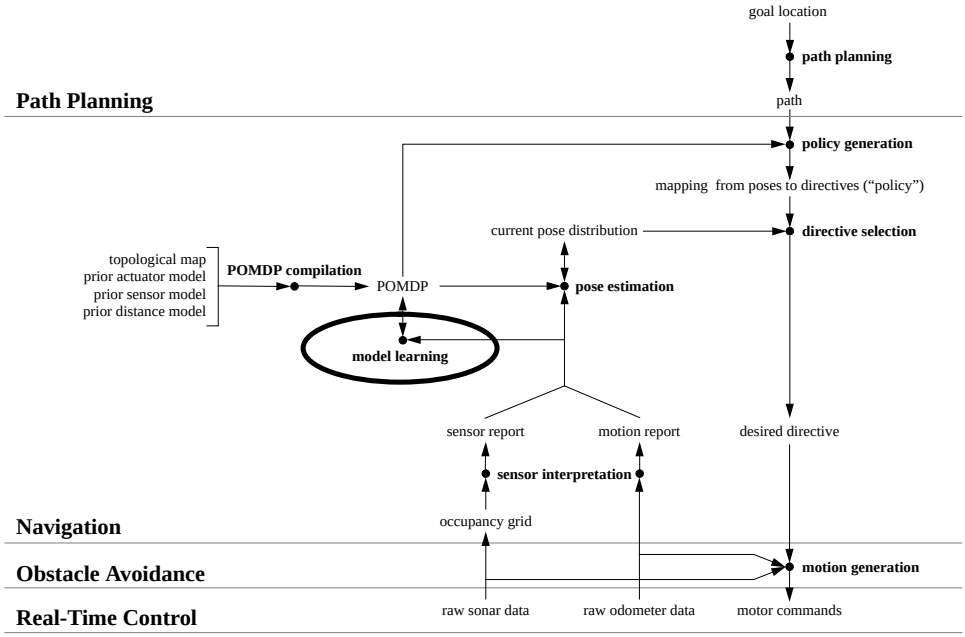


Figure 3.28: The Greedy POMDP-Based Navigation Architecture with Learning

- **Providing the robot with the necessary information** suffers from the problem that some information is difficult or impossible to provide by humans. For example, the actuator and sensor models depend on the environment of the robot, such as how slippery the floor is, how wide the corridors are, or how well the walls reflect ultrasonic waves. Thus, accurate models cannot be factory programmed. The models also depend on characteristics of the robot itself (its actuators and sensors, for example) and one cannot expect naive users of a robot (consumers) to be familiar with details of their newly purchased delivery robot. Even expert users have problems specifying the models because the models depend also on communication delays among distributed processes and their relationship to the speed with which the robot moves. Some information could be provided by users or expert users, but might be cumbersome to obtain. If the users do not know the exact lengths of their corridors, for example, they have to measure them – a task that the robot could do itself.

- **Letting the robot explore its environment autonomously**, an approach that many researchers have investigated, suffers from the problem that the robot cannot be used immediately and, during exploration, is likely to get into situations of confusion or danger that require human intervention, since it has no initial knowledge of its environment.

We therefore suggest using a third approach: providing the robot with the information that is easily available to humans, and then letting it autonomously learn the rest of the information needed for reliable office navigation while in the process of performing its office-navigation tasks.

We start by supplying the robot with a topological map of its environment and other information, such as initial approximate actuator and sensor models. The topological map can easily be obtained from a sketch drawn by people familiar with the environment in a way similar to what is done for interface design [Landay, 1995]. Figure 3.29 (center and right), for example, shows a sketch of a corridor environment and the corresponding topological map. This information is sufficient for the robot to perform office-navigation tasks and collect training data. Learning then uses this data to make navigation both more reliable and efficient by adapting the actuator and sensor models to the environment and learning distance information.

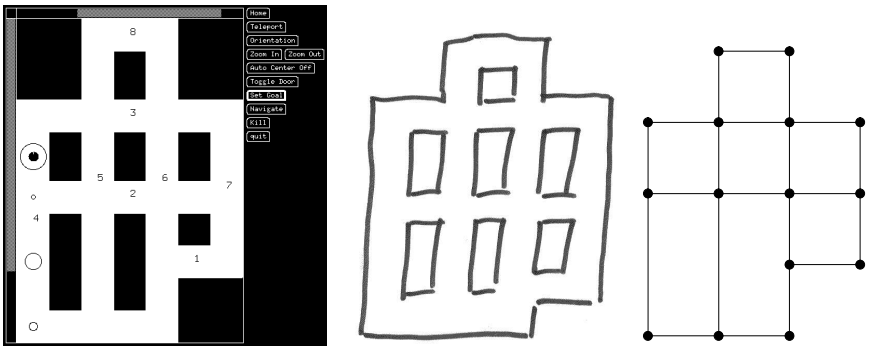


Figure 3.29: A Corridor Environment, its Sketch, and the Corresponding Topological Map

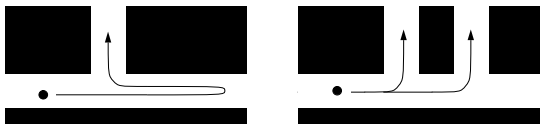


Figure 3.30: Two Examples for the Utility of Learning Distances

The advantage of learning distance information is illustrated in Figure 3.30. It takes the robot longer to notice that it overshot its destination when it is uncertain about the position of the orthogonal corridor (left) and it might even turn into the wrong corridor of two adjacent corridors if the first one is temporarily blocked (right). Notice, however, that we do not want to provide the robot with distance information that is generated from sketches of corridor environments because people often err with respect to distances, unless they measure them. Although the sketch of Figure 3.29 (center) correctly specifies the topology of the environment, some of the arc lengths are incorrect. It is therefore much more reliable and convenient to let the robot learn the distance models itself.

We desire that the learning be unsupervised and passive. *Unsupervised learning* means that the robot gets no additional information from a teacher after it has been supplied with the initial topological map and other easily available information. In particular, during office navigation, the robot is not told where it really is or what it really observed. This is desirable because providing the robot with correct pose information is a tedious task. Unsupervised learning, on the other hand, can be done autonomously and, ideally, without any external help. *Passive learning* means that the robot learns while it is performing office-navigation tasks; the learning method does not need to control the robot at any time (for example, to execute localization actions). This is desirable because then the robot does not need a separate training phase; it can be used immediately to perform office-navigation tasks and collects training data whenever it moves. This way it never stops learning and improves its office-navigation performance continuously. Since learning takes place while the robot is performing its office-navigation tasks, the training data reflect the environment in which the robot has to perform. The robot also gains more information about routes that it traverses more often and, as a result, learning focuses its attention on routes that are more relevant for the office-navigation tasks encountered.

Unsupervised, passive learning is difficult in the presence of state uncertainty, however, because there is no ground truth to compare against. For example, the robot's pose uncertainty prevents it from simply learning corridor lengths by first moving to the beginning of a corridor and then to its end while measuring the distance traveled. A similar problem arises for learning actuator and sensor models. For example, if the robot always knew its pose exactly, it could learn the sensor probabilities simply by counting frequencies, for example, how often sensor i reports feature f in state s. However, because the robot is uncertain about its pose it does not know exactly when it was in state s. Furthermore, the pose uncertainty of the robot can be quite significant

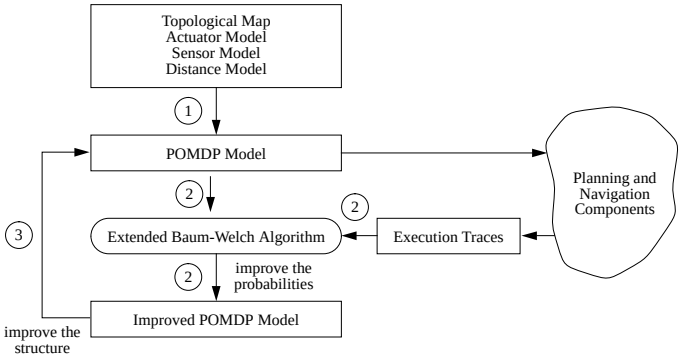


Figure 3.31: An Overview of GROW-BW

even with good models, as shown in Figures 3.23 and 3.20 (right). The sizes of the small circles in these figures are proportional to the probability mass with which the robot believes itself to be in each corridor. The amount of pose uncertainty shown is typical. In addition, robot learning is hard because it must run within the CPU and memory constraints of the robot's computers, and must deal with the fact that collecting data is time consuming.

Our POMDP learning method, called the *Grow Baum-Welch Method* (GROW-BW), is an unsupervised, passive learning technique that addresses these concerns. Whenever the robot moves, GROW-BW gains more and more experience with the environment, in form of execution traces, which it continually uses to change the structure of the initial POMDP and fine-tune the initial ("factory programmed") probabilities so that they more closely reflect the actual environment of the robot. This improves the accuracy of the actuator and sensor models and reduces the uncertainty about the corridor lengths, ultimately resulting in an improved navigation performance of the robot. GROW-BW extends the Baum-Welch method to decrease its memory consumption and training-data requirements, and to enable it to change the topology of the initial POMDP.

GROW-BW uses an extension of the Baum-Welch method as a subroutine. The Baum-Welch method overcomes the problem of unsupervised, passive learning in the presence of pose uncertainty by using the given POMDP and execution trace to derive state distributions for each point in time. It then uses these state distributions when counting frequencies and, this way, is able to determine improved transition and sensor probabilities (Section 3.2.3). To enable the Baum-Welch method to run on-board the robot, we have extended it to use a sliding time window on the execution trace, which decreases its memory requirements while producing comparable results and maintaining its efficiency. To reduce the need for training data, the extension utilizes additional knowledge in the form of constraints, such as symmetry in the sensors or the topological map. For example, it can utilize information such as the "left" and "right" sensors being identical or two corridors having the same (although unknown) length. The extended Baum-Welch method learns the best POMDP for a given structure. Based on the resulting POMDP, GROW-BW then decides whether to change the structure and repeat the process. The resulting method improves the actuator, sensor, and distance model efficiently in terms of running time, memory consumption, and the required length of the execution trace.

Figure 3.31 summarizes GROW-BW: First, GROW-BW compiles the initial POMDP from the topological map augmented with initial actuator and sensor models and an initial distance model (Section 3.3.2). Lacking other information, it simply assumes that actuation and sensing work perfectly (modulo a small amount of noise) and that all possible corridor lengths are equally likely ①. GROW then uses the extended Baum-Welch method to improve the POMDP based on the execution traces generated during office navigation (Figure 3.6) ②. The resulting POMDP has improved actuator and sensor models and less distance uncertainty but the same structure as the initial POMDP. GROW-BW then uses a hill-climbing technique that iteratively changes the structure of the POMDP based on the results of the extended Baum-Welch method ③. In the following, we

first describe our implementation of the Baum-Welch method, then our extensions, and finally GROW-BW in detail.

3.3.4.1 The Baum-Welch Method

The Baum-Welch method (Section 3.2.3) is an ideal candidate for the implementation of an unsupervised, passive learning method, because it is efficient and does not introduce transitions between states that were not present in the initial POMDP. Thus, the improved POMDP always conforms to the topological map.

When implementing the Baum-Welch method as part of our learning methods, we made the following design decisions:

Updating the Sensor Probabilities: We are not really interested in updating the observation probabilities $q(o|s)$ as is done by the Baum-Welch method; we want to update the sensor probabilities $q_i(f|s)$ instead. Our implementation of the Baum-Welch method therefore uses the following re-estimation formula instead of re-estimation formula A10 (from page 91):

A10'. Set $\overline{q}(f|s) := \sum_{t=1\ldots T}[r_{i,t}(f)\gamma_t(s)]/\sum_{t=1\ldots T}\gamma_t(s)$ for all $i \in I$, $f \in F(i)$, and $s \in S$.

Influence of the Initial POMDP: Because the Baum-Welch method converges to a local optimum, the final POMDP can, in theory, depend on the initial POMDP. We found that the Baum-Welch method is very robust towards variations of the initial probabilities. This might be due to the fact that our initial POMDP is usually pretty accurate: Its basic structure is correct and even rough estimates of the actuator and sensor models (like the assumption that actuation and sensing work almost perfectly) provide good estimates. For example, it is reasonably safe to assume that a sensor reports the correct feature with larger frequency than incorrect features (with the exception of the feature `unknown`). Thus, the initial POMDP is a good starting point for the hill-climbing search of the Baum-Welch method and perturbations of the initial POMDP do not change this starting point significantly. Consequently, local maxima do not appear to be a problem when applying the Baum-Welch method to office-navigation tasks, making it unnecessary to perturb the probabilities of the initial POMDP multiple times and applying the Baum-Welch method to each of the resulting POMDPs. Therefore, our implementation applies the Baum-Welch method only once, namely, to the initially given POMDP, after it has added a small amount of noise to the probabilities. Perturbing the probabilities of the initial POMDP is sometimes necessary to distinguish otherwise equivalent states in the initial POMDP since the Baum-Welch method is unable to remove symmetries in the POMDPs.

3.3.4.2 The Extended Baum-Welch Method

Despite its theoretical elegance, the Baum-Welch method has two deficiencies that make it impractical for robots: its memory and training-data requirements. We address these problems by extending the Baum-Welch method in straightforward ways [Koenig and Simmons, 1996f]. We have not seen these extensions before, but expect that variants of some of them might have been developed independently in the speech literature. The *extended Baum-Welch method* might no longer be guaranteed to converge, but this does not appear to be a problem in practice.

3.3.4.2.1 Memory Requirements The Baum-Welch method has to run on-board the robot and share its memory with many other processes that run concurrently. Traditional implementations of the Baum-Welch method need arrays of floating point numbers whose sizes are on the order of the product of the number of states and the length of the execution trace. This is a problem since even our smallest POMDPs have thousands of states, the Baum-Welch method needs execution traces with hundreds of action executions to get sufficient data, and the length of the execution trace grows over time as more data are collected. Since many other processes are run on the same on-board computer, the memory requirements of a learning method should be rather small and relatively constant.

C1. Set $\overline{p}(s'|s,a) := 0$ for all $s, s' \in S$ and $a \in A(s)$.

C2. Set $\overline{q}(f|s) := 0$ for all $i \in I$, $f \in F(i)$, and $s \in S$.

C3. Initialize $scale_1$ using A1 and α_1 using A2 (both from page 90).

C4. Set $start := 1$ and $end := x$.

C5. While $start < T$:

(a) If $end > T$, then set $end = T$.

(b) Calculate $scale_t$ and α_t for $t = start \ldots end$, working forward from $scale_{start}$ and α_{start} using A3(a) and A3(b) (from page 90). Previously calculated scaling factors and alpha values can be re-used.

(c) Approximate the beta values β_t for $t = start \ldots end$, initializing $\beta_{end}(s) := 1/scale_{end}$ for all $s \in S$ and working backward from β_{end} using A5(a) (from page 91). Previously calculated beta values cannot be re-used.

(d) If $end = T$, then set $newstart := T$ else set $newstart := end - la_{min} + 1$.

(e) For all $t = start \ldots newstart - 1$:

i. Calculate $\gamma_t(s, s')$ for all $s, s' \in S$ with $a_t \in A(s)$, using α_t and the approximation of β_{t+1} in A6 (from page 91).

ii. Calculate $\gamma_t(s)$ for all $s \in S$, using α_t and the approximation of β_t in A7 (from page 91).

iii. Set $\overline{p}(s'|s,a_t) := \overline{p}(s'|s,a_t) + \gamma_t(s,s')$ for all $s, s' \in S$ with $a_t \in A(s)$ (these values will be normalized in C6).

iv. Set $\overline{q}(f|s) := \overline{q}(f|s) + r_{i,t}(f)\gamma_t(s)$ for all $i \in I$, $f \in F(i)$, and $s \in S$ (these values will be normalized in C7).

(f) Forget all $scale_t$ and α_t for $t = start \ldots newstart - 1$, and all β_t for $t = start \ldots end$.

(g) Set $start := newstart$ and $end := newstart + x - 1$ (that is, move the time window).

C6. Set $\overline{p}(s|s,a) := \overline{p}(s'|s,a) / \sum_{s' \in S} \overline{p}(s'|s,a)$ for all $s, s' \in S$ and $a \in A$ (that is, normalize the transition probabilities).

C7. Set $\overline{q}(f|s) := \overline{q}(f|s) / \sum_{f \in F(i)} \overline{q}(f|s)$ for all $i \in I$, $f \in F(i)$, and $s \in S$ (that is, normalize the sensor probabilities).

Figure 3.32: The Window-Based Baum-Welch Method

Remember that the alpha values α_t take all the information in a prefix of the training trace into account and are calculated forward from $t = 1$. The beta values β_t, in contrast, take all the information in a suffix of the training trace into account and are calculated backward from $t = T$. The gamma values γ_t combine the information from the alpha and beta values and take all the information of the training trace into account. To reduce the amount of memory space of the Baum-Welch method, one could modify the Baum-Welch method to calculate the alpha and beta values anew for each point in time. This, however, would be giving up the running-time efficiency of the Baum-Welch method. Standard pruning methods make the memory requirements effectively proportional to the length of the execution trace while maintaining running-time efficiency [Young, 1994]. Here, we reduce the memory requirements even further while maintaining its running-time efficiency. We use a sliding *time window* on the execution trace. Time windows add a small overhead to the running time and cause a small loss in precision of the improved POMDP, but decouple the memory requirements from the length of the execution trace: the memory requirements are independent of the length of the execution trace and can be scaled (even dynamically) to the available memory space.

The *window-based Baum-Welch method* proceeds similarly to the traditional Baum-Welch method. It still calculates the alpha values precisely, since they can be calculated incrementally with only two arrays whose sizes are the number of states: one for the current alpha values and another one for the alpha values of the next time step. The beta values, however, are approximated using a sliding time window on the execution trace, that is, using a limited look-ahead. The gamma values are then calculated using the alpha and beta values, as before. Consequently, the gamma values at time t are approximated by conditioning them on all information in the execution trace before time t, but only a small look-ahead after that. This can approximate the gamma values closely because floors of buildings are usually constructed in ways that allow people to obtain sufficient clues about their current location from past experience and the local part of the environment only – otherwise they would easily become confused.

Figure 3.32 describes the window-based Baum-Welch method. For simplicity, the figure does not show how

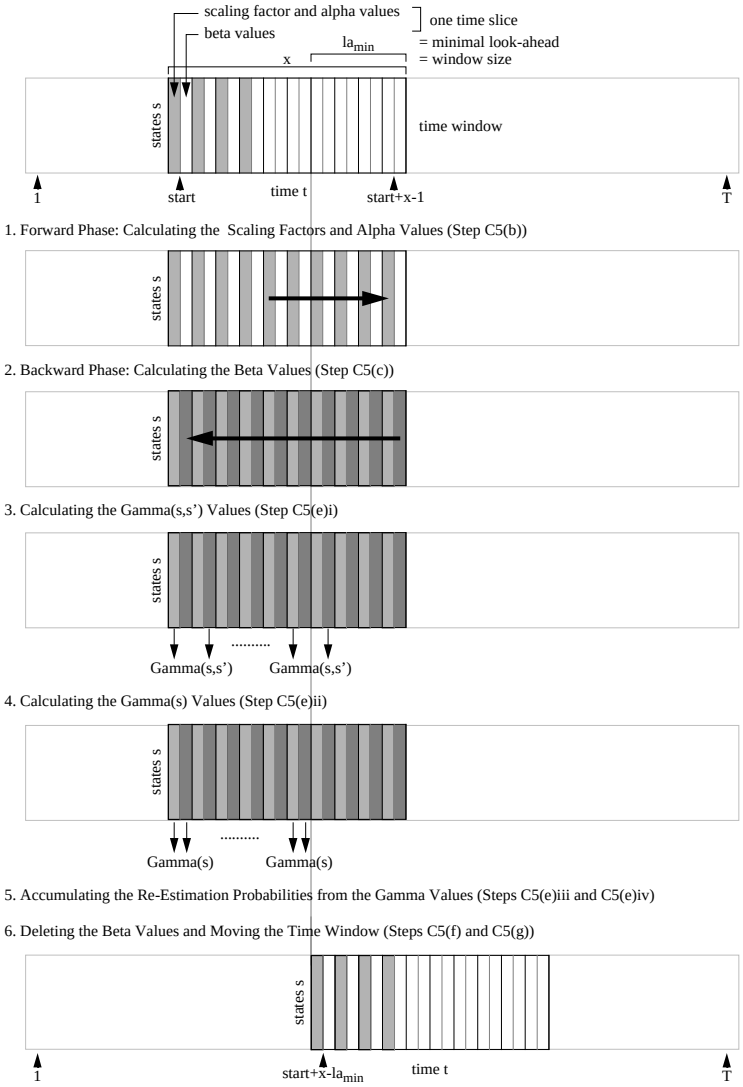


Figure 3.33: An Overview of the Window-Based Baum-Welch Method

the initial state distribution π is updated or the values $\gamma_T(s)$ are calculated. Figure 3.33 summarizes the calculations of the window-based Baum-Welch method between two movements of the time window, in a way similar to how Figure 3.4 summarized the calculations of the traditional Baum-Welch method. First, the window-based Baum-Welch method calculates the scaling factors and alpha values for all time steps in the time window, working forward from the first time step in the time window using the dynamic programming formulae A3(a) and A3(b) from page 90 (re-using previously calculated scaling factors and alpha values). Then, it calculates the beta values for all time steps in the time window under the approximation assumption that the last time step in the time window is also the last time step in the execution trace, working backward from the last time step in the time window using the dynamic programming formula A5(a) from page 91 (previously calculated beta values cannot be re-used). Next, it calculates the gamma values in a prefix of the time window, and accumulates the re-estimation probabilities from the gamma values to calculate the quantities in Steps A9 and A10 from page 91 incrementally. Finally, it deletes the scaling factors and alpha values that are no longer necessary and the beta values that cannot be re-used, moves the time window, and repeats the process. Once it has reached the end of the execution trace, the window-based Baum-Welch method normalizes the accumulated re-estimation probabilities, resulting in the probabilities that constitute

the improved POMDP. Notice that the window-based Baum-Welch method can read the motion and sensor reports sequentially, for example, from a file; they do not need to be in memory all the time.

A time window of size x stores the scaling, alpha, and beta values from $t = start \ldots start+x-1$. The beta values $\beta_t(s)$ are approximated with $(p(o_{t+1 \ldots start+x-1}|s_t = s, a_{t \ldots start+x-2})p(\text{can execute } a_{t \ldots start+x-2}|s_t = s))/(p(o_{t \ldots start+x-1}|o_{1 \ldots t-1}, a_{1 \ldots start+x-2})p(\text{can execute } a_{t-1 \ldots start+x-2}|o_{1 \ldots t-1}, a_{1 \ldots t-2}))$. This approximates the gamma values $\gamma_t(s, s')$ with $p(s_t = s, s_{t+1} = s'|o_{1 \ldots start+x-1}, a_{1 \ldots start+x-2})$ and the gamma values $\gamma_t(s)$ with $p(s_t = s|o_{1 \ldots start+x-1}, a_{1 \ldots start+x-2})$. To guarantee a *minimal look-ahead* la_{min}, the gamma values are only calculated for $t = start \ldots start + x - la_{min} - 1$. This way, the *actual look-ahead* of the gamma values $\gamma_t(s, s')$ and $\gamma_t(s)$ is $(start + x - 1) - t$, which is guaranteed to be at least as large as the minimal look-ahead la_{min}. The minimal look-ahead is a parameter of the window-based Baum-Welch method with $1 \leq la_{min} \leq x - 1$. It is chosen so that the gamma values are approximated closely. Currently, we use a constant value la_{min} for all t, but it is easy to modify the window-based Baum-Welch method to use arbitrary nonuniform minimal look-aheads.

With the window-based Baum-Welch method, there is a tradeoff between running time, precision of the improved POMDP, and memory requirements. All three factors depend on the window size x and the minimal look-ahead la_{min}.

Running-Time Overhead: In addition to a small amount of bookkeeping required for maintaining the time window, the window-based Baum-Welch method incurs overhead only for calculating the beta values repeatedly. While the traditional Baum-Welch method calculates every state distribution β_t only once, the window-based Baum-Welch method calculates it on average $x/(x - la_{min}) \geq 1$ times for long execution traces, because the time window is always moved by $x - la_{min}$ time steps after all x beta state distributions in the time window have been calculated once (we count the initialization of the beta values as one calculation).

Precision: The precision of the improved POMDP depends on the average actual look-ahead la_{eff}. The larger this value, the more precise the gamma values and, consequently, the improved POMDP. $la_{eff} = 1/(x - la_{min}) \times \sum_{t'=la_{min} \ldots x-1} t' = 1/2 \times (x + la_{min} - 1)$ for long execution traces, because $x - la_{min}$ gamma state distributions with look-aheads ranging from la_{min} to $x - 1$ are calculated each time before the time window is moved. Thus, the average actual look-ahead is usually much larger than la_{min} if the window size is large.

Memory Requirements: The window-based Baum-Welch method can operate with as much or as little memory space as is available. In particular, a time window of size x only needs arrays of floating point numbers whose sizes are on the order of x times the number of states, and thus its memory requirements no longer depend on the length of the execution trace. The memory requirements can be dynamically scaled to the available memory space and the required precision. In fact, the window-based Baum-Welch method can utilize additional memory space immediately when it becomes available to increase the window size.

To summarize, the precision can be increased by increasing the minimal look-ahead la_{min} or the window size x. Increasing the minimal look-ahead produces a small amount of running-time overhead, but leaves the memory requirements unchanged (if $x > la_{min} + 1$); increasing the window size decreases the overhead, but increases the amount of memory space needed. We therefore suggest making the window size as large as possible and setting the minimal look-ahead based on the average distance between corridor junctions, because the most useful sensor reports are obtained when the robot traverses corridor junctions. If the time window is sufficiently large, the window-based Baum-Welch method behaves identically to the Baum-Welch method, both in terms of running time and the POMDP that it produces.

3.3.4.2.2 Training-Data Requirements The traditional Baum-Welch method requires a long execution trace: As the degree of freedom (that is, the number of probabilities to be estimated) increases, so does the need for training data, to decrease the likelihood of over-fitting the model. Given the relatively slow speed at which mobile robots can move, we want the extended Baum-Welch method to learn good POMDPs with as short a travel distance as possible. Thus, we use several methods to decrease the number of probabilities that must be adjusted. All of the methods utilize domain knowledge to keep the number of probabilities

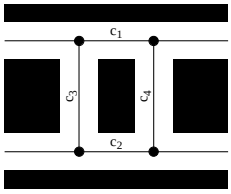


Figure 3.34: Equally Long Corridors

small. Nothing prevents one from using the Baum-Welch method with an initial POMDP that has not been derived from a topological map but consists only of a large number of completely interconnected states. The initial structure of our POMDP, on the other hand, already reduces the number of probabilities considerably by disallowing transitions that are clearly impossible, such as tele-porting to distant locations. We employ three additional techniques to reduce the degrees of freedom further or provide a better bias:

Leaving Probabilities Unchanged: The extended Baum-Welch method does not adjust probabilities that we believe to be approximately correct. Actuator and sensor models, for example, are often similar in different environments and consequently need be learned only once.

Imposing Equality Constraints: The extended Baum-Welch method constrains some probabilities to be identical. This has the advantage that the Baum-Welch method can now update a probability using all the information that applies to any probability in its class. If the equality constraints are only approximately correct, we trade off model quality and thus predictive power for the length of the execution trace required to learn the POMDP. Consider the following examples:

- **Actuator Model:** We assume that the models for the "left turn" and "right turn" actions are the same for all states. We further constrain the "left turn" and "right turn" probabilities to be symmetrical.
- **Sensor Model:** Instead of learning separate sensor models for each state, we learn them for classes of states (Figure 3.11). These classes reflect our prior knowledge about how the sensors are supposed to operate – they are currently predefined and not learned. For example, all states that have a wall on their left are construed to have the same probability that the "left" sensor reports a wall. The extended Baum-Welch method also assumes that the "left" and "right" sensors behave symmetrically, so their models are constrained to have the same probabilities. For example, the probability that the "left" sensor correctly reports a wall to the left of the robot equals the probability that the "right" sensor correctly reports a wall to the right of the robot.
- **Distance Model:** We constrain the transition probabilities of the "forward" actions for "corridor junction" states that lead into the same corridor to be identical. An example are states X and Y in Figure 3.12. This forces the length estimates for a corridor to be the same in both directions. In general, we group all corridor junction states that are known to lead into equally long corridors. These geometrical constraints can often be deduced from the topological map. One might know, for example, that two corridors are the same length, because both are intersected orthogonally by the same pair of corridors. In Figure 3.34, for example, it holds that $l_{true}(c_1) = l_{true}(c_2)$ and $l_{true}(c_3) = l_{true}(c_4)$, although the lengths themselves are unknown.

For example, after forming classes $c \in C$ of states (page 97) but before imposing equality constraints between different sensors, the re-estimation formula for the sensor probabilities A10' (from page 114) becomes

A10". Set $\bar{q}(f|c) := \sum_{t=1\ldots T} \sum_{s \in c} [r_{i,t}(f)\gamma_t(s)] / \sum_{t=1\ldots T} \sum_{s \in c} \gamma_t(s)$ for all $i \in I$, $f \in F(i)$, and $c \in C$.

Using Bayesian Probability Estimates: Imposing equality constraints enables the Baum-Welch method to operate with smaller execution traces. However, frequency-based estimates, as used by the Baum-Welch

method to re-estimate the probabilities, are not very reliable if the sample size is small. To understand why, consider the following analogy: If a fair coin were flipped once and came up heads, the frequency-based estimate would set $p(heads) = 1$. This completely overfits the sample (Section 3.2.3). If this model were used to predict future coin flips, one would be very surprised if the coin came up tails next time – this would be inconsistent with the learned model. This is mostly a problem for the transition probabilities of the "forward" actions that are used to estimate the corridor lengths since, to collect n data points, the robot has to pass through the corresponding state n times if no equality constraints on the lengths are known. Thus, n is typically very small. Learning the other transition and observation probabilities can take advantage of forming classes to quickly collect many data points. To avoid the problem of frequency-based estimates, we change the re-estimation formulae A9 (from page 91) and A10' (from page 114) to use Bayes rule (Dirichlet distributions) instead of maximum-likelihood estimates in form of frequencies. Both methods produce asymptotically the same results for long execution traces. However, using Bayes rule solves three problems for shorter execution traces that implementations of the Baum-Welch method often tackle using special techniques: The problem of over-fitting the execution trace, the problem that probabilities easily get close to zero and one (Section 3.2.3), and the problem of small denominators in the original re-estimation formulae A9 and A10 (from page 91). If the denominators of the re-estimation formulae A9 or A10 are small, then the execution trace does not contain sufficient experience to estimate these probabilities reliably.

Appendix 6.2 contains the derivation of the new re-estimation formulae. The re-estimation formula A9 for the transition probabilities, for example, becomes (probability classes are not shown):

A9'. Set $\bar{p}(s'|s,a) := (\sum_{t=1\ldots T-1|a_t=a} \gamma_t(s,s') + k\ p(s'|s,a)))/(\sum_{t=1\ldots T-1|a_t=a} \gamma_t(s) + k)$ for all $s, s' \in S$ and $a \in A(s)$.

In this formula, $p(s'|s,a)$ are the transition probabilities before learning and $k > 0$ is a constant whose magnitude indicates the confidence that one has in the initial probabilities. If k were set to zero, then the original re-estimation formula A9 resulted. If k were set to infinity, then the transition probabilities did not change at all. For us, $k = 1$ works well. It makes the probabilities change sufficiently while avoiding the problems of frequency-based estimates.

3.3.4.3 GROW-BW

The Baum-Welch method and the extensions we have made can improve the probabilities of a POMDP, but never change its structure. This is desirable since it guarantees that the improved POMDP always conforms to the topological map. It also poses a problem, however, because the distance model is partly encoded in the structure of the POMDP. In particular, the possible lengths $l \in [l_{min}(c), l_{max}(c)]$ of some corridor c are determined by the structure, while the probability distribution p_c over l is determined by the transition probabilities. Consequently, the extended Baum-Welch method cannot assign a positive probability $p_c(l)$ to lengths $l \notin [l_{min}(c), l_{max}(c)]$. Thus, it cannot learn the true length $l_{true}(c)$ if it is not within the bounds. Estimating a lower bound $l_{min}(c)$ on $l_{true}(c)$ is easy even if the bounds are unknown: We can use the smallest positive length according to our discretization granularity. Estimating an upper bound $l_{max}(c)$ on $l_{true}(c)$ is harder: We could, of course, estimate a ridiculously large value, but this has the drawback that the POMDPs become very large – and so do the running time and memory consumption. Since these factors determine the tractability of POMDP learning methods, we investigate learning methods that are able to actually change the structure of the POMDP.

Alternatives to the Baum-Welch method for learning POMDPs are described in [Chrisman, 1992, Stolcke and Omohundro, 1993, McCallum, 1995a], among others. These methods are able to change the structure of a POMDP, but have the disadvantage that they either require a long execution trace, learn task-specific representations only, or cannot utilize prior knowledge. Consequently, we have designed a novel learning method that we call the *Grow Baum-Welch Method* (GROW-BW) [Koenig and Simmons, 1996e]. GROW-BW achieves its power by utilizing the regularities in the structure of the POMDPs used by the POMDP-based navigation architecture. It takes advantage of the fact that the extended Baum-Welch method learns a good

GROW-BW uses the following parameters: $X = 0, 1, 2, \ldots$; $Y = 0, 1, 2, \ldots, X$; $Z = 0, 1, 2, \ldots$, and $P \in (0, 1)$. In its simplest form, it uses $X = Y = Z = 0$ and a small positive value for P. It then operates as follows:

1. For each corridor c: set $l_{max}(c) := l_{min}(c) + X + 1$. (If a lower bound $l_{min}(c)$ on $l_{true}(c)$ is not known, use $l_{min}(c) = 1$.)
2. Compile the POMDP (Section 3.3.2).
3. Use the extended Baum-Welch method on the POMDP and the given execution trace to determine improved $p_c(l)$ for all corridors c and lengths l with $l_{min}(c) \leq l \leq l_{max}(c)$ (Section 3.3.4.2).
4. For each corridor c: if $\sum_{l \geq l_{max}(c) - Y} p_c(l) \geq P$, then set $l_{max}(c) := l_{max}(c) + Z + 1$.
5. If any $l_{max}(c)$ was changed in Step 4, then go to Step 2, else stop.

Figure 3.35: GROW-BW

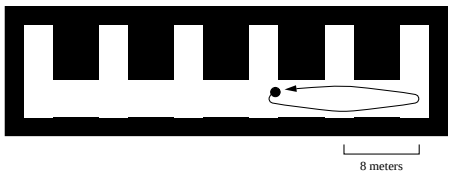


Figure 3.36: An Example of Myopic Effects

POMDP for the given structure, even if the structure is incorrect. This allows GROW-BW to start with a small POMDP, learn the best probabilities for that structure, see if the resulting model is "good enough," and grow the corridor lengths if not: For each corridor, GROW-BW starts with a small interval of possible lengths. It then compiles a POMDP for these lengths and uses the extended Baum-Welch method to assign probabilities to the lengths. If the possible lengths do not include the true length of the corridor, then the extended Baum-Welch method likely places a large amount of probability on the largest possible lengths. Therefore, if this probability mass exceeds a given threshold, GROW-BW increases the range of possible lengths for this corridor and repeats the procedure. Using GROW-BW is advantageous if no upper bounds on the corridor lengths are available (for example, because the distance model is completely unknown), if the given bounds might not include the true corridor lengths, or to avoid having to consider a large number of possible corridor lengths when the bounds are loose ($l_{max}(c) \gg l_{min}(c)$).

In the following, we describe GROW-BW in more detail and explain its parameters X, Y, and Z (all nonnegative integers) and its parameter P (a probability). X determines how many possible lengths to consider initially, Y determines how many of the largest possible lengths to consider when calculating the probability mass, P is the probability threshold that triggers growing the range of possible lengths, and Z specifies how many additional lengths to consider during the next iteration. GROW-BW starts with a small value $l_{max}(c)$, not necessarily an upper bound on $l_{true}(c)$, and grows it if necessary until there is a high probability that $l_{true}(c) \in [l_{min}(c), l_{max}(c)]$ (Figure 3.35). If no bounds on $l_{true}(c)$ are known, GROW-BW initially considers all lengths from one to $X + 2$. It then compiles a POMDP and uses the extended Baum-Welch method to learn the probabilities of that POMDP. Since it is easier for GROW-BW to determine the new probability distribution over the possible lengths if no self-transitions are present, we change the POMDP-compilation component so that it does not model dead-reckoning uncertainty with self-transitions. As a consequence, $l_{true}(c)$ now refers to the perceived length of corridor c, which includes the dead-reckoning error of the robot. If $l_{true}(c)$ is larger than $l_{max}(c)$, the extended Baum-Welch method likely places a high probability mass on the largest lengths considered. GROW-BW therefore adds up the probabilities assigned to the $Y + 1$ largest lengths considered. If the resulting probability mass is at least as large as the given threshold P, GROW-BW increases $l_{max}(c)$ by $Z + 1$. It then adds $Z + 1$ new parallel chains (Figure 3.12(b)) to the corridor and repeats the whole procedure. This way, if the initially chosen range of lengths was too small, it can be increased to include the true length of the corridor. GROW-BW could be modified to not increase $l_{max}(c)$ if the evidence that corridor c was actually traversed is small.

GROW-BW is a hill-climbing method and, thus, can suffer from myopic effects. Consider the most myopic

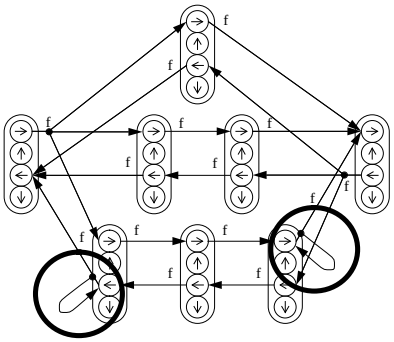


Figure 3.37: A Corridor with Self-Transitions

version of GROW-BW, that uses the parameter values $X = Y = Z = 0$. To simplify our argument, assume that a robot with perfect actuators and almost perfect sensors moves back and forth in the main corridor of the corridor environment shown in Figure 3.36. If $l_{max}(c) = 4$ for all corridors of the main corridor, then the best fitting model is the one where all traversed corridors are four meters long. The robot expects to see corridor openings every four meters, but sees them only every eight meters. For the purpose of the example, we assume that corridors are one meter wide and do not distinguish between small, medium, or large openings. If W denotes a wall and O a corridor opening, then

expected:	OWWWOWWWOWWWOWWWO
observed:	OWWWWWWWOWWWWWWWO

Thus, assuming almost perfect sensors, the model cannot explain four observations on each round-trip, and no distance model whose corridors are at most four meters long can do better. This leads GROW-BW to increase $l_{max}(c)$ to five meters for all traversed corridors. However, at this point the model where all corridors are four meters long is still among the models that, of all models considered, explain the observations best. Another such model is the one where adjacent corridors of the main corridor alternate between lengths three and five. If the extended Baum-Welch method learns the former model, then GROW-BW stops without having learned the true lengths of the corridors.

Notice that we have constructed this example artificially. Despite the theoretical limitations of hill-climbing, our experience with GROW-BW is that it appears to work well in practice. We attribute this to architectural features of buildings – they are usually constructed in ways that prevent people from getting lost, which appears to dampen myopic effects. However, it is possible that a problem similar to the one described could show up in conjunction with doors along a corridor. We therefore recommend to use a less myopic version of GROW-BW by setting the parameters X, Y, and possibly Z to values that are larger than the typical distance between adjacent doors. Similarly, P has to be chosen small enough to prevent GROW-BW from terminating prematurely.

If $l_{true}(c) > l_{max}(c)$, the execution traces can be inconsistent with the POMDP, in the sense that the model cannot explain the experience. One problem this might lead to is that the pose-estimation component of the POMDP-based navigation architecture may rule out all possible poses, leading the robot to become totally uncertain as to where it is. The robot then has to explicitly re-localize itself, which may take a fair amount of time. Another problem is that learning can no longer take place. As an example, consider again the corridor environment shown in Figure 3.36 and assume that the robot traverses the main corridor from beginning to end for a total distance of 40 meters. This, however, is impossible according to a model that assumes that $l_{max}(c) = 4$ for all corridors of the main corridor. We avoid this problem by having the POMDP-compilation component add self-transitions with a small transition probability Q (an additional parameter of GROW-BW) in both directions of the longest chain in the POMDP representation of each corridor (Figure 3.37). This way the probability $p_c(l)$ for every length l with $l \geq l_{min}(c)$ is positive. This does not mean, of course, that

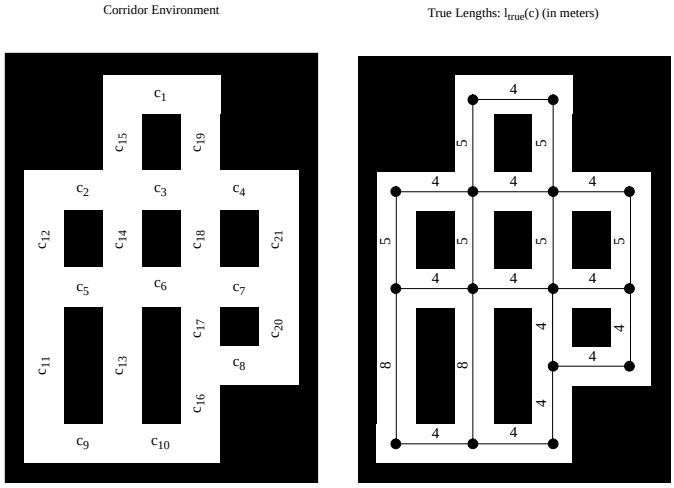


Figure 3.38: Corridor Environment

GROW-BW is no longer needed. Using such a POMDP directly with the extended Baum-Welch method would not work very well if $l_{true}(c) > l_{max}(c)$, because only the probabilities $p_c(l)$ for $l_{min}(c) \leq l < l_{max}(c)$ can be specified individually. The probabilities for $l_{max}(c) \leq l$ are completely determined by these probabilities since they are exponentially decreasing according to the following formula:

$$p_c(l) = \left(1 - \sum_{l_{min}(c) \leq l' < l_{max}(c)} p_c(l')\right) (1 - Q)\ Q^{l - l_{max}(c)}.$$

3.3.4.4 Experiments

We perform all experiments in the Xavier simulator with an environment that has a more complex topology than what we have available in our building, but the learning methods can be used unchanged on Xavier itself, since the only difference is how the execution traces are generated, via Xavier or its simulator. For all experiments, we use the prototypical, but locally ambiguous, corridor environment shown in Figures 3.29 and 3.38. This environment has fifteen topological nodes and twenty-one edges. It has many parallel corridors and indistinguishable corridor junctions, which amplifies the perceptual aliasing problem. Since the POMDP discretizes the possible lengths of corridors with a precision of one meter, the environment matches this assumption: All lengths are multiples of one meter. This way, the true distances can be represented by the learning methods, which allows us to evaluate them better. We use the following obvious equality constraints between the corridor lengths. These constraints are not necessary for the learning methods, but they increase the quality of the learned models if the number of traversals of each corridor is small:

$$\begin{aligned}
&l_{true}(c_1) = l_{true}(c_3) = l_{true}(c_6) = l_{true}(c_{10}) \\
&l_{true}(c_2) = l_{true}(c_5) = l_{true}(c_9) \\
&l_{true}(c_4) = l_{true}(c_7) = l_{true}(c_8) \\
&l_{true}(c_{11}) = l_{true}(c_{13}) \\
&l_{true}(c_{12}) = l_{true}(c_{14}) = l_{true}(c_{18}) = l_{true}(c_{21}) \\
&l_{true}(c_{15}) = l_{true}(c_{19})
\end{aligned}$$

$$l_{true}(c_{17}) = l_{true}(c_{20})$$

The only information that the robot has available is the topological map, the motion and sensor reports, and the equality constraints between the lengths. Given this information, the task of the robot is to refine the structure and probabilities of the POMDP, that is, to learn better actuator, sensor, and distance models. However, we do not inform it about its start pose, its route, or its destination. Instead, we let it gain experience with the environment by guiding it through the corridors. This learning task is a bit harder than learning during actual office navigation, since the robot does not even know its approximate start pose.

The experimental results illustrate that our learning methods are able to acquire good actuator, sensor, and distance models even with only a small execution trace. The resulting models help the robot to determine its pose more accurately and to navigate more efficiently and reliably. This illustrates that the learning methods can learn during office navigation even if they are not sure whether the robot follows the desired path to the goal.

The environment that we use to evaluate the learning methods is relatively small. This makes it possible to use learning methods whose running time is exponential in the length of the execution trace, for example learning methods that match the routes probabilistically against the topological map, possibly combined with branch-and-bound methods to prune the search space. The extended Baum-Welch method and GROW-BW have two advantages over these learning methods: First, the models that they learn (POMDPs) can directly be used by our probabilistic planning and acting methods. Thus, there is no need for a model transformation that might degrade the quality of the learned models. Second (and more importantly), the running time of our learning methods is only polynomial in the length of the execution trace (and the size of the POMDP).

3.3.4.4.1 Experiments with the Extended Baum-Welch Method For the experiments with the extended Baum-Welch method we use a uniform distribution over the lengths from two to nine meters for each corridor, yielding a POMDP with 2,472 states. This allows the extended Baum-Welch method to learn all lengths since they are all between four and eight meters long. Unless stated otherwise, we use the extended Baum-Welch method with a minimal look-ahead of five time steps and a window size of twenty time steps resulting in an average actual look-ahead of twelve time steps for long execution traces. This results in an overhead of calculating each beta value on average 1.3 times (instead of once) and a small amount of bookkeeping for maintaining the time window.

First Experiment: In the first experiment, we simultaneously learn the actuator, sensor, and distance models using an execution trace that consists of ten smaller execution traces with various start poses of the robot, that the robot is not informed about. In total, every corridor is traversed five times. The purpose of this experiment is to learn good actuator and sensor models for use in the subsequent experiments.

Figure 3.39 lists the learned sensor model together with the human estimates that the POMDP-based navigation architecture used originally. Open and closed doors are omitted, since they do not appear in the environment. Notice that the "left" and "right" sensors are reasonably accurate when they issue a report. The relatively high probability of reporting feature unknown is due to the sensors being quite conservative: They do not report features until they have collected sufficient evidence. It is also partly due to the fact that the sensor-interpretation component is implemented as an asynchronous process, and the execution trace fills in an unknown whenever the sensor-interpretation component is not able to issue a report to the pose-estimation component in time. Learning makes this problem less severe by changing the meaning of unknown. Initially, for example, the state distribution remains unchanged when the "left" sensor reports unknown with probability one. After learning, however, states in which there is a corridor opening to the left of Xavier become more likely when "left" sensor reports unknown because the sensor-interpretation component issues this report frequently in these cases.

Second Experiment: The second experiment holds the previously learned actuator and sensor models constant and learns the distance model again (from scratch), this time using an execution trace that is generated by

"left" and "right" sensors	original human estimates			learned probabilities		
	reality			reality		
sensor report	`wall`	`near-wall`	`opening`	`wall`	`near-wall`	`opening`
`wall`	0.90	0.54	0.05	0.784432	0.669705	0.010528
`small-opening`	0.03	0.01	0.20	0.000010	0.000010	0.001762
`medium-opening`	0.02	0.15	0.40	0.000649	0.000010	0.146208
`large-opening`	0.01	0.25	0.30	0.001779	0.000010	0.291644
`unknown`	0.05	0.05	0.05	0.213130	0.330265	0.549859

"front" sensor	original human estimates			learned probabilities		
	reality			reality		
sensor report	`wall`	`near-wall`	`opening`	`wall`	`near-wall`	`opening`
`wall`	0.98	0.50	0.02	0.034271	0.057838	0.000010
`unknown`	0.02	0.50	0.98	0.965729	0.942162	0.999990

Figure 3.39: Original and Learned Sensor Probabilities

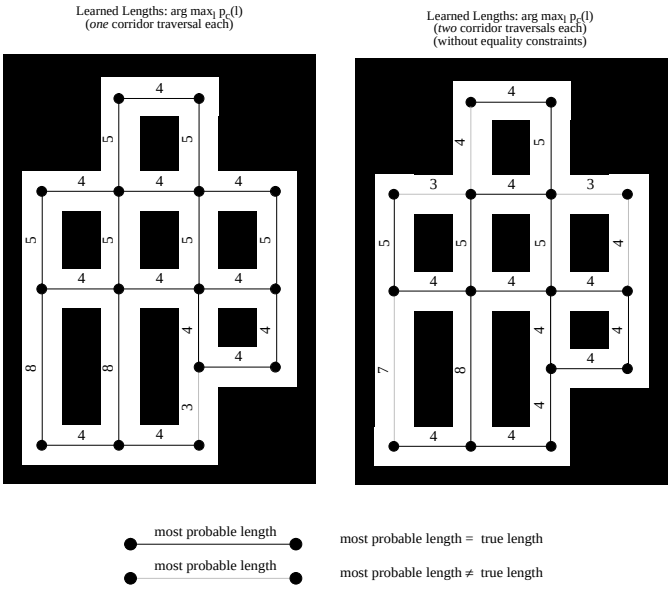


Figure 3.40: The Distances Learned by the Extended Baum-Welch Method

traversing every corridor only once, resulting in two execution traces with different start poses each, that the robot is not informed about (there is no single path that traverses every corridor exactly once). The extended Baum-Welch method needs fifteen iterations to converge and produces the distance model shown in Figure 3.40 (left), that is determined by the most probable length for each corridor. Even with only one traversal per corridor, the learned distance model is very accurate: The extended Baum-Welch method makes only one mistake, where it counts as a mistake if the most probable corridor length is not its true length. In general, it is our experience that the Baum-Welch method can learn good models with about one to three traversals of each corridor, depending on how confusing the environment is. Although the dead-reckoning error of the robot is not overly large, we cannot expect the Baum-Welch method to learn the corridor lengths perfectly: The estimates are actually the *perceived* lengths, and the perception can be distorted by dead-reckoning uncertainty, the discretization granularity used, and how sharply the robot turns within corridor junctions, which may change the distance traveled along the corridors by up to a meter.

While we can compare the learned corridor lengths against the true lengths, we cannot do the same

$$1/T \times (\ln p(o_{1\ldots T}|a_{1\ldots T-1}) + \ln p(\text{can execute } a_{1\ldots T-1}))$$

"how well a model explains (or, synonymously, generates) an execution trace"
(closer to zero is better)

	actuator and sensor models	
distance model	original	learned
original	-3.188194	-1.917637
learned	-2.917535	-1.684340

Figure 3.41: The Quality of the Learned Models: Fit

$$1/(T \ln |S|) \times \sum_{t=1\ldots T} \sum_{s \in S} [\alpha_t(s) \ln(\alpha_t(s))]$$

"how certain the robot is about its pose"
(closer to zero is better)

	actuator and sensor models	
distance model	original	learned
original	-0.101616	-0.050821
learned	-0.049178	-0.044137

Figure 3.42: The Quality of the Learned Models: Entropy

with the actuator and sensor models, since the correct models are unknown. To determine how good the learned models are, we therefore calculate how well they fit a (different) long evaluation execution trace. We use $1/T \times (\ln p(o_{1\ldots T}|a_{1\ldots T-1}) + \ln p(\text{can execute} a_{1\ldots T-1}))$ as a transformation of the fit $p(o_{1\ldots T}|a_{1\ldots T-1})p(\text{can execute } a_{1\ldots T-1})$ that makes it independent of the length of the execution trace. Learning improves the models if the value of this transformation gets closer to zero. Figure 3.41 shows that this is indeed the case both for the learned actuator and sensor models alone, for the learned distance model alone, and for their combination. To determine how much the learned models improve Xavier's on-line pose-estimation capabilities and thus its office-navigation performance, we calculate the average entropy of the alpha values (the pose distributions used during office navigation) after every action execution of the evaluation execution trace, that is, $1/(T \ln |S|) \times \sum_{t=1\ldots T} \sum_{s \in S} [\alpha_t(s) \ln(\alpha_t(s))]$. The *entropy* is a measure for how certain the robot knows its current pose, ranging from zero (absolute certainty at every point in time) to minus one (absolute ignorance, a uniform state distribution). If learning improves the models, we expect the entropy to get closer to zero, resulting in improved on-line pose estimates and thus better office-navigation performance. Figure 3.42 shows that this is indeed the case.

Third Experiment: To test the power of the equality constraints, we repeat the second experiment without them except that we require each corridor to be equally long in both directions. The model learned in this experiment has six corridors where the most probable length is not the correct one. Even if we use an execution trace that is twice as long (starting with the original execution trace), the result does not improve dramatically. Figure 3.40 (right) shows the learned distance model. There are still five corridors where the most probable length is not the correct one. Thus, the distance model is inferior to the distance model learned in the second experiment, despite the longer execution trace. We conclude that equality constraints are an effective means for reducing the length of the execution trace required to learn good models.

3.3.4.4.2 Experiments with GROW-BW For the experiments with GROW-BW we assume that no distance information is available. Again, we guide the robot through every corridor once and do not inform it about its start pose.

First Experiment: Our first experiment is similar to the second experiment in the previous section. It uses the extended Baum-Welch method directly. To make sure that it is able to learn the true lengths of the corridors,

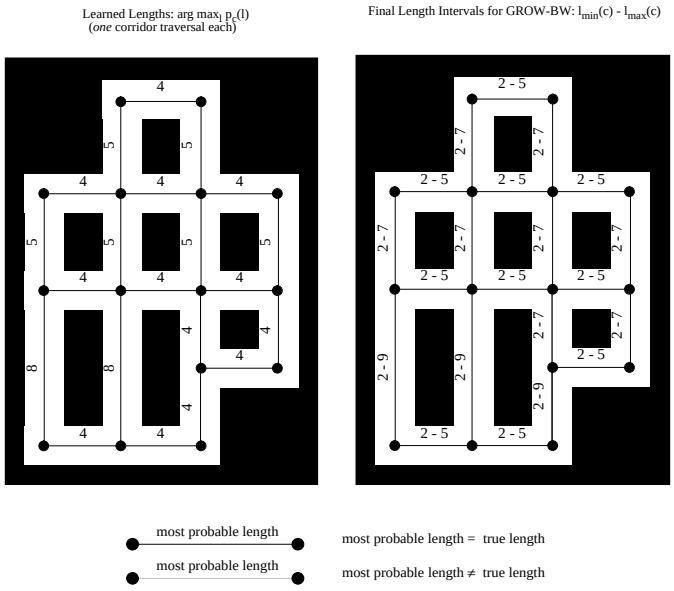


Figure 3.43: The Distances Learned by GROW-BW

we estimate their minimal and maximal lengths cautiously to guarantee that $l_{true}(c) \in [l_{min}(c), l_{max}(c)]$. We use $l_{min}(c) = 2$ meters and $l_{max}(c) = 14$ meters for every corridor c. The resulting POMDP has 6,672 states and 80,346 state transitions (action outcomes). Figure 3.43 (left) depicts the lengths with the largest probability $p_c(l)$ in the learned model: All 21 predicted lengths correspond to the true lengths.

Second Experiment: Our second experiment uses GROW-BW with the parameters $X = 0$, $Y = 0$, $Z = 1$, $P = Q = 0.05$, and $l_{min}(c) = 2$ meters for all corridors. That is, the initial estimate for every corridor is $l_{max}(c) = 3$ meters and, if GROW-BW extends a length, it increases it by two meters. GROW-BW assumes a uniform probability distribution over the possible lengths. Given this information, GROW-BW needs only four iterations to converge. Figure 3.43 (right) shows $l_{min}(c)$ and $l_{max}(c)$ for each corridor c in the final model. The corresponding POMDP has only 1,176 states and 16,260 state transitions, and is thus much smaller than the POMDP from our first experiment. This results in GROW-BW being 1.84 times faster than the extended Baum-Welch method to learn the same model, although it has to call the extended Baum-Welch method repeatedly. The probabilities $p_c(l)$ that GROW-BW learns are similar to those learned in the first experiment, and the lengths with the largest probability $p_c(l)$ are even identical: Again, all lengths are learned correctly.

We repeat both experiments eight more times with different robot routes. The results are summarized in Figure 3.44. A mistake in the column "corridors" indicates that, after learning, the length with the largest probability among all lengths in $[l_{min}(c), l_{max}(c)]$ is not the correct one. If a group of corridor lengths are constrained to be equal, then a mistake counts multiple times in the column "corridors." The column "groups" is identical to the column "corridors" except that we count only one mistake per group. In general, the more ambiguous the routes are (the harder it is to match them against the topological map), the more mistakes the learning methods make. In all experiments, both the extended Baum-Welch method and GROW-BW learn good (although not perfect) distance models with only one traversal of each corridor and err by only one meter when they make a mistake. The predicted corridor lengths are always identical.

The experiments show that the sizes of the POMDPs produced by GROW-BW are roughly between four and six times smaller than the size of the POMDP that we used in conjunction with the extended Baum-Welch method. As a result, GROW-BW is almost two times faster than the extended Baum-Welch method.[3] Thus,

[3]The seventh experiment in Figure 3.44 is an exception. It contains a highly ambiguous execution trace and GROW-BW expands the

	mistakes of ext. Baum-Welch		mistakes of GROW-BW		improvement in the number of states	improvement in running time
	corridors (out of 21)	groups (out of 8)	corridors (out of 21)	groups (out of 8)		
1	0	0	0	0	4.79×	1.80×
2	0	0	0	0	4.46×	1.76×
3	0	0	0	0	5.67×	1.67×
4	5	2	5	2	5.67×	2.08×
5	5	2	5	2	5.20×	1.99×
6	0	0	0	0	5.20×	1.97×
7	3	2	3	2	3.66×	0.53×
8	0	0	0	0	5.20×	1.78×

Figure 3.44: A Comparison of GROW-BW and the Extended Baum-Welch Method

Figure 3.45: A Long Corridor with Doors

GROW-BW produces results similar to those of the extended Baum-Welch method, but works on much smaller POMDPs and therefore needs less memory space and often less running time. We could also augment GROW-BW with a post-processing step that prunes the final POMDP, thus making it even smaller.

3.3.4.4.3 Further Experiments The extended Baum-Welch method has been used successfully to learn in more complex environments than the one used here. Consider, for example, the slightly more difficult corridor environment shown in Figure 3.45. The robot traverses a long corridor and has to learn the distances between adjacent doors and corridors. We let it traverse parts of the corridor seven times such that it passes each topological node a total of about three times. However, we do not inform the robot of its start poses, the status of the doors (all of which are open), or any distance constraints except that we require each corridor to be equally long in both directions. The initial distance uncertainty from one landmark to the next is given as a uniform distribution over the lengths from one to ten meters. Notice that none of the traveled routes contain turn actions. Thus, each of them can be matched in numerous ways against the topological map. Furthermore, doors are hard to detect: The robot cannot detect closed doors, misses open doors 22 percent of the time, and can confuse them with corridor openings. Nevertheless, the extended Baum-Welch method, using a minimal look-ahead of twenty time steps, learns all distances but two correctly: It places the corridor marked X one meter to the left of its correct position.

3.4 Related Work

There are several other approaches that use Markov models for office navigation: Dean *et al.* [Dean *et al.*, 1993] use Markov models, but, different from our approach, assume that the location of the robot is always known precisely. Nourbakhsh *et al.* [Nourbakhsh *et al.*, 1995] use models that are similar to Markov models. They do not assume that the location of the robot is known with certainty, but do not utilize any distance information.

upper bound of one corridor to a length of 21(!) meters, which requires ten iterations. We could not replicate this phenomenon when we used execution traces that traversed each corridor more than once.

The states of the robot are either at a topological node or somewhere in a connecting corridor. Burgard *et al.* [Burgard *et al.*, 1996] (University of Bonn) also use Markov models that do not assume that the location of the robot is known with certainty, but derive the Markov models from a very fine-grained tessellation of metric space. Cassandra *et al.* [Cassandra *et al.*, 1996] (Brown University) use Markov models similar to ours, but assume that the distances are known with certainty. Other researchers who are experimenting with POMDP-based navigation architectures similar to ours include Asoh *et al.* (Electrotechnical Laboratory, Ministry of International Trade and Industry, Japan), Mahadevan (Michigan State University), Murphy (Colorado School of Mines), and Gutierrez-Osuna (North Carolina State University).

The extended Baum-Welch method and GROW-BW, our learning methods, learn quantitative information that is difficult to obtain from humans (actuator, sensor, and distance models), but are able to utilize a large variety of qualitative and quantitative information that humans can easily provide. Since then, Shatkay and Kaelbling [Shatkay and Kaelbling, 1997] also applied the Baum-Welch method to map learning. In contrast to our approach, many traditional map learning approaches attempt to learn maps from scratch, not utilizing prior knowledge that is easily available. Furthermore, GROW-BW is a passive learning method, while many traditional map learning methods are active learning methods. In particular, most other approaches in the literature use active exploration to learn either metric or topological maps from scratch (sometimes assuming perfect actuators or sensors) with the goal to either map the environment completely or reach a given goal pose. Approaches whose properties have been analyzed formally, for example, include [Iyengar *et al.*, 1986, Oommen *et al.*, 1987, Rao *et al.*, 1991, Lumelsky *et al.*, 1990, Zelinsky, 1992]. Approaches that have been illustrated experimentally include [Kuipers and Levitt, 1988, Mataric, 1990, Thrun, 1993]. The approaches in [Basye *et al.*, 1989] and [Dean *et al.*, 1992] learn Markov models of the environment, as we do, but they use active exploration, while our approach is passive. The learning approach in [Engelson and McDermott, 1992] uses a passive learning approach, but it learns a topological map only. These approaches also differ from our learning method in that they learn their models from scratch.

3.5 Future Work

In this chapter, we reported first experiments with the POMDP-based navigation architecture. It is important that more experience be gained with this architecture, both on different robots and in different environments. This will happen automatically since other researchers are working with the POMDP-based navigation architecture already. Also, the architecture has already been used to conduct Ph.D. level work by others, for example in [Haigh, 1995] and [Goodwin, 1997].

Currently, the weakest spots of the architecture are the POMDP planning methods. We have not spent much time on them since the greedy POMDP planning methods work well in our office environment and more sophisticated POMDP planning methods are currently being investigated by other researchers. One can construct environments that confuse the greedy POMDP planning methods, for example, by providing no initial knowledge about the starting pose of the robot. Therefore, we would like to use slightly more computationally intensive POMDP planning methods that produce solutions of better quality, and thus provide a trade-off between running time and solution quality that mediates between optimal and greedy POMDP planning methods. While this could be done without a specific application in mind, it is probably best done in the context of the office-navigation task, since the office-navigation task defines which solutions are of acceptable quality and allows POMDP planning methods to exploit the structure of the POMDP to obtain these solutions efficiently. However, any novel approximate POMDP planning methods, whether developed by us or other researchers, can be used unchanged in the context of the office-navigation task.

The learning methods (the extended Baum-Welch method and GROW-BW) could be extended as well. We have assumed that they can be provided with a correct topological map. Although this is a realistic assumption for many robot learning scenarios, weakening it broadens the application area of our methods. We thus would like to extend our learning methods to be able to detect and correct inaccurate prior knowledge. For example, we want them to be able to correct slightly incorrect topological maps and, if necessary, learn them from

scratch. For the latter task it should be possible to combine them with the passive topological map learning method in [Engelson and McDermott, 1992] to extend the applicability of our learning methods to scenarios where a topological map is not available at all. It would also be interesting to investigate GROW-BW in the context of applications other than robot navigation.

3.6 Conclusions

This chapter presented a navigation architecture that uses partially observable Markov decision process models (POMDPs) for autonomous office navigation. POMDPs have been studied in operations research and provide a uniform, theoretically grounded framework for acting, planning, and learning. While we cannot use existing POMDP methods unchanged, we can build on previous work. Our POMDP-based navigation architecture provides for reliable and efficient navigation in office environments, even in the presence of noisy actuators and sensors as well as distance uncertainty.

The POMDP-based navigation architecture maintains a probability distribution over all poses, rather than a single estimate for the current robot pose. This way, the robot always has some idea as to what its current pose is. The POMDP-based navigation architecture uses probabilities to model actuator, sensor, and distance uncertainty explicitly, and updates the pose distribution using both motion reports and sensor reports about landmarks in the environment. Thus, it is robust towards sensor uncertainty without explicit exception handling. Different from Kalman filters, the POMDP-based navigation architecture discretizes the poses, which allows it to represent arbitrary probability distributions over them.

The POMDP-based navigation architecture uses POMDP planning methods for planning and acting. Our POMDP planning method neglects state uncertainty during planning and then accounts for it greedily during navigation. It achieves fast and reactive robot control at average travel speeds of around 50 centimeters per second and is robust, as illustrated by experiments that required the robot to navigate about 150 kilometers in total.

The POMDP-based navigation architecture uses the GROW-BW method in conjunction with the extended Baum-Welch method for unsupervised, passive learning. The extended Baum-Welch method decreases the memory requirements of the Baum-Welch method by using a sliding time window on the training data. It decreases the training-data requirements of the Baum-Welch method by imposing equality constraints between probabilities and using Bayesian probability estimates instead of maximum likelihood estimates. The GROW-BW method uses the extended Baum-Welch method as a subroutine. It is a hill-climbing method that is able to change the structure of the POMDP. It can simultaneously learn accurate actuator, sensor, and distance models that result in an improved office-navigation performance. The quality of the models improves as the length of the execution trace increases, and the training data are generated in the course of normal office navigation, without requiring a teacher or control of the robot.

We believe that probabilistic navigation techniques such as the POMDP-based navigation architecture hold great promise for getting robots reliable enough to operate unattended for long periods of time in complex and uncertain office environments. Applying POMDPs to office navigation also opens up new application areas for more theoretical results in the area of planning and learning with Markov models.

Chapter 4

Acting with Nonlinear Utility Functions

Planning methods have to decide which of the plans that solve a given planning task they consider best. In other words, they have to measure the quality of plans. For example, if they just choose any plan that solves the planning task, they implicitly assume that all solutions are equally preferable. Often, planning methods have more sophisticated preference models. For example, the planning method studied in Chapter 2 (Min-Max LRTA*) attempted to find plans that achieve the goal with minimal worst-case plan-execution cost and the POMDP planning methods studied in Chapter 3 attempted to find plans that achieve the goal with minimal average plan-execution cost. In this chapter, we extend these preference models.

The preference model of most probabilistic search and planning methods from artificial intelligence is to find plans with maximal probability of goal achievement or, if the goal can be achieved for sure, find plans that achieve the goal with minimal average plan-execution cost. The preference model of planning methods from robotics is often to find plans that achieve the goal with minimal worst-case plan-execution cost. Utility theory, on the other hand, suggests a more general preference model, namely, to find plans that achieve the goal with maximal average utility, where the utility is a strictly monotonically decreasing function of the cost. Maximizing average utility and minimizing average cost result in the same decisions if either the domain is deterministic or the utility function is linear. However, many domains are probabilistic. Nonlinear utility functions are then needed for planning with deadlines and planning with risk attitudes in high-stake one-shot planning domains. For example, when people have to decide whether they would like to get 4,500,000 dollars for sure or get 10,000,000 dollars with fifty percent probability, many people prefer the safe alternative although its average pay-off is clearly lower – they are risk-averse. Utility theory explains this as follows: If a person has a non-linear utility function that associates utility (here: pleasure) 0.00 with a wealth of 0 dollars, utility 0.74 with a wealth of 4,500,000 dollars, and utility 0.95 with a wealth of 10,000,000 dollars, then the average utility of getting 4,500,000 dollars for sure is 0.74, whereas the average utility of getting 10,000,000 dollars with fifty percent probability is only 0.48. Thus, the safe alternative maximizes the average utility for this person. Similarly, the utility of a delivery can be a non-linear function of the delivery time.

We study how to find plans efficiently that achieve the goal with maximal average utility for nonlinear utility functions in probabilistic domains. We focus on one particular class of nonlinear utility functions, namely, exponential utility functions, for the following reason: Planners that find plans that achieve the goal with minimal average cost often decompose planning tasks into subtasks that can be solved independently. The solutions can then be assembled into an overall solution of the planning task. This makes planning efficient. Unfortunately, planning tasks that are decomposable with respect to average cost are not necessarily decomposable with respect to average utility for nonlinear utility functions. Exponential utility functions are nonlinear utility functions that preserve the decomposability of planning tasks. This allows us to generalize the preference model of many search and planning methods from artificial intelligence without giving up efficiency. Exponential utility functions are well suited for expressing immediate soft deadlines and expressing a continuum of risk attitudes in high-stake one-shot planning domains, including risk-seeking behavior (such as

gambling) and risk-averse behavior (such as holding insurance). They can also trade-off between minimizing the worst-case, the average, and the best-case plan-execution cost.

We develop efficient methods that transform planning tasks with exponential utility functions to planning tasks that many search and planning methods from artificial intelligence can solve, including those that do not reason about plan-execution costs at all. The transformations are simple context-insensitive representation changes that can be performed locally on various representations of planning tasks. The additive planning-task transformation applies to planning tasks that can be solved with sequential plans (more precisely: planning tasks whose actions have deterministic outcomes but whose rewards can be nondeterministic) and transforms them to deterministic planning tasks. The original planning task can then be solved by finding a plan that achieves the goal with minimal plan-execution cost for the transformed planning task. The multiplicative planning-task transformation applies to planning tasks that can be solved with conditional plans and transforms them to probabilistic planning tasks. The original planning task can then be solved by finding a plan with maximal probability of goal achievement or a plan that achieves a goal with minimal average plan-execution cost for the transformed planning task. We illustrate the transformations using path planning for goal-directed navigation tasks in the presence of blockages.

To summarize, we study how to act in order to maximize average utility for exponential utility functions, including acting in the presence of deadlines and in the presence of risk attitudes in high-stake one-shot planning domains. Our main contribution is the following: First, we apply exponential utility functions from utility theory to robot navigation and other planning tasks from artificial intelligence. Second, we extend the range of planning methods that find plans with maximal average utility for exponential utility functions from dynamic programming methods from operations research to search and planning methods from artificial intelligence, using transformations that we call additive and multiplicative planning-task transformations. Other contributions are the following: We show how exponential utility functions can unify planning methods that minimize worst-case, average, and best-case plan-execution cost. We also show how studying exponential utility functions can provide an interesting interpretation for discounting.

We proceed as follows: Section 4.1 contrasts maximizing average utility with the preference models typically used in artificial intelligence. Section 4.2 motivates the importance of nonlinear utility functions for planning with deadlines and planning with risk attitudes in high-stake one-shot planning domains. Section 4.3 investigates the problem encountered when maximizing average utility for nonlinear utility functions in probabilistic domains, namely, the loss of decomposability. Section 4.4 introduces exponential utility functions, shows how they solve this problem, and illustrates that they are expressive for modeling immediate soft deadlines and modeling risk attitudes in high-stake one-shot planning domains. Section 4.5 describes our planning methods for finding plans that achieve the goal with maximal average utility for exponential utility functions: Section 4.5.1 discusses the additive planning-task transformation and Section 4.5.2 discusses the multiplicative planning-task transformation. Section 4.6 then applies the multiplicative planning-task transformation to Markov decision process models. Sections 4.5 and 4.6 also discuss advantages and limitations of our planning methods. Finally, Section 4.7 describes possible extensions of our planning methods, and Section 4.8 summarizes our conclusions.

4.1 Traditional Approaches

Plans in probabilistic domains can have more than one chronicle, where a *chronicle* is a specification of the state of the world over time, representing one possible course of execution of the plan.

We partition the state space into goal states and nongoal states. If the agent stops in a goal state, it has solved the planning task or, synonymously, achieved the goal, otherwise it has not achieved the goal. A plan achieves the goal if all of its chronicles achieve the goal, otherwise the plan does not achieve the goal.

Although the agent cannot determine apriori which chronicle results from the execution of the plan, it is often (but not always, see Chapter 2) realistic to assume that it is able to determine the probabilities with which the chronicles occur and the amount of resources consumed during their execution. We consider planning tasks with exactly one limited resource (such as time, energy, or money). The cost of a chronicle then is the amount

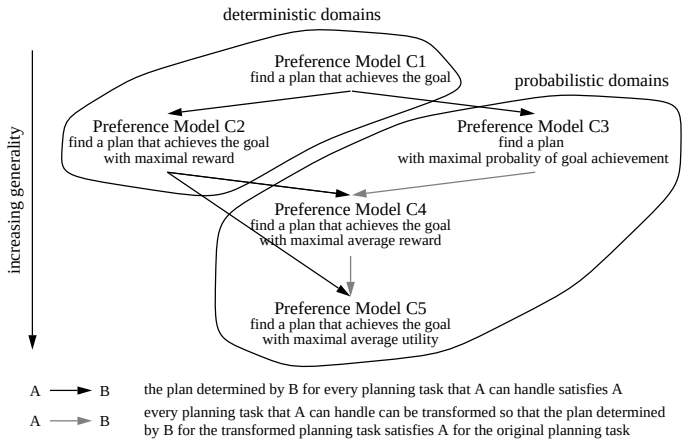


Figure 4.1: Preference Models for Planning Tasks

of that resource consumed. To be consistent with the terminology of the previous chapters, we model the cost as a negative reward. Thus, larger rewards (smaller amounts of the resource consumed) are preferable over smaller rewards (larger amounts of the resource consumed).

The probability and reward of a chronicle can be calculated from the action executions and their outcomes. We assume for now that the number of action executions is bounded for every chronicle of every possible plan and that cyclic plans do not exist. Thus, the agent stops after a bounded number of action executions, either in a goal state or a nongoal state. (We drop this assumption in Section 4.6.) We further assume that the rewards of action executions and the probabilities of their outcomes are known and do not change over time and that the rewards are bounded. The probability of a plan chronicle is calculated as the product of the probabilities of its action outcomes. This assumes probabilistic independence. The reward of the chronicle is calculated as the sum of the rewards of its action executions. This assumes an additive value function (which is, for example, a reasonable assumption for resource consumptions). Since a chronicle has a bounded number of action executions and each action execution results in a bounded reward, the reward of a chronicle is bounded as well.

Given these assumptions, not all planning methods share the same planning objective since they use different preference models to compare plans. In the following, we discuss common preference models and relate them to the preference model studied in this chapter.

Preference Model C1: Traditionally, planners have been used in deterministic domains with the objective to find plans that achieve the goal. Thus, they prefer plans that achieve the goal over ones that do not achieve the goal. This preference model is binary since a plan either achieves the goal or does not achieve the goal.

Preference Model C2: To make the preference model richer, planners then began to associate rewards (execution costs) with plans and preferred plans that achieve the goal with maximal reward (minimal cost). We are interested in generalizing these preference models to probabilistic domains, where the success and reward of a plan can vary from plan execution to plan execution. Several generalizations are possible.

Preference Model C3: When the goal cannot be achieved for sure, or it takes too long to find plans that achieve the goal, plans are often preferred that maximize the probability of goal achievement. In particular, if the execution of plan ps leads with probability p_i to chronicle i and the chronicles $i \in I$ achieve the goal, then its *probability of goal achievement* is $\sum_{i \in I} p_i$. Planning methods that maximize or approximately maximize the probability of goal achievement include [Bresina and Drummond, 1990, Blythe, 1994, Goldman and Boddy, 1994, Draper *et al.*, 1994, Kushmerick *et al.*, 1995].

Preference Model C4: When the goal can be achieved for sure, plans are often preferred that achieve the goal with maximal average reward. In particular, if the execution of plan ps leads with probability p_i to a

chronicle with reward r_i, then its *average reward* is $er(ps) := \sum_i [p_i r_i]$ (er = expected reward). Even if the goal cannot be achieved for sure, preference model C4 is often used by making every state a goal state and then introducing a reward (measured in units of the resource consumption) for stopping in the original goal states. The amount of this reward has to be determined empirically. The larger it is, the more important it becomes to stop in the original goal states. Planning methods that maximize or approximately maximize average reward include [Smith, 1988, Etzioni, 1991, Russell and Wefald, 1991, Goodwin and Simmons, 1992, Boutilier and Dearden, 1994, Boutilier *et al.*, 1995b] and many others.

Preference Model C5: In this chapter, we prefer plans that achieve the goal with maximal average utility, where the utility is a strictly monotonically increasing function that maps rewards r to the resulting real-valued utility $u(r)$. In particular, if the execution of plan ps leads with probability p_i to a chronicle with reward r_i, then its *average utility* is $eu(ps) := \sum_i [p_i u(r_i)]$ (eu = expected utility). Notice that, different from Wellman and Doyle [Wellman and Doyle, 1991], preference model C5 does not use utilities to explain what goals are. Instead, it combines goals with utilities. This makes it possible to plan efficiently by using goals to direct the search effort. In the remainder of this chapter, we always implicitly assume that utility functions are strictly monotonically increasing in the reward.

In the following, we discuss the relationships among the various preference models, to show that preference model C5 subsumes the other ones. Figure 4.1 summarizes the discussed relationships.

The following relationships are obvious: **Preference Model C2 versus C1:** Preference model C2 generalizes preference model C1, since it continues to prefer plans that achieve the goal over plans that do not achieve the goal. **Preference Model C3 versus C1:** Preference model C3 generalizes preference model C1, since it continues to prefer plans in deterministic domains that achieve the goal over ones that do not achieve the goal. **Preference Model C4 versus C2:** Preference model C4 generalizes preference model C2, since it continues to prefer plans in deterministic domains that achieve the goal with maximal reward. **Preference Model C5 versus C2:** Preference Model C5 generalizes preference model C2, since it continues to prefer plans in deterministic domains that achieve the goal with maximal reward. This is so since the utility function is strictly monotonically increasing and thus a larger utility implies a larger reward and vice versa.

Preference Model C4 versus C3: Preference model C4 generalizes preference model C3 for a particular reward structure. We transform the original planning task (with preference model C3) by making every state a goal state and introducing rewards. The reward for stopping in an original goal state is $r > 0$ (the precise value does not matter) and there are no other rewards. If a plan for the original planning task achieves a goal state with probability p, then its transformation achieves a goal state with average reward $p \times r + (1-p) \times 0$ which is proportional to p. Thus, a larger probability of goal achievement always implies a larger average reward, and vice versa. Consequently, a planner with preference model C4 determines a plan for the transformed planning task that maximizes the probability of goal achievement for the original planning task. This generalizes preference model C3, since it continues to prefer plans that maximize the probability of goal achievement.

Preference Model C5 versus C4: Preference model C5 generalizes preference model C4 for particular utility functions, namely, all linear utility functions $u(r) = m \times r + n$ with $m > 0$. Then, the average utility of plan ps is $eu(ps) = m \times er(ps) + n$ which is proportional to $er(ps)$. Thus, a planner with preference model C5 determines a plan that achieves the goal with maximal average reward. This generalizes preference model C4, since it continues to prefer plans that achieve the goal with maximal average reward.

In the following, we show that preference model C5 is more general than preference model C4 because, for nonlinear utility functions in probabilistic domains, a plan that achieves the goal with maximal average utility does not necessarily maximize the average reward, and vice versa. As an example, we use path planning for the goal-directed navigation tasks from the previous chapter and assume that the only limited resource is the travel time of the robot. In this example, the planning uncertainty results from external events, namely, people opening and closing doors. Goodwin [Goodwin, 1997] uses this example in the context of preference model C4.

Example 1: Consider the simple navigation task shown in Figure 4.2. The robot drives at speed v meters per second and can take either Path 1 or Path 2, both of which solve the navigation task. If the robot takes

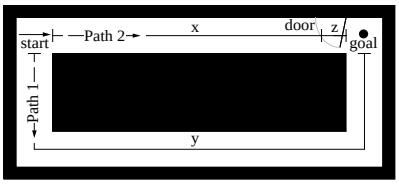


Figure 4.2: Navigation Example

Path 1, it reaches the goal location in y/v seconds for sure. Thus, it receives reward $r(path_1) = -y/v$ with probability $p(path_1) = 1$. Path 2 leads through a door that the robot is not able to open. The robot does not know whether the door is open or closed, but it knows that the door is usually open with probability $p_1(path_2)$ and closed with probability $p_2(path_2) = 1 - p_1(path_2)$ and that it can find out about the state of the door by using its sonar sensors, which requires it to move close to the door. If the robot takes Path 2, then the door is open with probability $p_1(path_2)$ and the robot reaches the goal location in $(x + z)/v$ seconds and thus receives reward $r_1(path_2) = -(x + z)/v$. With the complementary probability $p_2(path_2)$ the door is closed and the robot has to return to its starting position and take Path 1 to the goal. In this case it reaches the goal location in $(2x + y)/v$ seconds and thus receives reward $r_2(path_2) = -(2x + y)/v$. To make the example more concrete, assume that $p_1(path_2) = p_2(path_2) = 0.50$, $x = 29$ meters, $y = 86$ meters, $z = 1$ meter, and $v = 0.25$ meters per second. Then, Path 1 reaches the goal in 344.00 seconds with probability 1, and Path 2 reaches the goal in 120.00 seconds with probability 0.50 and in 576.00 seconds with probability 0.50.

Preference model C4 decides between Path 1 and Path 2 as follows: The (average) reward of Path 1 is -344.00 seconds and the average reward of Path 2 is $0.50 \times (-120.00) + 0.50 \times (-576.00) = -348.00$ seconds. Consequently, preference model C4 chooses Path 1 over Path 2.

Preference model C5 decides between Path 1 and Path 2 as follows: Assume that the utility function is $u(r) = (2^{1/300})^r$, where r is the negative travel time of the robot, measured in seconds (the negative travel time of Path 1 is, for example, -344.00 seconds). We justify this particular utility function in Section 4.2.1. The average utility of Path 1 from Example 1 is $u(-344.00) = 0.45$. The average utility of Path 2 is $0.50 \times u(-120.00) + 0.50 \times u(-576.00) = 0.51$. Thus, preference model C5 chooses Path 2 over Path 1, contrary to most path planners and other search and planning methods from artificial intelligence.

Instead of using preference model C5 directly, we often use an equivalent preference model. In the following, we discuss this preference model and why it sometimes has advantages.

Preference Model C5 (alternative version): We prefer plans that achieve the goal with maximal certainty equivalent, where the *certainty equivalent* of plan ps is $ce(ps) := u^{-1}(eu(ps))$ (ce = certainty equivalent). Again, the utility is a strictly monotonically increasing function that maps rewards r to the resulting real-valued utility $u(r)$ and thus its inverse (denoted by a superscript of minus one) is a strictly monotonically increasing function that maps utilities to their corresponding rewards. An agent with preference model C5 is indifferent between a plan and a deterministic plan (that is, obtaining a certain reward for sure) if and only if the reward of the deterministic plan is the same as the certainty equivalent of the other plan. This explains the name of this concept from utility theory. (Notice that the certainty equivalent of deterministic plans is their (average) reward.)

Since the utility function is strictly monotonically increasing, a larger certainty equivalent always implies a larger average utility, and vice versa. Thus, a planner that achieves the goal with maximal certainty equivalent determines a plan that achieves the goal with maximal average utility, and vice versa. The advantage of certainty equivalents over average utilities is that differences in certainty equivalents are meaningful whereas differences in utility are meaningless. We explain this in the following.

Utilities are defined only up to positively linear transformations, where a positively linear transformation of a function f is a function g such that there exist real constants m and n with $m > 0$ and $g(x) = m \times f(x) + n$ for all x. Consequently, all utility functions $m \times u(r) + n$ with $m > 0$ are equivalent. The difference in utility

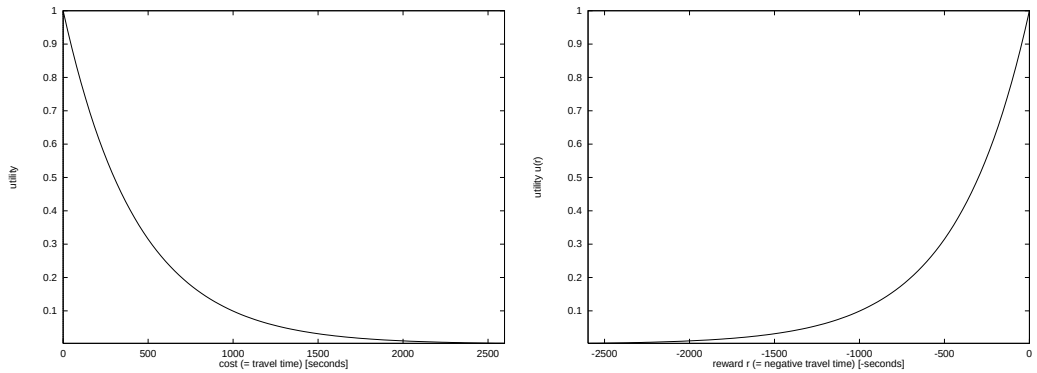


Figure 4.3: Immediate Soft Deadline (left) and Corresponding Utility Function (right)

between two plans ps and ps' is $m \times (eu(ps) - eu(ps'))$ for $m > 0$ and thus depends on m. This means that utility differences are meaningless in the sense that their magnitudes are not correlated with how much better one plan is over another. On the other hand, the certainty equivalent of any plan is the same for all equivalent utility functions. Therefore, differences between certainty equivalents are meaningful and easy to interpret since certainty equivalents are measured in the same unit as the rewards. Thus, if we want to compare the quality of two plans, we have to use certainty equivalents instead of utilities. For example, the inverse function of $u(r) = (2^{1/300})^r$ is $u^{-1}(r) = 300 \log_2 r$. Thus, the certainty equivalent of Path 1 from Example 1 is $u^{-1}(0.45) = -344.00$ seconds. The certainty equivalent of Path 2 is $u^{-1}(0.51) = -290.53$ seconds. Thus, choosing Path 2 over Path 1 leads to a difference in certainty equivalents of $(-290.53) - (-344.00) = 53.47$ seconds.

To summarize, we have discussed several preference models for planning and shown that maximizing average utility generalizes the other ones. Unfortunately, utility theory is a purely descriptive theory that specifies only what optimal plans are, but not how they can be obtained other than by enumerating every chronicle of every possible plan. Operations research has picked up on the results from utility theory and uses dynamic programming methods to find plans with maximal average utility [Marcus *et al.*, 1997]. For example, an early method in the context of Markov decision process models is [Howard and Matheson, 1972] and an early method in the context of linear stochastic systems is [Jacobson, 1973]. These methods do not utilize available domain knowledge. Artificial intelligence has investigated knowledge-based search and planning methods that scale up to larger domains, but traditionally not for finding plans that achieve the goal with maximal average utility. In the remainder of this chapter, we first argue why finding plans that maximize average utility for nonlinear utility functions is important and then study how to solve planning tasks efficiently that involve such utility functions.

4.2 Nonlinear Utility Functions

In the following, we give two examples of utility functions that are strictly monotonically increasing but nonlinear in the rewards, namely, utility functions for acting in the presence of immediate soft deadlines and in high-stake one-shot planning domains. In both cases, we argue that it is better to maximize the average utility rather than the average reward. We also give a short overview of how utility theory uses nonlinear utility functions to model risk attitudes for planning tasks in high-stake one-shot planning domains.

4.2.1 Immediate Soft Deadlines

Immediate deadlines coincide with the time at which the execution of the plan begins. *Soft deadlines* are those whose utility does not drop to zero immediately after the deadline has passed but rather declines slowly. We use path-planning examples to illustrate how to model immediate soft deadlines with nonlinear utility functions.

As an example of immediate soft deadlines, consider indoor delivery tasks. Often, delivery requests are not made in advance, the goods are needed right away, and the utility of the deliveries declines over time. These situations can be modeled with immediate soft deadlines. An example is the delivery of emergency medicine with hospital delivery robots to patients that have unexpectedly developed problems that deteriorate their health until they take the medicine.

Another example of immediate soft deadlines is the delivery of printouts on demand. Imagine, for instance, that you are debugging a program on your computer. To get a better overview of the program, you print it out and send your office delivery robot to fetch the printout from the remote printer room. In this case, the printout is needed right away, but you do not need it any longer if you find the problem with the program before the printout has been delivered to you. Thus, the utility of the delivery is highest if it could be done in no time. It decreases over time because the probability increases that you find the problem before the time of delivery. Thus, the printout delivery task is one with an immediate soft deadline. Its utility function is nonlinear in the plan-execution time: The preference model is to maximize the expectation of the probability that the problem has not been found before the time of delivery. Thus, the utility is this probability. If the probability to find the problem in any period of time Δt is p, then the probability that it has not been found after n such time periods (assuming probabilistic independence) is $(1-p)^n$. Thus, the utility function is exponentially increasing in the negative plan-execution time. An example of such a utility function is shown in Figure 4.3 (right). It is the exponential utility function $u(r) = (2^{1/300})^r$, where r is the negative travel time of the robot, measured in seconds. Thus, the utility of the delivery halves every five minutes.

A third example of immediate soft deadlines arises for navigation with outdoor rovers, that can be damaged during navigation to their destination. The preference model is to maximize the expectation of the survival probability. Thus, the utility is this probability. The utility function is nonlinear in the plan-execution time: If the probability of damage in any period of time Δt is p, then the probability of surviving n such time periods (assuming probabilistic independence) is $(1-p)^n$. Thus, the utility function is again exponentially increasing in the negative plan-execution time.

4.2.2 Risk Attitudes in High-Stake One-Shot Planning Domains

Maximizing average reward is reasonable if the execution of a plan is repeated a large number of times since the variance of the total reward (summed over all plan executions) approaches zero according to the law of large numbers. However, planning systems are often used to make decisions in high-stake one-shot planning domains. *High-stake domains* are domains in which huge wins or losses are possible. In high-stake one-shot planning domains, many people do not maximize average reward. We call them *risk-sensitive agents* to differentiate them from *risk-neutral agents* that maximize average reward. For example, *risk-seeking agents* (gamblers) are risk-sensitive agents that focus on the desirable outcomes, the highest possible rewards. They hope to be able to do much better than average. *Risk-averse agents* (insurance holders) are risk-sensitive agents that focus on the undesirable outcomes. They are afraid to do much worse than average. Consider, for example, that you could participate once in one (and only one) of the following two free lotteries.

Lottery	Probability	Reward
Lottery 1	50 percent	10,000,000 dollars
	50 percent	0 dollars
Lottery 2	100 percent	4,500,000 dollars

Many people are risk-averse and prefer Lottery 2, although the average reward (pay-off) of Lottery 1 is slightly larger. These people would be very disappointed fifty percent of the time if an automated planning

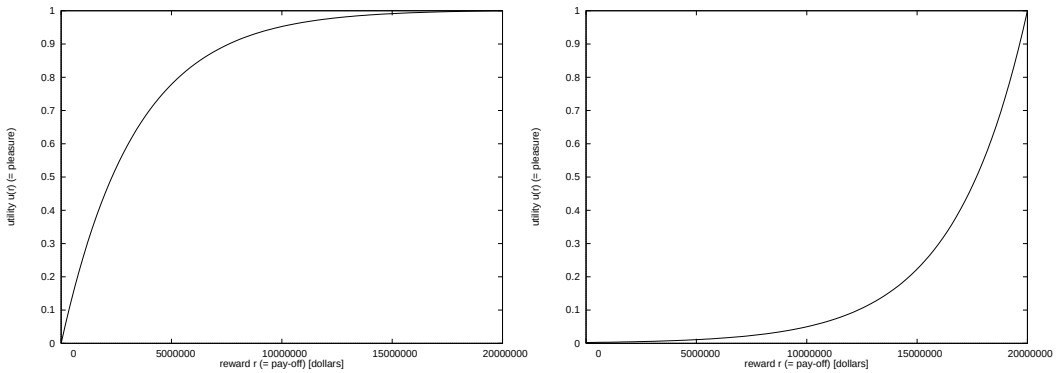


Figure 4.4: Purely Risk-Averse Attitude (left) and Purely Risk-Seeking Attitude (right)

system chose Lottery 1 for them. In general, the plans produced by planning systems should reflect the risk attitudes of the people that depend on them. Unfortunately, artificial intelligence has largely ignored how to incorporate risk attitudes into their search and planning methods.

Planning systems are usually not used to decide which lotteries to play, but often have to solve other planning tasks in high-stake one-shot planning domains. People are risk-averse for some of these planning tasks and risk-seeking for others. Examples of risk-averse planning domains are Lunar rover navigation [Simmons *et al.*, 1995] and marine oil spill containment [Blythe, 1996]. Many people prefer to avoid the huge losses that are possible in these domains. An example of a risk-seeking planning domain is robot contest participation. Imagine that your task is to design a robot for the annual AAAI robot competition [Simmons, 1995], where it has to complete a given navigation task (for example, "find the coffee pot") in as short a time as possible. You want the robot to win the competition, but – in case it loses – do not care whether it makes second or last place. You know that your robot is not much faster than your competitors' robots, maybe even a bit slower. In this case, many people prefer their robot to take chances.

Utility theory [Bernoulli, 1738, von Neumann and Morgenstern, 1947] provides a well-understood normative framework for making rational decisions according to a given risk attitude, provided that the agent accepts a few simple axioms and has unlimited planning resources available. According to utility theory, it is optimal to select the plan with maximal average utility for execution, where the utility is a strictly monotonically increasing (but not necessarily linear) function of the reward. We also make the standard assumption that the utility function is twice continuously differentiable. This allows us to describe more easily how the risk attitude of agents depends on the form of their utility functions. As an example, consider an agent with the utility function

$$u(r) \quad = \quad -\frac{e^6}{e^6-1}(e^{-0.0000003})^r + \frac{e^6}{e^6-1},$$

where r is the reward of the lottery, measured in dollars. (Utility functions are determined only up to positively linear transformations. We arbitrarily chose it so that $u(0) = 0$ and $u(20,000,000) = 1$, which explains why the utility function looks so complicated.) This utility function, shown in Figure 4.4 (left), associates the following utilities with the rewards of Lottery 1 and Lottery 2:

Reward	Utility
0 dollars	0.00
4,500,000 dollars	0.74
10,000,000 dollars	0.95

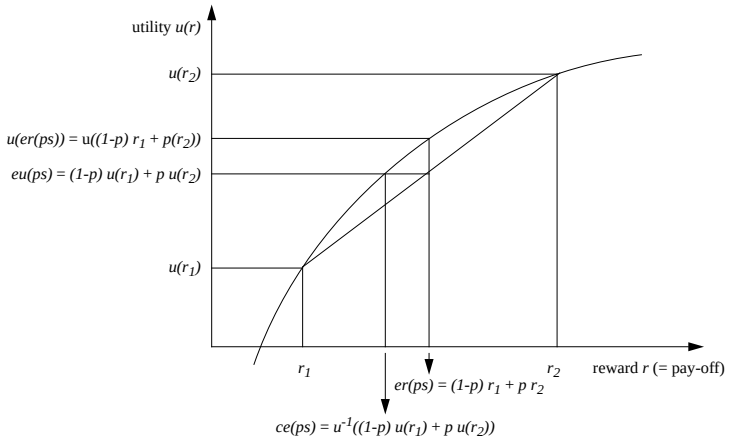


Figure 4.5: A Concave Utility Function

Thus, for an agent with this utility function and no prior wealth, the average utility of Lottery 1 is only (0.00 + 0.95)/2 = 0.48, whereas the average utility of Lottery 2 is 0.74. Similarly, the certainty equivalent of Lottery 1 is only 2,148,532 dollars, whereas the certainty equivalent of Lottery 2 is 4,500,000 dollars. Consequently, the agent clearly chooses to participate in Lottery 2. Of course, other agents can have other utility functions and thus arrive at different conclusions.

Agents can be partly risk-averse, risk-neutral, and risk-seeking. There are people, for example, that play the state lottery (risk-seeking attitude) and have insurance (risk-averse attitude). These people are mixtures of the following three pure risk attitudes [Watson and Buede, 1987]:

Purely risk-averse agents prefer a deterministic lottery over any nondeterministic lottery with the same average reward. The certainty equivalent of any nondeterministic lottery is smaller than its average reward. Purely risk-averse agents have concave utility functions, one of which is shown in Figure 4.4 (left). Figure 4.5 illustrates why agents with concave utility functions are purely risk-averse. It depicts a lottery ps with two rewards. Reward r_1 is won with probability $1 - p$ and reward r_2 is won with probability p. The figure shows that indeed $eu(ps) < u(er(ps))$ or, equivalently, $ce(ps) < er(ps)$.

Purely risk-neutral agents are indifferent between any lotteries with the same average reward. The certainty equivalent of any lottery is the same as its average reward. Purely risk-neutral agents have linear utility functions.

Purely risk-seeking agents prefer any nondeterministic lottery over a deterministic lottery with the same average reward. The certainty equivalent of any nondeterministic lottery is larger than its average reward. Purely risk-seeking agents have convex utility functions, one of which is shown in Figure 4.4 (right).

How does one determine which utility function to use for a given planning task? Since utility functions encode the individual risk attitudes of agents, they must be elicited from them on an individual basis. Since utility functions are determined only up to positively linear transformations, one can establish an arbitrary scale by fixing the utilities of two rewards, say the utility of the smallest possible reward r_1 and the largest possible reward r_2. Assume that $u(r_1) = 0$ and $u(r_2) = 1$. To determine the utility of any reward r, one asks the agent which probability p makes it indifferent between the following two lotteries: Either it receives reward $r - w$ for sure, or it receives reward $r_1 - w$ with probability $1 - p$ and reward $r_2 - w$ with probability p, where w is the *wealth* of the agent (the dollar amount that it owns initially). Then, $u(r) = u(r - w + w) = (1 - p)u(r_1 - w + w) + pu(r_2 - w + w) = p$. Farquhar [Farquhar, 1984] surveys more practical assessment procedures for utility functions.

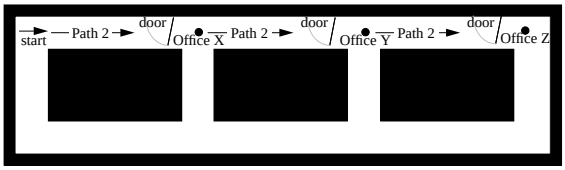


Figure 4.6: Expanded Navigation Example

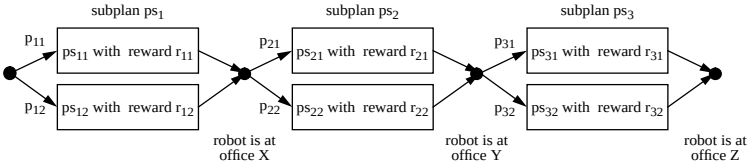


Figure 4.7: Sequential Plan with Three Parts

4.3 Maintaining Decomposability: Exponential Utility Functions

Utility theory specifies only what optimal plans are (namely, those that maximize average utility) but not how they can be obtained other than by enumerating every chronicle of every possible plan. How to plan efficiently is addressed by search and planning methods from artificial intelligence, at least for preference models different from maximizing average utility. These planners exploit the structure of the planning tasks, most notably their decomposability. This property allows planners to use the divide-and-conquer principle, which efficiently solves planning tasks by dividing them into parts, finding subplans for each part individually (or, at least, with only a small amount of interaction among the parts), and then assembling the subplans into an overall plan. Planning methods that use the divide-and-conquer principle include means-ends-analysis, nonlinear planning, and dynamic programming methods.

In this section, we show that planning tasks are not decomposable with respect to arbitrary utility functions even if the planning tasks are decomposable with respect to linear utility functions (that is, average reward). This explains why planning with arbitrary nonlinear utility functions can be harder than planning with linear utility functions: the divide-and-conquer principle cannot necessarily be used to solve planning tasks for arbitrary nonlinear utility functions, not even those that it can solve for linear utility functions. To illustrate this point, we use the following extension of Example 1 in Figure 4.2 [Koenig and Simmons, 1996d].

Example 2: Consider the following sequential planning task that consists of three parts. A robot operates in the environment shown in Figure 4.6. It has to first visit office X to pick up a form, then obtain a signature in office Y, and finally deliver the signed form to office Z. The plan of the robot consists of trying to take the route through the door (Path 2) for all three parts of the planning task (Figure 4.7). The first subplan ps_1 corresponds to reaching office X. It has two possible subchronicles ps_{11} and ps_{12}. Subchronicle ps_{11} denotes the case where the door is open, whereas subchronicle ps_{12} corresponds to the door being closed. Subchronicle ps_{11} has reward r_{11} and occurs with probability p_{11}, subchronicle ps_{12} has reward r_{12} and occurs with probability p_{12} (where $p_{11}+p_{12}=1$), and similarly for the other subplans and subchronicles. We assume that the probabilities of any two different doors being open are independent (a reasonable assumption).

The average utility of the plan from Example 2, the concatenation $ps_1 \cdot ps_2 \cdot ps_3$ of the three subplans, is

$$eu(ps_1 \cdot ps_2 \cdot ps_3) \quad = \quad \sum_{i=1}^{2} \sum_{j=1}^{2} \sum_{k=1}^{2} [p_{1i} p_{2j} p_{3k} u(r_{1i} + r_{2j} + r_{3k})].$$

The computation of the average utility can be *decomposed* if it can be split into terms each of which contains the probabilities and rewards of only one of the subplans. In general, the computation of the average utility cannot be decomposed, as can be seen by considering, for example, the utility function $u(r) = r^2$.

Few researchers have addressed this problem, which can be approached in two obvious ways:

First, one can accept the loss of decomposability and attempt to solve the harder planning task as efficiently as possible. One can enumerate all chronicles of a plan to determine its average utility. Unfortunately, the number of chronicles (here: 2^3) is exponential in the number of parts of the planning task (here: 3), making their enumeration intractable for larger planning tasks. This problem can be reduced with approximative planning methods [Haddawy and Hanks, 1993] or planning methods with limited look-ahead [Kanazawa and Dean, 1989].

Second, one can approximate the actual utility function with a utility function that makes it possible to decompose the computation of the average utility. An obvious solution is to use an additive utility function. Utility functions are *additive* if $u(r_1) + u(r_2) = u(r_1 + r_2)$ for all rewards r_1 and r_2. The planning task of Example 2 is decomposable with respect to additive utility functions. The average utility of the plan from Example 2 is

$$\begin{aligned} eu(ps_1 \cdot ps_2 \cdot ps_3) &= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}u(r_{1i}+r_{2j}+r_{3k})] \\ &= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}(u(r_{1i})+u(r_{2j})+u(r_{3k}))] \\ &= \sum_{i=1}^{2}[p_{1i}u(r_{1i})] + \sum_{j=1}^{2}[p_{2j}u(r_{2j})] + \sum_{k=1}^{2}[p_{3k}u(r_{3k})] \\ &= eu(ps_1) + eu(ps_2) + eu(ps_3). \end{aligned} \tag{4.1}$$

Thus, the computation of the average utility can be decomposed and performed in time linear in the size of the plan, without having to enumerate all eight of its chronicles. The divide-and-conquer principle can be used to find plans with maximal average utility: The planner can determine separate subplans for the three parts of the planning task, each of which maximizes the average utility of its part. These three subplans combined then form a plan with maximal average utility for the planning task.

$$\begin{aligned} \max_{x,y,z} eu(ps_1(x) \cdot ps_2(y) \cdot ps_3(z)) &\overset{\text{Formula 4.1}}{=} \max_{x,y,z}[eu(ps_1(x)) + eu(ps_2(y)) + eu(ps_3(z))] \\ &= \max_x eu(ps_1(x)) + \max_y eu(ps_2(y)) + \max_z eu(ps_3(z)). \end{aligned}$$

Unfortunately, the only additive utility functions are the linear utility functions, and Formula 4.1 just says that the average reward of a plan is the sum of the average rewards of its subplans.[1]

Approximating nonlinear utility functions with linear utility functions leads to the same results if the domain is deterministic. Furthermore, it leads to good results if the nonlinear utility functions are approximately linear. Many people, for example, are roughly risk-neutral in low-stake domains, resulting in approximately linear utility functions. Often, however, the utility functions are highly nonlinear and cannot be approximated closely with linear utility functions, especially in the presence of deadlines and in the presence of risk attitudes in

[1] To be precise, the only additive utility functions are the linear utility functions $u(r) = m \times r$ (with $m > 0$ to ensure that the utility functions are strictly monotonically increasing in the reward) but these are equivalent to all linear utility functions since utility functions are defined only up to positively linear transformations.

high-stake one-shot planning domains. Appendix 6.3 gives an example that shows that linear utility functions are not well suited for approximating the exponential utility functions that we used in Section 4.2.1 to model soft, immediate deadlines. In the example, the approximation error is roughly 2 1/2 minutes for a delivery task whose average travel time is only about 9 minutes.

We therefore investigate other utility functions that can decompose the computation of the average utility. We first show which property these utility functions must have and then which utility functions possess it. The result is that not only linear but also exponential utility functions can decompose the computation of the average utility. Exponential utility functions allow one, for example, to solve the delivery task from Appendix 6.3 precisely and efficiently.

Consider again Example 2 in Figure 4.6 and assume that the first two subplans have already been executed, resulting in a reward of w. Then, the agent is at office Y and evaluates the subplan for the third part of the planning task according to

$$ce_w(ps_3) := u^{-1}(\sum_k [p_{3k} u(r_{3k} + w)]).$$

This is the certainty equivalent of the subplan after all of its rewards have been increased by w. This value depends on w, the reward already accumulated, and thus on the subplans chosen for the previous parts of the planning task and the resulting subchronicles. This means that, in general, the Markov property (Section 3.2) no longer holds in pose space (where states correspond to poses of the robot): how to act in the future depends on how the current pose was reached. Thus, the divide-and-conquer principle can no longer be used in pose space. The Markov property could be restored by adding the accumulated reward to the state information. This, however, would increase the state space and thus also the number of plans that need to be considered, which makes planning inefficient.

To apply the divide-and-conquer principle, we need to evaluate subplans without knowing w. Thus, their evaluation must be the same for all w. Since $ce_w(ps_3) = ce(ps_3)$ for $w = 0$, we rank them according to their certainty equivalents. Now consider subplan ps_3 and a deterministic subplan whose reward is $ce(ps_3)$. Both subplans have the same certainty equivalent and thus we rank them as equally good. This is only correct if they also have the same certainty equivalent after all of their rewards have been increased by w. The resulting certainty equivalent is $ce_w(ps_3)$ for subplan ps_3 and $ce(ps_3) + w$ for the deterministic subplan. Thus,

$$u^{-1}(\sum_k [p_{3k} u(r_{3k} + w)]) = ce_w(ps_3) = ce(ps_3) + w = u^{-1}(\sum_k [p_{3k} u(r_{3k})]) + w. \tag{4.2}$$

In other words, a utility function must have the following property to decompose the calculation of the average utility: if the rewards of all chronicles of a plan are increased by some real value w, then its certainty equivalent increases by w as well. In utility theory, this property is known as the *delta property* [Howard and Matheson, 1972] or *constant local risk aversion* [Pratt, 1964], where constant local risk aversion means that the local risk aversion, defined as $u''(r)/u'(r)$, does not depend on r. The local risk aversion measures the degree of risk sensitivity better than the second derivative of the utility function since positively linear transformations of a utility function change its second derivative but not its local risk aversion [Keeney and Raiffa, 1976]. A large positive value corresponds to strongly risk-seeking agents, zero corresponds to risk-neutral agents, and a large negative value corresponds to strongly risk-averse agents. A constant local risk aversion implies that the risk attitude of an agent is independent of its wealth (that is, its choices do not depend on how wealthy it is).

Linear utility functions possess the delta property. The only nonlinear utility functions that possess the delta property are the exponential utility functions: the convex exponential functions $u(r) = \gamma^r$ with $\gamma > 1$, concave exponential functions $u(r) = -\gamma^r$ with $0 < \gamma < 1$, and their positively linear transformations [Howard and

Matheson, 1972, Watson and Buede, 1987]. This means that planning tasks are decomposable with respect to linear utility functions (that is, average reward) if and only if the planning tasks are decomposable with respect to exponential utility functions. For conciseness, we often refer to both convex and concave exponential utility functions at the same time: $u(r) = \pm\gamma^r$. According to this notation, for example, the certainty equivalent of a plan ps whose execution leads with probability p_i to a chronicle with reward r_i is $ce(ps) = \log_\gamma[\pm \sum_i [p_i(\pm\gamma^{r_i})]]$. We do not explicitly consider their positively linear transformations since they do not increase the power of planners.

4.4 Advantages of Exponential Utility Functions

We showed that planning tasks with exponential utility functions are decomposable if the planning tasks are decomposable with respect to linear utility functions (that is, average reward). This makes it possible to use the divide-and-conquer principle to plan efficiently with exponential utility functions, which is why we study these nonlinear utility functions. In this section, we argue that they can express or approximate preference models that linear utility functions cannot express at all, or approximate only poorly. For example, exponential utility functions can model a continuum of risk attitudes in high-stake one-shot planning domains, including risk-seeking and risk-averse attitudes, and they trade-off between maximizing the worst-case (minimax principle), average, and best-case reward. Furthermore, exponential utility functions are expressive for modeling immediate soft deadlines and make it easy to work with plans whose rewards can be characterized only with continuous probability distributions. In subsequent sections, we show how existing planners can plan with exponential utility functions. This allows us to generalize the preference models of many search and planning methods from artificial intelligence without giving up efficiency.

4.4.1 Expressiveness of Exponential Utility Functions

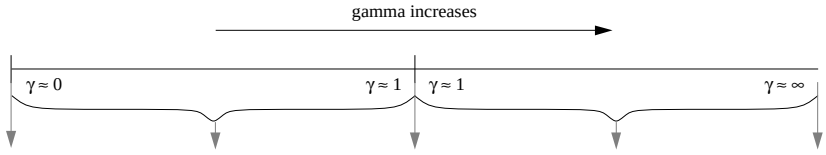
Exponential utility functions are perhaps the most often used utility functions in utility theory [Watson and Buede, 1987] and specialized assessment procedures are available that make it easy to elicit them from decision makers [Farquhar, 1984, Farquhar and Nakamura, 1988]. They may not fit all planning tasks, since they are bounded from above or below and parameterized with only one parameter γ. However, we show in this section that they are expressive. In particular, convex exponential utility functions can model immediate soft deadlines, as well as a continuum of risk-seeking attitudes in high-stake one-shot planning domains. Concave exponential utility functions can model a continuum of risk-averse attitudes in high-stake one-shot planning domains. Exponential utility functions also bridge a gap between approaches previously studied in artificial intelligence and robotics since they can trade-off between maximizing the worst-case, average, and best-case reward.

Immediate Soft Deadlines: Convex exponential utility functions can model, either exactly or approximately, many immediate soft deadlines, as argued in Section 4.2.1. Figure 4.3 (right), for example, modeled an immediate soft deadline with the convex exponential utility function $u(r) = (2^{1/300})^r$, where r is the negative travel time of the robot, measured in seconds.

Convex exponential utility functions can express immediate soft deadlines of different softness. The smaller γ, the softer the deadline. Similarly, the larger γ, the harder the deadline.

Risk Attitudes in High-Stake One-Shot Planning Domains: Exponential utility functions can also model, either exactly or approximately, risk attitudes in high-stake one-shot planning domains. For example, the risk-averse utility function in Figure 4.4 (left) is a positively linear transformation of the concave exponential utility function $u(r) = -(e^{-0.0000003})^r$ and the risk-seeking utility function in Figure 4.4 (right) is a positively linear transformation of the convex exponential utility function $u(r) = (e^{0.0000003})^r$, where r is the pay-off of the lottery, measured in dollars.

Exponential utility functions can express a continuum of pure risk attitudes ranging from being strongly risk-



risk attitude	purely extremely risk-averse	purely risk-averse	purely risk-neutral	purely risk-seeking	purely extremely risk-seeking
formalization	min-max-ing	utility theory	utility theory	utility theory	max-max-ing
utility of plan	$eu(ps) = u(\min_i r_i)$	$eu(ps) = \sum_i [p_i \cdot u(r_i)]$	$eu(ps) = \sum_i [p_i \cdot u(r_i)]$	$eu(ps) = \sum_i [p_i \cdot u(r_i)]$	$eu(ps) = u(\max_i r_i)$
utility function		concave exponential $u(r) = -m \cdot \gamma^r + n$ $(m > 0, 0 < \gamma < 1)$	linear $u(r) = m \cdot r + n$ $(m > 0)$	convex exponential $u(r) = m \cdot \gamma^r + n$ $(m > 0, \gamma > 1)$	

(plan *ps* leads with probability $p_i > 0$ to a chronicle with reward r_i for all i)

Figure 4.8: Continuum of Risk Attitudes

averse to being risk-neutral to being strongly risk-seeking (Section 4.2.2). The smaller γ, the more risk-averse (and less risk-seeking) the agent. Similarly, the larger γ, the more risk-seeking (and less risk-averse) the agent:

Proposition 1 *If the execution of a plan ps leads with probability $p_i > 0$ to a chronicle with reward r_i, then*

- *for any utility function $u(r) = -\gamma^r$ with $0 < \gamma < 1$:*

$$\lim_{\gamma \to 0} ce(ps) = \min_i r_i \qquad \text{(Case 1)}$$
$$\lim_{\gamma \to 1} ce(ps) = er(ps) \qquad \text{(Case 2)}$$

- *for any utility function $u(r) = \gamma^r$ with $\gamma > 1$:*

$$\lim_{\gamma \to 1} ce(ps) = er(ps) \qquad \text{(Case 3)}$$
$$\lim_{\gamma \to \infty} ce(ps) = \max_i r_i \qquad \text{(Case 4).}$$

Proposition 1 is common knowledge in utility theory, that we prove in Appendix 6.4. The proposition is depicted in Figure 4.8 and can be summarized as follows:

Purely Extremely Risk-Averse: Case 1 shows that the optimal plan for risk-averse agents approaches the plan with maximal **worst-case reward** as gamma approaches zero. Thus, the agents behave as if nature were an opponent that hurt them as much as possible by deliberately choosing the chronicle with the smallest reward. We call these agents *purely extremely risk-averse*. They find optimal plans using the minimax principle. A prominent example is Murphy's law, which states that everything that can go wrong will indeed go wrong. This law is attributed to Captain Ed Murphy from Wright Field Aircraft Lab [Dickson, 1978].

Purely Risk-Neutral: Cases 2 and 3 show that the optimal plan for risk-averse and risk-seeking agents approaches the plan with maximal **average reward** as gamma approaches one. Thus, these agents assume that nature is indifferent towards them and flips coins. They are purely risk-neutral (Section 4.2.2).

Purely Extremely Risk-Seeking: Case 4 shows that the optimal plan for risk-seeking agents approaches the plan with maximal **best-case reward** as gamma approaches infinity. Thus, the agents behave as if nature were a friend that helped them as much as possible by deliberately choosing the chronicle with the largest reward. We call these agents *purely extremely risk-seeking*. They assume, for example, that they win the grand prize whenever they play the lottery. A prominent example is Gladstone Gander, Donald Duck's lucky cousin.

Proposition 1 also shows that exponential utility functions are able to bridge a gap between methods previously studied in artificial intelligence and robotics, namely, methods that maximize worst-case reward (minimax principle) and methods that maximize average reward. We showed that the former preference model is the one of purely extremely risk-averse agents, whereas the latter preference model is the one of purely risk-neutral agents. However, planning methods that maximize the worst-case reward and planning methods that maximize the average reward do not have two totally different preference models. Concave exponential utility functions balance the two preference models seamlessly: They can approximate both preference models asymptotically and cover a continuum of risk attitudes in between. Similarly, convex exponential utility functions can trade-off between maximizing best-case and average reward. Thus, exponential utility functions are able to bridge a gap between methods previously studied in artificial intelligence and robotics.

The difference between finding plans that maximize the worst-case reward (minimax principle) and the average reward has been discussed extensively in robotics [Erdmann, 1989, Brost and Christiansen, 1993, LaValle and Hutchinson, 1994], in particular in the manipulation and motion planning literature, where one distinguishes nondeterministic and probabilistic uncertainty [Erdmann, 1992]. Plans with maximal worst-case reward are used to solve planning tasks with nondeterministic uncertainty and plans with maximal average reward are used to solve planning tasks with probabilistic uncertainty. Often, planning tasks are formulated as planning tasks with nondeterministic uncertainty and then solved with minimax methods, especially since domains from robotics can often be solved with them, as was briefly discussed in Section 2.3.1. The best known minimax method from robotics is probably the pre-image method [Lozano-Perez *et al.*, 1984]. Other minimax methods from robotics include [Lozano-Perez *et al.*, 1984, Latombe *et al.*, 1991, Latombe, 1991, Erdmann, 1984]. Minimax methods from artificial intelligence were discussed in Chapter 2. Examples include the Min-Max LRTA* method [Koenig and Simmons, 1995b], the Parti-Game method [Moore and Atkeson, 1995], the IG method [Genesereth and Nourbakhsh, 1993], and the $\hat{\text{Q}}$-learning method [Heger, 1996].

4.4.2 Handling Continuous Reward Distributions with Exponential Utility Functions

We have assumed that a chronicle of a plan is characterized by its probability and reward. Sometimes, however, chronicles can be characterized only by continuous probability distributions over the rewards. The resulting average utility can be expressed in closed form for exponential utility functions, as the average reward plus or minus a fraction of the variance. This is why exponential utility functions make it easy to work with continuous probability distributions over the rewards.

If the execution of plan ps leads with probability p_i to a chronicle with reward r_i, then the average utility of the plan is

$$eu(ps) = \sum_i [p_i u(r_i)].$$

If, on the other hand, the execution of plan ps leads with probability p_i to a chronicle with rewards r that are distributed according to probability distribution $f_i(r)$, then the average utility of the plan is

$$eu(ps) = \sum_i [p_i \int_{-\infty}^{\infty} f_i(r) u(r)\, dr].$$

An advantage of exponential utility functions is that the integral can easily be solved for normal distributions, probably the most common continuous probability distributions. If $N(\mu, \sigma, r)$ denotes a normal distribution of r with mean μ and standard deviation σ, then

$$\begin{aligned}
eu(ps) &= \sum_i [p_i \int_{-\infty}^{\infty} N(\mu_i, \sigma_i, r) u(r)\, dr] \\
&= \sum_i [p_i \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma_i} e^{\frac{-(r-\mu_i)^2}{2\sigma_i^2}} (\pm\gamma^r)\, dr] \\
&= \pm \sum_i [p_i \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma_i} e^{\frac{-(r-\mu_i)^2}{2\sigma_i^2} + r \ln \gamma}\, dr] \\
&= \pm \sum_i [p_i\, e^{\mu_i \ln \gamma + \frac{1}{2}\sigma_i^2 \ln^2 \gamma} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma_i} e^{\frac{-(r-(\mu+\sigma_i^2 \ln \gamma))^2}{2\sigma_i^2}}\, dr] \\
&= \pm \sum_i [p_i\, e^{\mu_i \ln \gamma + \frac{1}{2}\sigma_i^2 \ln^2 \gamma} \times 1] \\
&= \sum_i [p_i\, (\pm\gamma^{\mu_i + \frac{1}{2}\sigma_i^2 \ln \gamma})] \\
&= \sum_i [p_i\, u(\mu_i + \frac{1}{2}\sigma_i^2 \ln \gamma)]. \qquad (4.3)
\end{aligned}$$

Thus, a planning task for which the rewards r of chronicle i are distributed according to normal distribution $N(\mu_i, \sigma_i, r)$ is equivalent to a planning task for which the reward of chronicle i is deterministic, namely, $r = \mu_i + \frac{1}{2}\sigma_i^2 \ln \gamma$ with probability one. The second term of this expression is negative for concave exponential utility functions and positive for convex exponential utility functions. This shows that the average utility of normal distributions over the rewards can be characterized in closed form for exponential utility functions. It also shows that the expectation can be separated from the variance. Some researchers approximate plans with maximal average utility with plans that maximize average reward plus or minus a fraction of the variance [Filar *et al.*, 1989, Karakoulas, 1993]. Formula 4.3 shows that this is not an approximation but maximizes average utility exactly for exponential utility functions and normal-distributed rewards.

4.5 Planning with Exponential Utility Functions

In this section, we show how to find plans efficiently that achieve the goal with maximal average utility (preference model C5) if the utility function is exponential. In particular, we show how existing planners from artificial intelligence can be used to plan with exponential utility functions, including those planners that do not reason about rewards at all.

We showed that planning tasks with exponential utility functions are decomposable if the planning tasks are decomposable with respect to linear utility functions (that is, average reward). This observation can be used to make planning with exponential utility functions efficient. However, standard planners from artificial intelligence cannot be used directly to maximize average utility for exponential utility functions. This is so because, among all preference models of standard planners from artificial intelligence that we studied, only preference model C4 works with rewards in probabilistic domains, but it requires additive utility functions and only linear utility functions are additive.

In the following, we develop efficient methods that transform planning tasks with exponential utility functions to planning tasks that existing planners from artificial intelligence can solve (Figure 4.9). The transformations

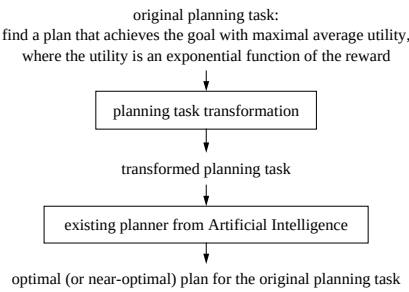


Figure 4.9: Planning-Task Transformations

are such that an optimal plan for the transformed planning task is also optimal for the original one, and a good ("satisficing") plan for the transformed planning task is also good ("satisficing") for the original one. Thus, the transformed planning task can be solved with both optimal and suboptimal ("satisficing") planners.

We study two transformations, which we call the additive and multiplicative planning-task transformations. Both transformations are simple context-insensitive representation changes. The importance of representation changes for planning has been recognized earlier, for example in [Newell, 1965, Newell, 1966, Amarel, 1968, Hayes and Simon, 1976, Kaplan and Simon, 1990, Fink, 1995]. Our representation changes are fast and scale well. They can be performed on a variety of planning-task representations without changing their kind and size, and can be used as black-box methods (that is, they can be applied without an understanding of how or why they work).

The additive planning-task transformation applies to special cases of planning tasks, namely, planning tasks where each subplan ("action") starts in only one state and is "deterministic" in the sense that its execution always ends in the same state although it can result in different subchronicles and rewards. The start and end states of the subplan can be different, and the start and end states of different subplans can be different as well. These planning tasks can be solved optimally with sequential plans (unconditional sequences of subplans). An example are planning tasks with a number of given subgoals that have to be achieved in some predetermined ordering. Some delivery tasks fit this description. The transformed planning task requires one to find a plan that achieves the goal with maximal reward in a deterministic domain. It can be solved with all planners whose preference model is C2 or C4.

The multiplicative planning-task transformation applies to planning tasks that can be solved optimally with conditional plans. The transformed planning task requires one to find a plan with maximal probability of goal achievement or, alternatively, a plan that achieves the goal with maximal average reward in a probabilistic domain. It can be solved with many planners whose preference model is C3 and all planners whose preference model is C4, provided that either the exponential utility function of the original planning task is convex and all rewards are negative, or the exponential utility function is concave and all rewards are positive. This is, for example, the case for planning with immediate soft deadlines, for risk-seeking planning with resource consumptions, and for risk-averse planning with lottery pay-offs.

Consequently, the multiplicative planning-task transformation applies to a larger class of planning tasks but has the disadvantage that the transformed plans, in some cases, cannot be solved with all planners. We also show that the multiplicative planning-task transformation has the disadvantage that it can amplify the errors of planners that can find only near-optimal plans for the transformed planning tasks, whereas the additive planning-task transformation leaves the errors unchanged.

For both the additive and multiplicative planning-task transformation, the transformed planning tasks are solved with existing planners from artificial intelligence, which extends their functionality to planning with exponential utility functions. It also makes planning with exponential utility functions as fast as planning for traditional preference models from artificial intelligence and enables one to participate in performance improvements achieved by other researchers in the currently very active field of deterministic and probabilistic

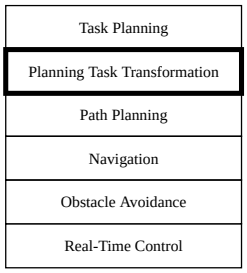


Figure 4.10: Augmented Mobile-Robot Architecture

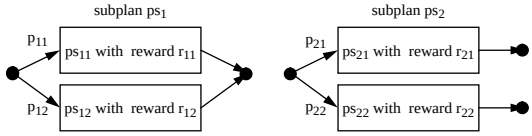


Figure 4.11: "Deterministic" (left) and "Nondeterministic" (right) Subplans

planning. Furthermore, it makes it easy to integrate the planning-task transformations into agent architectures to change their preference models. For example, we can easily modify the mobile-robot architecture to plan paths in the presence of immediate soft deadlines or risk attitudes in high-stake one-shot planning domains. This can be done by using its path planner unchanged on a transformation of the path-planning task, which is produced by a new planning-task transformation layer that sits between the task-planning and path-planning layers (Figure 4.10).

We first describe the additive planning-task transformation and then the multiplicative planning-task transformation.

4.5.1 The Additive Planning-Task Transformation

The additive planning-task transformation applies to special cases of planning tasks. These planning tasks consist of a goal and a collection of atomic subplans ("actions") that can be used as building blocks to construct plans that achieve the goal. Each subplan starts in only one state (which is no restriction) and be "deterministic" in the sense that its execution always ends in the same state although it can result in different subchronicles and rewards. The start and end states of the subplan can be different, and the start and end states of different subplans can be different as well. Figure 4.11 shows a subplan that satisfies this requirement (left) and one that does not satisfy it (right). The requirement is similar to the assumption of the planning methods in [Loui, 1983] and [Wellman *et al.*, 1995], which also apply to planning tasks with subplans whose effects are deterministic but whose rewards can be nondeterministic.

The *additive planning-task transformation* converts the planning task by modifying all of its subplans (everything else remains the same): If a subplan can be executed in state s and its execution leads with probability p_i and reward r_i to state s' (for all i), then it is replaced with a deterministic subplan. This subplan can be executed in state s and its execution leads with probability one and reward $u^{-1}(\sum_i [p_i u(r_i)])$ to state s'. This reward is the certainty equivalent of the original subplan for the exponential utility function. The transformation is such that a plan that achieves the goal with maximal reward for the transformed planning task also achieves the goal with maximal average utility for the original planning task. Figure 4.12 summarizes the additive planning-task transformation. Notice that the transformation does not depend on the exact representation of the subplans.

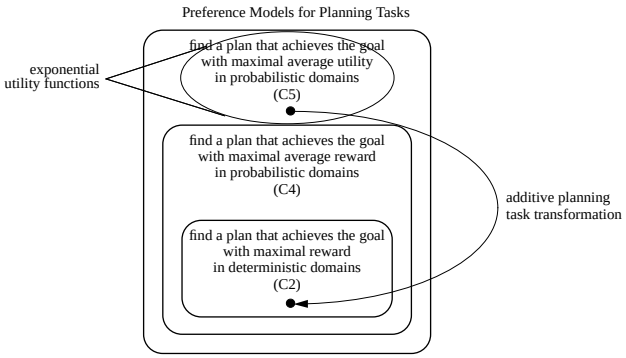


Figure 4.12: Additive Planning-Task Transformation

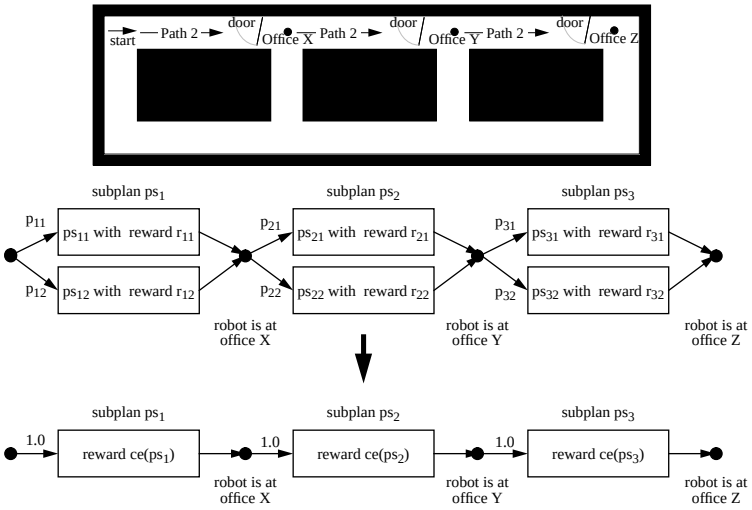


Figure 4.13: Transformed Sequential Plan for the Additive Planning-Task Transformation

As an example consider again Example 2, which is repeated in Figure 4.13 (top). A robot has to first visit office X to pick up a form, then obtain a signature in office Y, and finally deliver the signed form to office Z. One of the subplans for moving from office X to office Y is to move on a shortest path to office Y and, if the door is closed, return to office X and take the long path to office Y. This subplan can be executed when the robot is at office X and always moves the robot to office Y, although its execution can result in different travel times. If the robot reaches office Y in 120.00 seconds with probability 0.50 and in 576.00 seconds with probability 0.50, then the subplan is replaced with a deterministic subplan that can be executed when the robot is at office X and whose execution moves the robot with reward $u^{-1}(0.50\,u(-120.00) + 0.50\,u(-576.00))$ to office Y.

We now explain why the additive planning-task transformation works.

Among all plans that achieve the goal for the original planning task with maximal average utility, there is always a *sequential plan*, that is, a sequence of subplans. This is so because the execution of every subplan always ends in only one state. Since exponential utility functions satisfy the delta property, it is unimportant how this state was reached or how much reward was accumulated. Thus, there must be an optimal plan that always executes the same subplan in this state. Thus, if a planner determines a plan that achieves the goal for the original planning task with maximal average utility *among all sequential plans*, then this plan also achieves the goal for the original planning task with maximal average utility *among all plans*. In other words,

planning can be restricted to sequential plans without giving up optimality. (We discuss suboptimal planning in Section 4.5.3.)

Now consider the certainty equivalent of any sequential plan for the original planning task. As an example, we use again the plan in Figure 4.13 (center). It solves the planning task from Example 2 by trying to take the route through the door (Path 2) for all three parts of the planning task. Its certainty equivalent for the original planning task is

$$
\begin{aligned}
& ce(ps_1 \cdot ps_2 \cdot ps_3) \\
&= u^{-1}(eu(ps_1 \cdot ps_2 \cdot ps_3)) \\
&= u^{-1}(\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}u(r_{1i}+r_{2j}+r_{3k})]) \\
&= \log_\gamma[\pm\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}(\pm\gamma^{r_{1i}+r_{2j}+r_{3k}})]] \\
&= \log_\gamma\sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}\gamma^{r_{1i}+r_{2j}+r_{3k}}] \\
&= \log_\gamma(\sum_{i=1}^{2}[p_{1i}\gamma^{r_{1i}}]\times\sum_{j=1}^{2}[p_{2j}\gamma^{r_{2j}}]\times\sum_{k=1}^{2}[p_{3k}\gamma^{r_{3k}}]) \\
&= \log_\gamma\sum_{i=1}^{2}[p_{1i}\gamma^{r_{1i}}]+\log_\gamma\sum_{j=1}^{2}[p_{2j}\gamma^{r_{2j}}]+\log_\gamma\sum_{k=1}^{2}[p_{3k}\gamma^{r_{3k}}] \\
&= \log_\gamma[\pm\sum_{i=1}^{2}[p_{1i}(\pm\gamma^{r_{1i}})]]+\log_\gamma[\pm\sum_{j=1}^{2}[p_{2j}(\pm\gamma^{r_{2j}})]]+\log_\gamma[\pm\sum_{k=1}^{2}[p_{3k}(\pm\gamma^{r_{3k}})]] \\
&= u^{-1}(eu(ps_1))+u^{-1}(eu(ps_2))+u^{-1}(eu(ps_3)) \\
&= ce(ps_1)+ce(ps_2)+ce(ps_3). \qquad (4.4)
\end{aligned}
$$

This means that the computation of the certainty equivalent of a sequential plan for the original planning task can be decomposed: it is the sum of the certainty equivalents of its subplans. Instead of calculating the certainty equivalent of the sequential plan directly, we can first transform the plan. Its structure remains unchanged, but all of its subplans are transformed (as described above): they are made deterministic, and their new rewards are the same as the certainty equivalents of the original subplans. The reward of the transformed plan is the same as the certainty equivalent of the original plan. Figure 4.13 (bottom), for example, shows the transformation of the plan in Figure 4.13 (center). The reward of the transformed plan is $ce(ps_1)+ce(ps_2)+ce(ps_3)$, which is the certainty equivalent of the original plan according to Formula 4.4.

The additive planning-task transformation can be used to find a plan that achieves the goal with maximal average utility for the original planning task: We transform all subplans of the planning task. Then, a plan that achieves the goal with maximal reward for the transformed planning task also achieves the goal with maximal average utility for the original planning task. This is so because there is a bijection from all plans that achieve the goal for the transformed planning task to all sequential plans that achieve the goal for the original planning task, and the reward of a plan for the transformed planning task is the same as its certainty equivalent for the original planning task (as shown above). Consequently, if a plan achieves the goal with maximal reward for the transformed planning task, then it achieves the goal with maximal average utility among all sequential plans for the original planning task, and as we have argued, it is impossible for a nonsequential plan to have a larger average utility.

To summarize, the original planning task can be solved by applying the additive planning-task transformation

and then solving the transformed planning task with *any* planner with preference model C2 or C4, including heuristic search methods such as the A* method. It is easy to derive admissible heuristic functions for the transformed planning task because the rewards of the transformed subplans can easily be overestimated without calculating them explicitly. (We need to overestimate the rewards instead of underestimating them because we operate with rewards instead of costs.) The reward of any subplan for the transformed planning task is its certainty equivalent for the original planning task, and the certainty equivalent is always at most as large as the best-case reward. To see this, consider any plan ps that leads with probability p_i to a chronicle with reward r_i. Then,

$$ce(ps) = u^{-1}(\sum_i [p_i u(r_i)]) \leq u^{-1}(\sum_i [p_i u(\max_i r_i)]) = u^{-1}(u(\max_i r_i)) = \max_i r_i.$$

Consequently, any heuristic function that is admissible for the best-case reward of the original planning task is also admissible for the reward of the transformed planning task. For concave exponential utility functions, a heuristic function that is at least as informed (if not better informed) can be obtained as follows: For concave exponential utility functions, the certainty equivalent of any plan is at most as large as its average reward (Section 4.2.2). This means that any heuristic function that is admissible for the average reward of the original planning task is also admissible for the reward of the transformed planning task if the exponential utility function is concave.

In the following, we give two examples, both indoor delivery tasks, that fit the assumption of the additive planning-task transformation.

The first example is the planning task from Example 2: A robot has to first visit office X to pick up a form, then obtain a signature in office Y, and finally deliver the signed form to office Z.

The planning task specifies the subgoals, the ordering in which they have to be achieved, and the subplans that can be used to achieve them. This planning task satisfies the assumptions of the additive planning-task transformation because all subplans are able to move the robot to its destination, although the robot can take different paths and thus incur different travel times.

The additive planning-task transformation applies as follows:

$$\begin{aligned} \max_{x,y,z} ce(ps_1(x) \cdot ps_2(y) \cdot ps_3(z)) &\overset{\text{Formula 4.4}}{=} \max_{x,y,z} [ce(ps_1(x)) + ce(ps_2(y)) + ce(ps_3(z))] \\ &= \max_x ce(ps_1(x)) + \max_y ce(ps_2(y)) + \max_z ce(ps_3(z)). \end{aligned}$$

Thus, to find a plan with maximal average utility, a planner can determine separate subplans for the three parts of the planning task, each of which maximizes the average utility for its part. The three subplans combined then form a plan with maximal average utility for the planning task. In this case, the parts of the planning task can be solved completely independently. This is no longer possible in the next example.

The second example is similar to the first example: A robot starts at the secretary's office with the task of collecting ten signatures on a form and returning it to the secretary. We assume that all ten people are in their offices. This planning task is essentially one of task sequencing: we have to determine the order in which to visit the ten offices.

The planning task specifies the subgoals and the subplans that can be used to achieve them, but not the ordering in which the subplans have to be achieved. This planning task satisfies the assumptions of the additive planning-task transformation because all subplans are able to move the robot to its destination, although the robot can take different paths and thus incur different travel times.

The additive planning-task transformation converts the planning task to a traveling salesman problem on a directed graph with deterministic edge rewards. The reward of an edge is the certainty equivalent of the

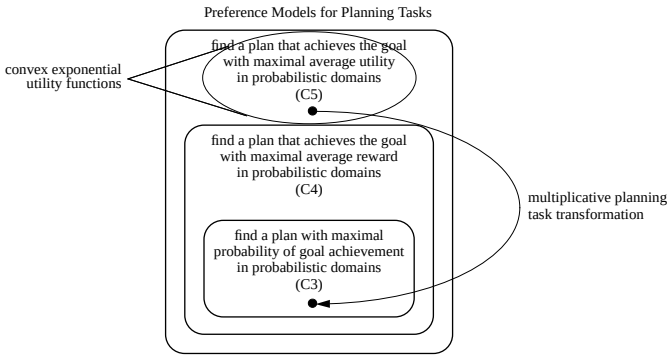


Figure 4.14: Multiplicative Planning-Task Transformation

navigation plan with maximal average utility between the corresponding two locations. The traveling salesman problem can then be solved with traveling salesman methods, traditional search and planning methods from artificial intelligence, or scheduling methods in case additional sequencing constraints are given, such as "office X must be visited before office Y."

While the additive planning-task transformation is useful, it cannot be used to solve all planning tasks because it is often not the case that each subplan is atomic and always ends in the same state. As an example, consider again a robot that has to collect ten signatures. The subplans move the robot to a specified office. The subplans are not atomic in practice because their execution can be interrupted. For example, while the robot moves to some office, it might have to take a detour because a door is closed. If this detour leads past another office, it can be advantageous for the robot to go first to that office in order to obtain another signature on the way to its original destination. This suggests considering "move to the door" as a subplan because this would allow the robot to re-plan and change its destination when it recognizes that the door is closed. However, this subplan can end in two different states: the robot is at the door and the door is either open or closed. Thus, it does not satisfy the assumption of the additive planning-task transformation and we have to consider ways of finding *conditional* plans that achieve the goal with maximal average utility for exponential utility functions.

4.5.2 The Multiplicative Planning-Task Transformation

The multiplicative planning-task transformation is more general than the additive planning-task transformation. It applies to planning tasks that can be solved optimally with conditional plans. The planning tasks again consist of a goal and a collection of atomic subplans ("actions") that can be used as building blocks to construct plans that achieve the goal, but it is no longer required that the execution of each subplan end in only one state.

For now, we assume that the **exponential utility function is convex and all rewards are negative**. This is, for example, the case for planning with immediate soft deadlines and for risk-seeking planning with resource consumptions. We discuss the other cases later in this section.

The *multiplicative planning-task transformation* converts the planning task by modifying all of its subplans (everything else remains the same): If a subplan can be executed in state s and its execution leads with probability p_i and reward r_i to state s_i (for all i), then it is replaced with a subplan that can be executed in state s and whose execution leads with probability $p_i\gamma^{r_i}$ to state s_i (for all i) and with probability $1-\sum_i[p_i\gamma^{r_i}]$ to a new nongoal state ("death") in which execution stops. The rewards do not matter. The transformation is such that a conditional plan that achieves the goal with maximal probability for the transformed planning task and always stops in the goal states or "death" also achieves the goal with maximal average utility for the original planning task. The restriction of always stopping in the goal states or "death" is there only to ensure that the plan achieves the goal for the original planning task. We later show in Section 4.6.2 how the

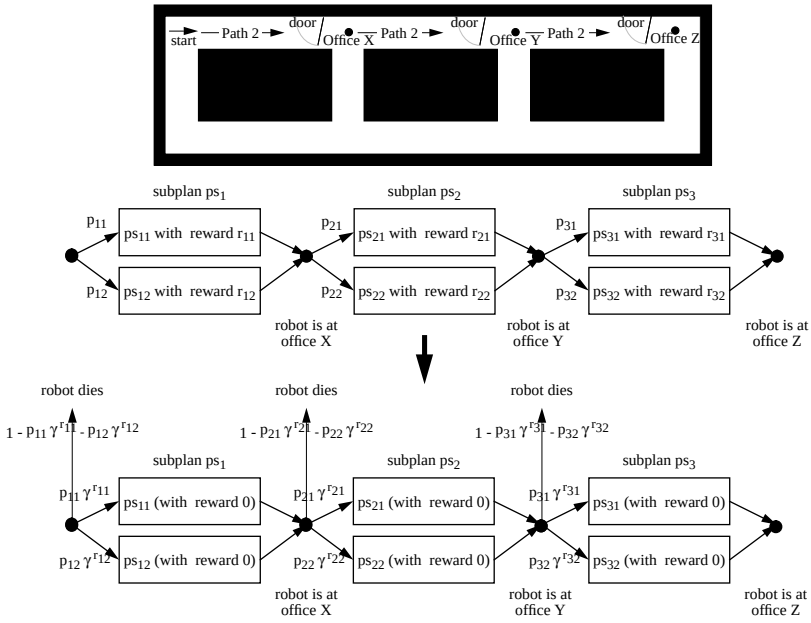


Figure 4.15: Transformed Sequential Plan for the Multiplicative Planning-Task Transformation

restriction can be dropped. Figure 4.14 summarizes the multiplicative planning-task transformation. Notice that the transformation does not depend on the exact representation of the subplans.

As an example consider again Example 2, that is repeated in Figure 4.15 (top). A robot has to first visit office X to pick up a form, then obtain a signature in office Y, and finally deliver the signed form to office Z. One of the subplans for moving from office X to office Y is to move on a shortest path to office Y and, if the door is closed, return to office X and take the long path to office Y. If the robot reaches office Y in 120.00 seconds with probability 0.50 and in 576.00 seconds with probability 0.50, then the subplan is replaced with a subplan that can be executed when the robot is at office X and whose execution moves the robot with probability $0.5\gamma^{-120.00} + 0.5\gamma^{-576.00}$ to office Y and with probability $1 - 0.5\gamma^{-120.00} - 0.5\gamma^{-576.00}$ to the new nongoal state "death."

We now explain why the multiplicative planning-task transformation works.

We first consider sequential plans as a special case and then conditional plans. As an example, we use again the sequential plan from Figure 4.15 (center). It solves the planning task from Example 2 by trying to take the route through the door (Path 2) for all three parts of the planning task. Its average utility for the original planning task is

$$\begin{aligned} & eu(ps_1 \cdot ps_2 \cdot ps_3) \\ &= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}u(r_{1i} + r_{2j} + r_{3k})] \\ &= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}\gamma^{r_{1i}+r_{2j}+r_{3k}}] \\ &= \sum_{i=1}^{2}[p_{1i}\gamma^{r_{1i}}] \times \sum_{j=1}^{2}[p_{2j}\gamma^{r_{2j}}] \times \sum_{k=1}^{2}[p_{3k}\gamma^{r_{3k}}] \end{aligned}$$

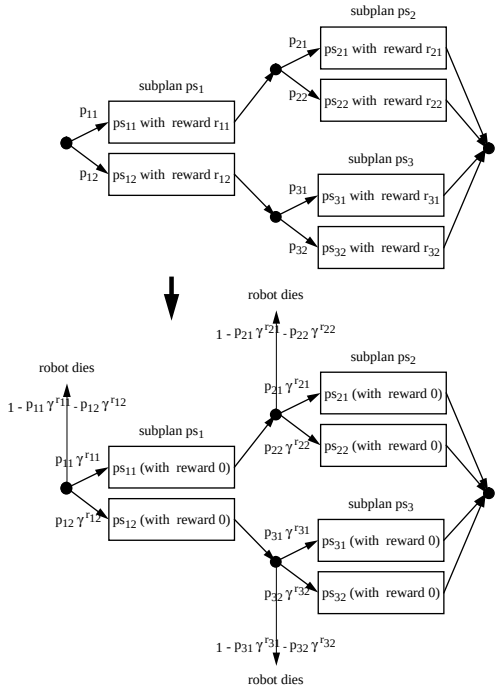


Figure 4.16: Transformed Conditional Plan for the Multiplicative Planning-Task Transformation

$$= \sum_{i=1}^{2} \overline{p}_{1i} \times \sum_{j=1}^{2} \overline{p}_{2j} \times \sum_{k=1}^{2} \overline{p}_{3k}, \tag{4.5}$$

where the parameters $\overline{p}_{mn}$ are new values with $\overline{p}_{mn} := p_{mn}\gamma^{r_{mn}}$. These values satisfy $0 \leq \overline{p}_{mn} \leq p_{mn}$ according to our assumption that the exponential utility function is convex ($\gamma > 1$) and all rewards are negative ($r_{mn} < 0$).

Instead of calculating the average utility of the sequential plan directly, we can first transform the plan. Its structure remains unchanged, but all of its subplans are transformed (as described above): If a subplan can be executed in state s and its execution leads with probability p_i and reward r_i to state s_i (for all i), then it is replaced with a subplan that can be executed in state s and whose execution leads with probability $p_i\gamma^{r_i}$ to state s_i (for all i) and with probability $1 - \sum_i [p_i\gamma^{r_i}]$ to a new nongoal state ("death") in which the execution stops. The rewards do not matter. The average utility of the original plan is the same as the probability of not dying while executing the transformed plan, which is the product of the probabilities of not dying while executing its subplans. Figure 4.15 (bottom), for example, shows the transformation of the plan in Figure 4.15 (center). The probability of not dying during the execution of the transformed plan is $\sum_{i=1}^{2}[p_{1i}\gamma^{r_{1i}}] \times \sum_{j=1}^{2}[p_{2j}\gamma^{r_{2j}}] \times \sum_{k=1}^{2}[p_{3k}\gamma^{r_{3k}}]$, which is the average utility of the original plan according to Formula 4.5.

Now consider an arbitrary conditional plan for the original planning task. We can use the same transformation on the conditional plan and it remains true that the average utility of the original conditional plan is the same as the probability of not dying while executing the transformed conditional plan. This is so because the average utility of the original conditional plan is the sum of the utility contributions of its chronicles, where the utility contribution of a chronicle is the product of its probability and utility. A chronicle is a sequence of subchronicles. If subchronicle i of the chronicle has probability p_i and reward r_i, then the utility contribution of the chronicle is

$$\prod_i p_i \, u(\sum_i r_i) = \prod_i p_i \, \gamma^{\sum_i r_i} = \prod_i [p_i \gamma^{r_i}] = \prod_i \overline{p}_i .$$

This is the same as the probability of the same chronicle for the transformed conditional plan. The sum of these probabilities is the probability of not dying during the execution of the transformed conditional plan. Figure 4.16, for example, shows a conditional plan and its transformation. The average utility of the original conditional plan and the probability of not dying during the execution of the transformed conditional plan are $p_{11}\gamma^{r_{11}} \sum_{j=1}^{2} [p_{2j}\gamma^{r_{2j}}] + p_{12}\gamma^{r_{12}} \sum_{k=1}^{2} [p_{3k}\gamma^{r_{3k}}]$.

The multiplicative planning-task transformation can be used to find a conditional plan with maximal average utility for the original planning task: We first transform all subplans of the planning task. Then, a conditional plan that minimizes the probability of dying for the transformed planning task also maximizes the average utility for the original planning task. Preference model C5, however, requires one to find a conditional plan that *achieves the goal* with maximal average utility. This complicates matters but the multiplicative planning-task transformation can be used for this purpose as well: We first transform all subplans of the planning task. Then, a conditional plan that achieves the goal with maximal probability for the transformed planning task and always stops in the goal states or "death" also achieves the goal with maximal average utility for the original planning task. This is so because there is a bijection from all conditional plans of the transformed planning task that always stop in the goal states or "death" to all conditional plans that achieve the goal for the original planning task, and the probability of not dying during the execution of a conditional plan for the transformed planning task is the same as its average utility for the original planning task (as shown above). Now consider all conditional plans for the transformed planning task that always stop in the goal states or "death." Their probability of achieving the goal is the same as the probability of not dying during their execution. Consequently, a conditional plan among them that achieves the goal with maximal probability is also a conditional plan among them that maximizes the probability of not dying during its execution, and it also achieves the goal with maximal average utility among all conditional plans that achieve the goal for the original planing task.

To summarize, the original planning task can be solved by applying the multiplicative planning-task transformation and then solving the transformed planning task with any planner that has preference model C3 and is able to consider only conditional plans that always stop in the goal states or "death." Thus, perhaps surprisingly, planners that do not reason about rewards at all can be used to find conditional plans with maximal average utility. The transformed planning task can also be solved with *any* planner with preference model C4 by declaring "death" another goal state and making the rewards for stopping in goal states other than "death" one and all other rewards zero. Notice that the planner with preference model C4 has to be able to handle zero rewards.

If **the exponential utility function is convex but not all rewards are negative**, then the $\overline{p}_{mn}$ are not necessarily probabilities: they are still nonnegative, but their sum $\sum_n \overline{p}_{mn}$ can exceed one. There are dynamic programming methods that can solve such planning tasks, but some other methods might break. I expect that many planners from artificial intelligence are still able to solve the transformed planning tasks because they can deal with parameters $\overline{p}_{mn}$ whose sum can be larger than one, although they can then no longer be interpreted as probabilities. Similarly, the value $1 - \sum_n \overline{p}_{mn}$ can become negative and then no longer be interpreted as the probability of dying. In this case, we do not model the probabilities of dying explicitly, which is no problem, since they do not enter the calculations explicitly.

So far, we assumed that the exponential utility function is convex. Now assume that **the exponential utility function is concave and all rewards are positive**. This is, for example, the case for risk-averse planning with lottery pay-offs. In this case, the average utility from Formula 4.5 becomes

$$eu(ps_1 \cdot ps_2 \cdot ps_3)$$

$$
\begin{aligned}
&= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}\,u(r_{1i}+r_{2j}+r_{3k})] \\
&= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2}[p_{1i}p_{2j}p_{3k}\,(-\gamma^{r_{1i}+r_{2j}+r_{3k}})] \\
&= -\sum_{i=1}^{2}[p_{1i}\gamma^{r_{1i}}] \times \sum_{j=1}^{2}[p_{2j}\gamma^{r_{2j}}] \times \sum_{k=1}^{2}[p_{3k}\gamma^{r_{3k}}] \\
&= -\sum_{i=1}^{2}\overline{p}_{1i} \times \sum_{j=1}^{2}\overline{p}_{2j} \times \sum_{k=1}^{2}\overline{p}_{3k},
\end{aligned}
$$

where the parameters $\overline{p}_{mn}$ are new values with $\overline{p}_{mn} := p_{mn}\gamma^{r_{mn}}$. These values satisfy $0 \leq \overline{p}_{mn} \leq p_{mn}$ according to our assumption that the exponential utility function is concave ($0 < \gamma < 1$) and all rewards are positive ($r_{mn} > 0$).

This means that we can continue to use the multiplicative planning-task transformation from above, but now the *negative* probability of not dying while executing any conditional plan for the transformed planning task is the same as its average utility for the original planning task. Consequently, if a conditional plan achieves the goal with *minimal* probability for the transformed planning task and always stops in the goal states or "death," then it achieves the goal with maximal average utility for the original planning task. Achieving the goal with minimal probability is the same as achieving "death" with maximal probability for conditional plans that always stop in the goal states or "death." Thus, the transformed planning task can be solved with any planner that has preference model C3 and is able to consider only conditional plans that always stop in the original goal states or "death" by making "death" the only goal state. The transformed planning task can also be solved with *any* planner with preference model C4 by declaring "death" another goal state and making the rewards for achieving "death" one and all other rewards zero.

If **the exponential utility function is concave but not all rewards are positive**, then the $\overline{p}_{mn}$ are not necessarily probabilities: they are still nonnegative, but their sum can exceed one. Again, many planners might still be able to solve the transformed planning task because they can deal with parameters $\overline{p}_{mn}$ whose sum can be larger than one. Similarly, the value $1 - \sum_n \overline{p}_{mn}$ can become negative and then no longer be interpreted as the probability of dying. In this case, we do not model the probabilities of dying explicitly, which is no problem for finding a conditional plan that achieves the goal with minimal probability for the transformed planning task and always stops in the goal states or "death," since the probabilities of dying do not enter the calculations explicitly, but it is no longer possible to find a conditional plan that achieves "death" with maximal probability and always stops in the original goal states or "death" (because this required us to model the probabilities of dying explicitly).

4.5.3 Suboptimal Planning with the Planning-Task Transformations

We have assumed so far that a planner is available that finds optimal plans for the transformed planning tasks. However, both the additive and multiplicative planning-task transformation work equally well with *suboptimal planners*, which can find only near-optimal ("satisficing") plans for the transformed planning tasks. These plans then solve the original planning tasks only approximately. The *error* incurred is the difference in certainty equivalents for the original planning tasks between these plans and the plans that achieve the goal with maximal average utility. In this section, we analyze the error for the additive and multiplicative planning-task transformation. For the additive planning-task transformation, we show that a suboptimal planner that has a certain absolute or relative error for the transformed planning task has the same absolute or relative error, respectively, for the original planning task. The multiplicative planning-task transformation, on the other hand, can magnify the error of a suboptimal planner.

Consider the additive planning-task transformation. In this case, the reward of any plan for the transformed planning task is the same as its certainty equivalent for the original planning task. Thus, the larger its reward for the transformed planning task, the larger its certainty equivalent for the original planning task, and a near-optimal plan for the transformed planning task is also near-optimal for the original planning task. This means that the additive planning-task transformation can be used in conjunction with either optimal or suboptimal planners to produce plans that either maximize or approximately maximize, respectively, average utility. Furthermore, a suboptimal planner that has a certain absolute or relative error for the reward of the transformed planning task has the same absolute or relative error, respectively, for the certainty equivalent of the original planning task.

Now consider the multiplicative planning-task transformation and assume that the exponential utility function is convex. In this case, the probability of not dying during the execution of any plan is the same as its average utility for the original planning task. Thus, the larger the probability of not dying during the execution of the transformed plan, the larger the average utility of the original plan, and a near-optimal plan for the transformed planning task is also near-optimal for the original planning task. This means that the multiplicative planning-task transformation can be used in conjunction with either optimal or suboptimal planners to produce plans that either maximize or approximately maximize, respectively, average utility. An analogous argument holds for concave exponential utility functions. Both for convex and concave exponential utility functions, however, it is not the case that a suboptimal planner with a certain absolute or relative error for the transformed planning task has the same absolute or relative error for the original planning task. This is a disadvantage of the multiplicative planning-task transformation. We explain this in the following.

Assume that the utility function is convex exponential. If the optimal plan for the transformed planning task achieves the goal with probability p and stops in only the goal states or "death," then the probability of not dying during its execution is p. Consequently, the average utility of the optimal plan that achieves the goal for the original planning task is p and its certainty equivalent is $\log_\gamma p$. Now consider a suboptimal planner with additive error $\epsilon > 0$. If this planner is used to solve the transformed planning task, it can potentially find a plan that stops in only the goal states or "death" and whose probability of goal achievement is only $p - \epsilon$ (but not worse). Thus, the corresponding plan for the original planning task achieves the goal but its average utility is only $p - \epsilon$ and its certainty equivalent is only $\log_\gamma [p - \epsilon]$. To determine how close to optimal this plan is for the original planning task, we need to consider how close its certainty equivalent is to the certainty equivalent of the optimal plan, not how close their average utilities are (Section 4.1). The following table summarizes this data.

	probability of goal achievement for the transformed planning task	certainty equivalent for the original planning task
optimal plan	p	$\log_\gamma p$
found plan (worst case)	$p - \epsilon$	$\log_\gamma [p - \epsilon]$

Thus, the resulting error for the original planning task, $\log_\gamma p - \log_\gamma [p - \epsilon] = \log_\gamma \frac{p}{p - \epsilon}$, increases as p and γ decrease. It can get arbitrarily large. To summarize, the reason for this is that we have to compare certainty equivalents for the original planning task and they correspond to the logarithms of the probability of goal achievement for the transformed planning task, and taking the logarithm can amplify the error.

Now assume that the utility function is convex exponential and consider a suboptimal planner with relative error $\epsilon > 0$.

	probability of goal achievement for the transformed planning task	certainty equivalent for the original planning task
optimal plan	p	$\log_\gamma p$
found plan (worst case)	$(1 - \epsilon)p$	$\log_\gamma [1 - \epsilon] + \log_\gamma p$

Thus, the resulting error for the original planning task, $\log_\gamma p - \log_\gamma [1 - \epsilon] - \log_\gamma p = \log_\gamma \frac{1}{1 - \epsilon}$, is additive. Figure 4.17 shows its graph.

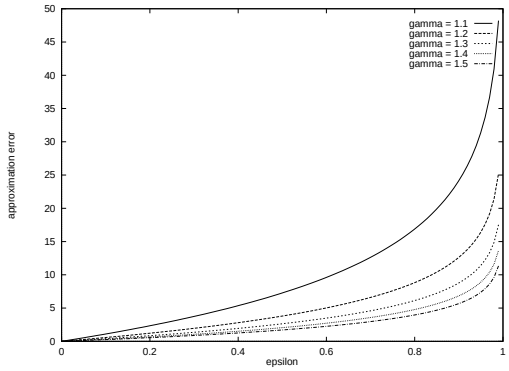


Figure 4.17: Error for Convex Exponential Utility Functions

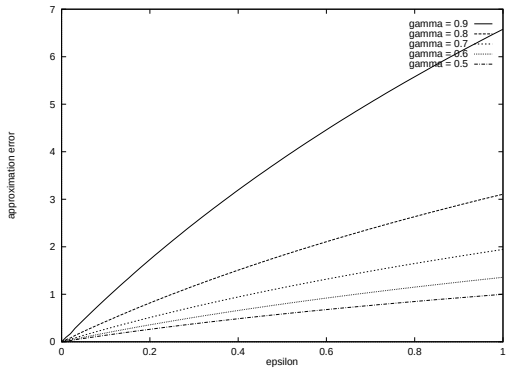


Figure 4.18: Error for Concave Exponential Utility Functions

Now assume that the utility function is concave exponential and consider a suboptimal planner with additive error $\epsilon > 0$.

	probability of goal achievement for the transformed planning task	certainty equivalent for the original planning task
optimal plan	p	$\log_\gamma p$
found plan (worst case)	$p + \epsilon$	$\log_\gamma [p + \epsilon]$

Thus, the resulting error for the original planning task, $\log_\gamma p - \log_\gamma [p + \epsilon] = \log_\gamma \frac{p}{p+\epsilon}$, increases as p decreases and γ increases. It can get arbitrarily large.

Finally, assume that the utility function is concave exponential and consider a suboptimal planner with relative error $\epsilon > 0$.

	probability of goal achievement for the transformed planning task	certainty equivalent for the original planning task
optimal plan	p	$\log_\gamma p$
found plan (worst case)	$(1 + \epsilon)p$	$\log_\gamma [1 + \epsilon] + \log_\gamma p$

Thus, the resulting error for the original planning task, $\log_\gamma p - \log_\gamma [1 + \epsilon] - \log_\gamma p = \log_\gamma \frac{1}{1+\epsilon}$, is additive. Figure 4.18 shows its graph.

4.6 Extension: Cyclic Plans

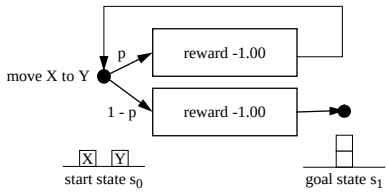


Figure 4.19: Plan for Stacking Two Blocks

So far, we studied planning tasks with plans that have the following property: the rewards of all their chronicles are finite. Now we drop this standard assumption in utility theory [Toulet, 1986] and consider planning tasks with *cyclic plans*. A plan is cyclic if a state can be repeated with positive probability during its execution. Consider, for example, a plan for stacking two blocks with a move action that takes one minute to execute and fails with probability $p > 0$ (Figure 4.19). If executing the move action more than once leads to results that are probabilistically independent, then the execution of this plan leads with probability $p^i(1-p)$ to a chronicle with reward $-i-1$ for all integers $i \geq 0$. Thus, the rewards of its chronicles are unbounded. Furthermore, only cyclic plans are able to solve the stacking task if absolutely reliable move actions are not available. In this section, we therefore show how to find plans that achieve the goal with maximal average utility (preference model C5) if the utility function is exponential and the plans can be cyclic. We assume that the exponential utility function is convex and all rewards are negative. This is, for example, the case for planning with immediate soft deadlines and for risk-seeking planning with resource consumptions. We discuss the other cases at the end of this section.

We first show how to use goal-directed Markov decision process models to model planning tasks with cyclic plans. We then show how to apply the multiplicative planning-task transformation to them. The transformed planning tasks can be solved with any planner that has preference model C3 *and is able to consider only plans that achieve the goal for the original planning task.* Next, we show how this restriction can be removed. This makes it possible to solve the transformed planning task with *any* planner with preference model C3. We then demonstrate the multiplicative planning-task transformation on a simple probabilistic blocks-world planning task. Finally, we present an interesting interpretation for discounting that the multiplicative planning-task transformation provides for goal-directed Markov decision process models.

4.6.1 Modeling the Planning Task

Convenient models of planning tasks with cyclic plans are *goal-directed Markov decision process models* (GDMDPs) [Koenig, 1991]. These are totally observable Markov decision process models (MDPs) for which a given goal state has to be achieved. For example, we have already used GDMDPs in Section 2.5.5 to express goal-directed reinforcement-learning tasks and goal-directed exploration tasks. GDMDPs are also the kinds of models that the path-planning layer of the mobile-robot system from Chapter 3 uses. They have also been used for robot navigation in [Dean *et al.*, 1993]. Notice that we use GDMDPs only to describe the planning tasks, not necessarily to represent or solve them. In fact, our statements hold for all planners that solve GDMDPs independent of the representations or planning methods that they use.

GDMDPs consist of a finite set of states S; a start state $s_{start} \in S$; and a set of goal states $G \subseteq S$. Each state $s \in S \setminus G$ has a finite set of actions $A(s) \neq \emptyset$ that can be executed in s. GDMDPs further consist of a transition function p (a function from $S \times A$ to probability distributions over S), where $p(s'|s,a)$ denotes the transition probability that the system transitions from state s to state s' when action a is executed; and an immediate reward function (a function from $S \setminus G \times A \times S$ to the real numbers), where $r(s,a,s')$ denotes the finite immediate reward received when the execution of action a in state s leads to state s'.

A *GDMDP process* is a stream of <state, action, immediate reward> triples: The process is always in exactly one state and makes state transitions at discrete time steps. It starts in state s_{start}. When it is

in a goal state, execution stops and the process does not generate any further rewards. When it is in a nongoal state s, a decision maker chooses an action $a \in A(s)$ for execution. The action execution leads with probability $p(s'|s,a)$ to immediate reward $r(s,a,s')$ and a transition to state s'. The decision maker knows the specification of the GDMDP and observes the actions and states.

Notice a difference to the earlier sections. Here, all non-goal state have at least one action defined. Thus, execution can stop only in goal states. This is sufficient to model planning tasks since preference model C5 requires one to find a plan that achieves the goal, and therefore stopping in non-goal states cannot be optimal. In earlier sections, it was possible for non-goal states not to have any action defined, and execution had to stop in these states. This assumption was necessary to satisfy our earlier requirement that all plans be acyclic. Since we now discuss cyclic plans, it is no longer necessary.

We define plans to be mappings from states to actions (also known as "stationary, deterministic policies"). A plan always selects the action for execution that it has assigned to the current state of the GDMDP process. A plan achieves the goal if the probability that it terminates in a goal state within a given number of action executions approaches one as the bound approaches infinity, otherwise it does not achieve the goal.

GDMDPs are special cases of totally observable Markov decision process models. Notice that, in Chapter 3, we used immediate rewards of the form $r(s,a)$ to describe MDPs instead of immediate rewards of the form $r(s,a,s')$. This is so because the immediate rewards $r(s,a)$ can be interpreted as the average immediate rewards obtained when executing action a in state s:

$$r(s,a) := \sum_{s' \in S} [p(s'|s,a) r(s,a,s')].$$

For planning with linear utility functions (that is, average reward) the immediate rewards $r(s,a)$ summarize the information that is contained in the immediate rewards $r(s,a,s')$. To see this, consider Bellman's equation (Section 3.2.2, Formula 3.3) with the immediate rewards $r(s,a,s')$ instead of the immediate rewards $r(s,a)$. In the following equations, γ specifies the discount factor rather than the parameter of the exponential utility function. (Section 4.6.4 explains why we use the same symbol for both entities.)

$$\begin{array}{lcll} v(s) & = & \max_{a \in A(s)} \sum_{s' \in S} [p(s'|s,a)(r(s,a,s') + \gamma v(s'))] & \text{for all } s \in S \\ v(s) & = & \max_{a \in A(s)} [\sum_{s' \in S} [p(s'|s,a) r(s,a,s')] + \gamma \sum_{s' \in S} [p(s'|s,a) v(s')]] & \text{for all } s \in S \\ v(s) & = & \max_{a \in A(s)} [r(s,a) + \gamma \sum_{s' \in S} [p(s'|s,a) v(s')]] & \text{for all } s \in S. \end{array}$$

A similar transformation cannot be performed for nonlinear utility functions. This is why we have to work directly with the immediate rewards $r(s,a,s')$.

4.6.2 Applying the Multiplicative Planning-Task Transformation

If all actions of the GDMDP are "deterministic" in the sense that their execution always ends in the same state (which can be different for different actions), then the optimal plan is not cyclic. Thus, we need to consider only acyclic plans and can apply the additive planning-task transformation unchanged (Section 4.5.1). In the following, we study the case where some actions of the GDMDP can end in more than one state. Thus, we need to apply the multiplicative planning-task transformation. We show that the multiplicative planning-task transformation applies unchanged to GDMDPs [Koenig and Simmons, 1994a]. The transformed planning task can be solved with any planner that has preference model C3 and is able to consider only plans that achieve the goal for the original planning task. The restriction is due to the fact that a plan with maximal average utility for a convex exponential utility function does not necessarily achieve the goal even if the goal can be achieved. This is different from maximizing average reward: a plan with maximal average reward

always achieves the goal if the goal can be achieved since all immediate rewards are negative. However, we show that the transformed planning task can be solved with *any* planner with preference model C3 if one first deletes all states from the original planning task from which the goal cannot be achieved.

We use the multiplicative planning-task transformation from Section 4.5.2 unchanged on the GDMDP. It transforms the GDMDP by modifying all of its actions (everything else remains the same): If the execution of action $a \in A(s)$ leads with probability $p(s'|s,a)$ to immediate reward $r(s,a,s')$ and state s' (for all $s' \in S$), then it is replaced with an action $a \in A(s)$ whose execution leads with probability $p(s'|s,a)\gamma^{r(s,a,s')}$ to state s' (for all $s' \in S$) and with probability $1 - \sum_{s' \in S}[p(s'|s,a)\gamma^{r(s,a,s')}]$ to a new nongoal state ("death") in which execution stops. The immediate rewards do not matter. The transformation is such that a plan that maximizes the probability of goal achievement for the transformed planning task also maximizes the average utility for the original planning task.

Notice the difference between plans for the original and transformed planning tasks: The execution of every plan for the original planning task stops only in the goal states but can cycle forever with positive probability. The execution of every plan for the transformed planning task stops either in the goal states or the nongoal state "death" but cannot cycle forever with positive probability. This difference is only for convenience. It does not change the nature of the planning tasks. Instead of stopping the execution of a plan for the transformed planning task once it reaches "death," we could also let it cycle forever in "death."

In the following, we explain why the multiplicative planning-task transformation works for GDMDPs.

Consider any plan that maps state s to action $a(s)$. We first calculate the average utility of the plan for the original planning task, then its probability of goal achievement for the transformed planning task, and finally show that these two values are identical. Notice that the probability of goal achievement of a plan for the transformed planning task is the same as the probability of not dying during its execution since its execution never stops in nongoal states other than "death."

We now calculate the average utility of the plan for the original planning task. Let $eu(s)$ be the average utility of state s. This is the average utility that is obtained for the original planning task if the execution of the plan starts in state s. Thus, the average utility of the plan is $eu(s_{start})$. To calculate it, we derive Bellman's equation for the average utilities of all states. To calculate the average utility of a goal state s, notice that the execution stops in the state and no further rewards are obtained. Thus,

$$eu(s) = u(0) = \gamma^0 = 1.$$

To calculate the average utility of a nongoal state s, notice that the execution of action $a(s)$ leads with probability $p(s'|s,a(s))$ to immediate reward $r(s,a(s),s')$ and a transition to state s'. In state s', the agent faces a lottery with certainty equivalent $ce(s')$. For exponential utility functions, the certainty equivalent of first receiving reward $r(s,a(s),s')$ and then participating in a lottery with certainty equivalent $ce(s')$ is, according to Formula 4.2, $r(s,a(s),s') + ce(s')$. The corresponding average utility is $u(r(s,a(s),s') + ce(s'))$. Thus,

$$\begin{aligned}
eu(s) &= \sum_{s' \in S}[p(s'|s,a(s))u(r(s,a(s),s') + ce(s'))] \\
&= \sum_{s' \in S}[p(s'|s,a(s))\gamma^{r(s,a(s),s')+ce(s')}] \\
&= \sum_{s' \in S}[p(s'|s,a(s))\gamma^{r(s,a(s),s')}\gamma^{ce(s')}] \\
&= \sum_{s' \in S}[p(s'|s,a(s))\gamma^{r(s,a(s),s')}u(ce(s'))] \\
&= \sum_{s' \in S}[p(s'|s,a(s))\gamma^{r(s,a(s),s')}eu(s')]
\end{aligned}$$

$$= \sum_{s' \in S} [\overline{p}(s'|s, a(s)) eu(s')],$$

where the parameters $\overline{p}(s'|s, a)$ are new values with $\overline{p}(s'|s, a) := p(s'|s, a)\gamma^{r(s,a,s')}$. This result corresponds to the plan-evaluation step in [Howard and Matheson, 1972] with the "certain equivalent gain" $\overline{g} = 0$. The values $\overline{p}(s'|s, a)$ satisfy $0 \leq \overline{p}(s'|s, a) \leq p(s'|s, a)$ according to our assumption that the exponential utility function is convex ($\gamma > 1$) and all immediate rewards are negative ($r(s, a, s') < 0$).

To summarize, Bellman's equation for the average utilities $eu(s)$ of the states is

$$eu(s) = \begin{cases} 1 & \text{if } s \in G \\ \sum_{s' \in S} [\overline{p}(s'|s, a(s)) eu(s')] & \text{otherwise} \end{cases} \qquad \text{for all } s \in S. \tag{4.6}$$

That this set of equations has a unique solution, which corresponds to the average utilities of the states, follows directly from [Denardo and Rothblum, 1979], an application of [Howard and Matheson, 1972] to stopping problems that also considers the case of mixed negative and positive immediate rewards.

Instead of calculating the average utility of the plan directly, we can first transform it: Its structure remains unchanged but all of its actions are transformed (as described above): If the execution of action $a \in A(s)$ leads with probability $p(s'|s, a)$ to immediate reward $r(s, a, s')$ and state s' (for all $s' \in S$), then it is replaced with an action $a \in A(s)$ whose execution leads with probability $p(s'|s, a)\gamma^{r(s,a,s')} = \overline{p}(s'|s, a)$ to state s' (for all $s' \in S$) and with probability $1 - \sum_{s' \in S} [p(s'|s, a)\gamma^{r(s,a,s')}] = 1 - \sum_{s' \in S} \overline{p}(s'|s, a)$ to a new nongoal state ("death") in which execution stops. The immediate rewards do not matter.

We now calculate the probability of goal achievement of the plan for the transformed planning task. Let $pga(s)$ be the probability of goal achievement of state s (pga = probability of goal achievement). This is the probability of goal achievement for the transformed planning task if the execution of the plan starts in state s. Thus, the probability of goal achievement of the plan is $pga(s_{start})$. To calculate it, we derive Bellman's equation for the probabilities of goal achievement of all states. To calculate the probability of goal achievement of a goal state s, notice that the execution stops in goal states. Thus,

$$pga(s) = 1.$$

To calculate the probability of goal achievement of a nongoal state s, notice that the execution of action $a(s)$ leads with probability $\overline{p}(s'|s, a(s))$ to state s' and with probability $1 - \sum_{s' \in S} \overline{p}(s'|s, a(s))$ to "death." The probability of goal achievement of state s' is $pga(s')$. The probability of goal achievement of "death" is zero since execution stops but "death" is not a goal state. Thus,

$$\begin{aligned} pga(s) &= \sum_{s' \in S} [\overline{p}(s'|s, a(s)) pga(s')] + (1 - \sum_{s' \in S} \overline{p}(s'|s, a(s))) \times 0 \\ &= \sum_{s' \in S} [\overline{p}(s'|s, a(s)) pga(s')]. \end{aligned}$$

To summarize, Bellman's equation for the probabilities of goal achievement $pga(s)$ of the states is

$$pga(s) = \begin{cases} 1 & \text{if } s \in G \\ \sum_{s' \in S} [\overline{p}(s'|s, a) pga(s')] & \text{otherwise} \end{cases} \qquad \text{for all } s \in S. \tag{4.7}$$

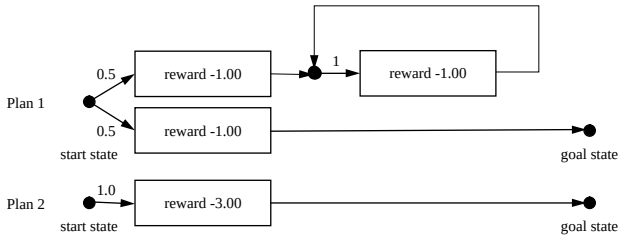


Figure 4.20: Example of a Plan that Maximizes Average Utility but does not Achieve the Goal

That this set of equations has a unique solution, which corresponds to the probabilities of goal achievement of the states, follows directly from [Mine and Osaki, 1970].

A comparison of Formulae 4.6 and 4.7 shows that $eu(s) = pga(s)$ for all $s \in S$. Consequently, the probability of goal achievement $pga(s_{start})$ of a plan for the transformed planning task is the same as its average utility $eu(s_{start})$ for the original planning task.

The multiplicative planning-task transformation can be used to find a plan with maximal average utility for the original planning task: We first transform all actions of the planning task. Then, a plan that maximizes the probability of goal achievement for the transformed planning task also maximizes the average utility for the original planning task. Preference model C5, however, requires one to find a plan that *achieves the goal* with maximal average utility. This complicates matters but in the following we show that the multiplicative planning-task transformation can be used for this purpose as well: one only has to remove all states from the original planning task from which the goal cannot be achieved.

We first demonstrate that a plan that maximizes the average utility for the original planning task does not necessarily achieve the goal. As an example, consider the two plans from Figure 4.20 and assume that the utility function is $u(r) = 2^r$. Then, the average utility of Plan 1 is $0.50u(-\infty) + 0.50u(-1) = 0.25$ and the (average) utility of Plan 2 is $1.00u(-3) = 0.125$. Thus, Plan 1 has a larger average utility than Plan 2, but only Plan 2 achieves the goal.

To solve this problem, we have to make sure that the plan that maximizes the average utility for the original planning task also achieves the goal. We say that a goal can be achieved from state s if a plan exists that achieves the goal if its execution starts in state s. We then use the following property:

Theorem 24 *A plan with maximal average utility for a GDMDP planning task always achieves the goal if the exponential utility function is convex, all immediate rewards are negative, and the goal can be achieved from all states.*

Proof by contradiction: Suppose that there is a plan ps with maximal average utility that does not achieve the goal. Since the goal can be achieved from all states, there must be some state s that is reached with positive probability during the execution of plan ps such that plan ps' differs from plan ps in only the action assigned to s, plan ps' reaches a goal state with positive probability from state s, and plan ps reaches a goal state with probability zero from state s. To see this, consider the set of all states that are reached with positive probability during the execution of plan ps and from which plan ps reaches a goal state with probability zero. At least one such state exists and all of them are nongoal states. The statement then follows for one of these states, which we called s, since a goal can be achieved from all of those states.

Now consider all chronicles of plan ps that do not contain state s. Plan ps' has the same chronicles with the same probabilities and utilities. The rewards of all chronicles of plan ps that contain state s are minus infinity (since all immediate rewards are negative and the chronicles do not stop) and their utilities are zero (since the exponential utility function is convex). On the other hand, at least one chronicle of plan ps' that contains

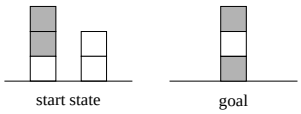


Figure 4.21: Blocks-World Planning Task

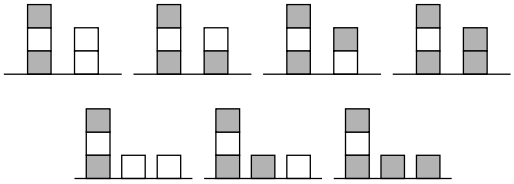


Figure 4.22: Goal States for the Blocks-World Planning Task

state s achieves the goal. Its reward is finite and its utility is positive. The utilities of the other chronicles of plan ps' that contain state s are nonnegative (since the exponential utility function is convex). Therefore, the average utility of plan ps' is strictly larger than that of plan ps. This, however, is a contradiction. ■

To summarize, if the goal can be achieved from all states of the original planning task and a plan achieves the goal with maximal probability for the transformed planning task, then it also maximizes the average utility for the original planning task and thus also achieves the goal for the original planning task. Consequently, the original planning task can be solved by applying the multiplicative planning-task transformation and then solving the transformed planning task with *any* planner with preference model C3. The transformed planning task can also be solved with *any* planner with preference model C4 by declaring "death" another goal state and making the immediate rewards for stopping in goal states other than "death" one and all other immediate rewards zero. This is so because no plan for the transformed planning task can cycle forever with positive probability. Thus, it always achieves an original goal state or "death." Notice that the planner with preference model C4 has to be able to handle immediate rewards that are zero.

If the goal cannot be achieved from all states of the original planning task, one can remove all states from which the goal cannot be achieved (and all actions whose execution can lead to these states) before applying the multiplicative planning-task transformation. The states can be removed, for example, with the method described in [Koenig, 1991]. Their removal does not eliminate plans that achieve the goal and it does not change the average utility of any plan. It only eliminates some plans that do not achieve the goal. Thus, the removal does not affect which plan achieves the goal with maximal average utility. It does, however, guarantee that the plan with maximal average utility also achieves the goal (Theorem 24). This plan also achieves the goal with maximal average utility before the removal of the states. This makes it possible to solve the transformed planning task with methods that are able to maximize the probability of goal achievement, but are not able to consider only plans that achieve the goal for the original planning task. An example is value-iteration (Section 3.2.2).

4.6.3 Example: A Blocks-World Planning Task

In this section, we demonstrate the multiplicative planning-task transformation on a simple block-world planning task with five blocks that are either black or white [Koenig and Simmons, 1994b]. The spatial relationships among the stacks are unimportant, resulting in a state space that contains 162 states. In every blocks-world state, one can paint a block white or black or move a block that has a clear top onto either the table or a different block that has a clear top. The results of action executions are probabilistically independent. Painting a block takes three minutes and is always successful. Moving a block takes only one minute, but it is

very unreliable. With probability 0.10, the moved block ends up at its intended destination. With probability 0.90, however, the gripper loses the block and it ends up directly on the table. Moving a block to the table always succeeds. Figure 4.21 shows the start state and the goal, which is to build a stack of three blocks that contains a black block, a white block, and a black block, in this order. Consequently, there are seven goal states that differ in the configuration and color of the two remaining blocks. All of them are equally preferable (Figure 4.22).

Plans of the blocks-world planning task can have cycles. For example, the state remains unchanged if moving a block from the table onto another block fails. This also shows that not all actions end in the same state. We therefore represent the blocks-world planning task as a GDMDP and apply the multiplicative planning-task transformation. Instead of explicitly enumerating all transition probabilities and immediate rewards, we make use of the fact that the multiplicative planning-task transformation works on various representations and use a more compact representation of GDMDPs, namely, probabilistic STRIPS rules.

The original STRIPS notation [Fikes and Nilsson, 1971] was designed to specify planning tasks in deterministic domains. Probabilistic STRIPS rules [Koenig, 1991] augment them in a straightforward way to probabilistic domains such as GDMDPs. For example, the move action of the blocks-world planning task can be modeled with three probabilistic STRIPS rules, one for moving block X from block Y to block Z, one for moving block X from the table to block Y, and one for moving block X from block Y to the table. As an example, the first of these rules is:

```
RULE 1: move(X,Y,Z)
  precond:  on(X,Y), clear(X), clear(Z), notequal(X,Z)
  outcome:  /* the intended outcome */
    prob:   0.10
    reward: -1.00
    delete: on(X,Y), clear(Z)
    add:    on(X,Z), clear(Y)
  outcome:  /* failure: block X falls onto the table */
    prob:   0.90
    reward: -1.00
    delete: on(X,Y)
    add:    clear(Y), ontable(X)
```

Additional STRIPS rules describe the start state and the goal states.

The multiplicative planning-task transformation then converts the probabilistic STRIPS rules. Assume, for example, that the utility function is $u(r) = 2^r$, where r is the negative plan execution time, measured in minutes. Then, the transformation of the probabilistic STRIPS rule shown above is:

```
TRANSFORMED RULE 1: move(X,Y,Z)
  precond:  alive, on(X,Y), clear(X), clear(Z), notequal(X,Z)
  outcome:  /* the intended outcome */
    prob:   0.05
    delete: on(X,Y), clear(Z)
    add:    on(X,Z), clear(Y)
  outcome:  /* failure: block X falls onto the table */
    prob:   0.45
    delete: on(X,Y)
    add:    clear(Y), ontable(X)
  outcome:  /* death */
    prob:   0.50
    delete: alive
    add:    dead
```

The start and goal states remain unchanged.

Notice that the goal of the original blocks-world planning task can be achieved from every state: one can first unstack every block, then build a stack of three blocks (repeating the move actions until they finally succeed), and finally paint the blocks. Thus, a plan that maximizes the probability of goal achievement for the transformed blocks-world planning task also achieves the goal with maximal average utility for the

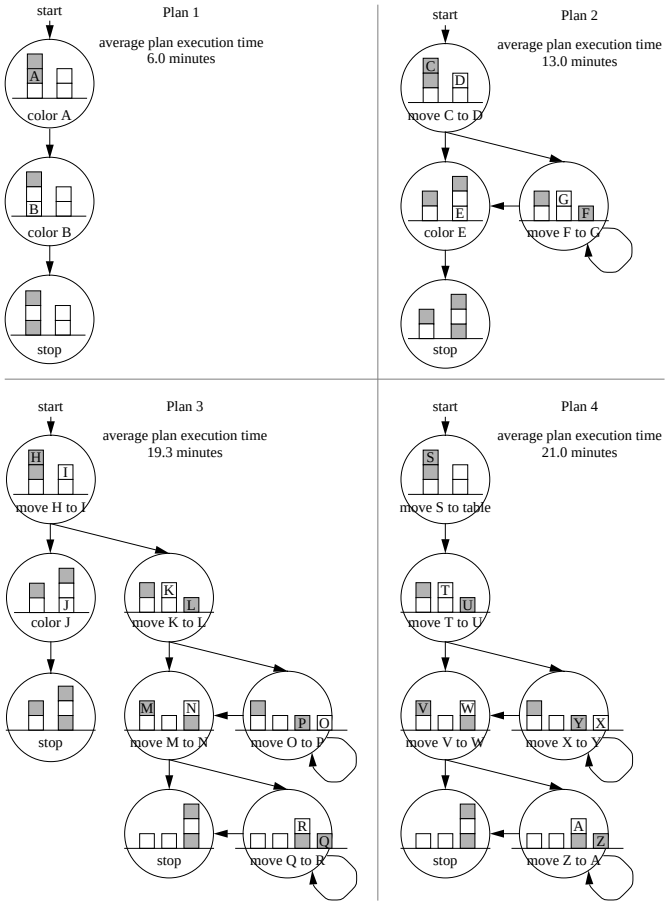


Figure 4.23: Some Plans for the Blocks-World Planning Task

original blocks-world planning task. In the following, we describe the results of planning for the blocks-world planning task.

Figure 4.23 shows four plans that solve the blocks-world planning task. The purpose of the symbols A, B, C, and so on, is to enable the action commands to refer to specific blocks. Thus, the symbols do not necessarily mark the same blocks in different states. Figure 4.24 illustrates how the certainty equivalents of the plans vary with the natural logarithm of γ for the convex exponential utility function $u(r) = \gamma^r$, where $\gamma > 1$ and r is the negative plan-execution time, measured in minutes.

Plan 1 is deterministic and takes six minutes to execute. Thus, its (average) reward is -6.00, and this is the maximal average reward of all plans, not just the four plans shown in Figure 4.23. However, Plan 1 does not necessarily maximize the certainty equivalent. Its certainty equivalent equals its reward for all values of γ since Plan 1 is deterministic. The other three plans are probabilistic. Thus, their certainty equivalents increase as γ increases, and different plans can be optimal for different γ. Figure 4.24 shows that Plan 1 is optimal in the interval $\ln\gamma \in (0.00, 0.93]$. For $\ln\gamma \in [0.94, 4.58]$, Plan 3 is optimal, and Plan 4 should be chosen for $\ln\gamma \in [4.59, \infty)$. These statements hold for all plans, not just the four plans shown in the picture.

As γ approaches one and $\ln\gamma$ approaches zero, the certainty equivalent of each plan approaches its average reward. The certainty equivalent of Plan 4, for example, approaches -21.00. Thus, Plan 1 is optimal for γ approaching one. In contrast, as γ approaches infinity, the certainty equivalent of each plan approaches its best-case reward. When executing Plan 4, for example, the agent can reach a goal state in only three minutes if it is lucky. Thus, the certainty equivalent of Plan 4 approaches -3.00, and this is the best-case reward of all

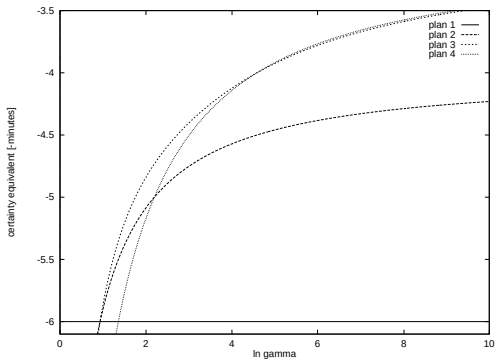


Figure 4.24: Certainty Equivalents of the Four Blocks-World Plans

plans. Thus, Plan 4 is optimal for γ approaching infinity. The certainty equivalent of Plan 3 also approaches -3.00 and thus Plan 3 is also optimal for γ approaching infinity. However, for $\ln\gamma \geq 4.59$, the certainty equivalent of Plan 4 is always larger than the certainty equivalent of Plan 3.

4.6.4 Discounting

The path-planning layer of the mobile-robot system from Chapter 3 maximizes the average *discounted* reward for a given GDMDP planning task. Recall that the discount factor $0 < \gamma \leq 1$ specifies the relative value of an immediate reward after t action executions compared to the same immediate reward one action execution earlier. In this section, we first review two standard interpretations of discounting, namely, that it models agents that can save and borrow resources at a given interest rate and agents that die with a given probability after every action execution. We then use the multiplicative planning-task transformation to provide a novel interpretation of discounting, which is often more useful, namely, that, in certain situations, discounting can be interpreted as a special case of calculating the average utility for nonlinear utility functions. In particular, we show that maximizing the average discounted reward for GDMDP planning tasks with the goal-reward representation is the same as maximizing the average utility for the same GDMDP planning task with the action-penalty representation and a convex exponential utility function. (The first planning task can be obtained by applying the multiplicative planning-task transformation to the second planning task.) Consequently, maximizing average utility for exponential utility functions generalizes the concept of discounting. This explains why we chose to use the symbol γ for both the discount factor and the parameter of exponential utility functions. Many researchers have formulated their planning tasks as GDMDPs with the goal-reward representation and then maximized the average discounted reward to solve it. An example in the context of robot navigation is [Lin, 1993]. Other examples include [Sutton, 1990, Whitehead, 1991a, Peng and Williams, 1992, Thrun, 1992b]. Consequently, these researchers have already used a special case of the multiplicative planning-task transformation without knowing it.

Discounting has nice mathematical properties (it ensures that the average total reward is always finite even for cyclic plans that never terminate) and was originally motivated by collecting interest for money. If an agent receives immediate reward $r \neq 0$ at time t and the interest rate is $i = (1-\gamma)/\gamma \geq 0$, then the immediate reward is worth $(1+i)r = r/\gamma$ at time $t+1$. Thus, the discount factor is the relative value at time $t+1$ of an immediate reward r received at time $t+1$ compared to the same immediate reward received at time t:

$$0 < \frac{r}{r/\gamma} = \gamma \leq 1.$$

Consequently, the discount factor γ can be interpreted as modeling agents that can save or borrow resources at interest rate $(1-\gamma)/\gamma$.

Robots usually cannot invest their resources and earn interest. Discount factors are then often interpreted as taking the possibility of death of the agent into account. When an agent dies, it cannot collect any further rewards. Thus, if it dies with probability $0 \leq 1 - \gamma < 1$ between time t and time $t + 1$, then it cannot collect the immediate reward $r \neq 0$ at time $t + 1$ and thus the average value of this immediate reward at time t is $\gamma r + (1-\gamma)0 = \gamma r$. Thus, the discount factor is the relative value at time t of an immediate reward r received at time $t + 1$ compared to the same immediate reward received at time t:

$$0 < \frac{\gamma r}{r} = \gamma \leq 1.$$

Consequently, the discount factor γ can be interpreted as modeling agents that die with probability $1 - \gamma$ after every action execution. The navigation example for outdoor rovers that we discussed in Section 4.2.1 can be interpreted in this way.

In the following, we provide a novel interpretation for discounting. Recall from Section 2.5.6.1 that the goal-reward representation rewards the agent for stopping in a goal state, but does not reward or penalize it for executing actions. The action-penalty representation penalizes the agent for every action that it executes, but does not reward or penalize it for stopping in a goal state. We consider any GDMDP planning task whose start state is not a goal state. (The other GDMDP planning tasks are trivial.) We show that the average discounted reward of any plan for the GDMDP planning task with the goal-reward representation and discount factor γ^{-1} (with $0 < \gamma^{-1} < 1$) is proportional to its average utility for the same GDMDP planning task with the action-penalty representation and exponential utility function $u(r) = \gamma^r$ (with $\gamma > 1$).

To see this, consider any plan and any chronicle of the plan. Assume that the chronicle contains i actions. Since the start state is not a goal state and the execution stops only in goal states it must be that $i > 1$ and, if i is finite, the chronicle must end in a goal state.

Now consider the GDMDP planning task with the goal-reward representation and discount factor γ^{-1}. The goal-reward representation uses the following immediate rewards

$$r(s, a, s') = \begin{cases} 1 & \text{if } s' \in G \\ 0 & \text{otherwise} \end{cases} \qquad \text{for all } s \in S \setminus G, a \in A(s), \text{ and } s' \in S.$$

Then, the discounted reward of the chronicle is $(\gamma^{-1})^{i-1} = \gamma^{-i+1}$.

Now consider the same GDMDP planning task with the action-penalty representation and exponential utility function $u(r) = \gamma^r$. The action-penalty representation uses the following immediate rewards

$$r(s, a, s') = -1 \qquad \text{for all } s \in S \setminus G, a \in A(s), \text{ and } s' \in S.$$

Then, the reward of the chronicle is $-i$ and its utility is γ^{-i}.

To summarize, the discounted reward of every chronicle for the GDMDP planning task with the goal-reward representation is γ times its utility for the same GDMDP planning task with the action-penalty representation. Notice that this also holds if the chronicle contains an infinite number of actions. In this case, its discounted reward is zero for the GDMDP planning task with the goal-reward representation, and its reward is minus infinity and its utility is zero for the GDMDP planning task with the action-penalty representation.

This means that the average discounted reward of every plan for the GDMDP planning task with the goal-reward representation is γ times its average utility for the same GDMDP planning task with the action-penalty representation. Consequently, a plan with maximal average discounted reward for the GDMDP planning task

with the goal-reward representation is also a plan with maximal average utility for the same GDMDP planning task with the action-penalty representation. Similarly, a plan that achieves the goal with maximal average discounted reward for the GDMDP planning task with the goal-reward representation is also a plan that achieves the goal with maximal average utility for the same GDMDP planning task with the action-penalty representation.

Recall from Section 2.5.6.3 that, in deterministic domains, maximizing the discounted reward for any planning task with the goal-reward representation leads to the same result as maximizing the undiscounted reward for the same planning task with the action-penalty representation. This is no longer true in probabilistic domains. We showed that maximizing the average discounted reward for any planning task with the goal-reward representation in probabilistic domains leads to the same result as maximizing the average (undiscounted) utility for the same planning task with the action-penalty representation and a convex exponential utility function (as shown above). The discount factor used determines the shape of the utility function, for example, the softness of an immediate soft deadline or the amount of risk-sensitivity. More generally, assume that we want to maximize the average utility of a GDMDP planning task with the action-penalty representation and a convex exponential utility function, but the immediate rewards are nonhomogeneous. The corresponding planning task with the goal-reward representation then has nonhomogeneous discounting since the discount factor $\gamma^{r(s,a,s')}$ depends on the state, the executed action, and the resulting successor state.

We feel that this interpretation of discounting applies to a larger class of agents than the other two interpretations since agents often have nonlinear utility functions but usually cannot earn interest on unused resources or have to be concerned about dying after every action execution.

4.6.5 Other Cases

We have shown how to find plans for GDMDPs that achieve the goal with maximal average utility if the exponential utility function is convex and all immediate rewards are negative. We now discuss the problems that the other cases pose for the multiplicative planning-task transformation – addressing these problems is future work. Planning tasks with positive immediate rewards have the problem that it is optimal to cycle for an arbitrarily long time before achieving the goal if a cycle with positive total reward exists that can be traversed with probability one. We therefore do not discuss the cases where the immediate rewards can be positive. The remaining case is the one where the exponential utility function is concave and all immediate rewards are negative. In this case, Bellman's equation for the average utilities $eu(s)$ of the states (Formula 4.6) is

$$eu(s) = \begin{cases} -1 & \text{if } s \in G \\ \sum_{s' \in S}[\overline{p}(s'|s,a(s))eu(s')] & \text{otherwise} \end{cases} \qquad \text{for all } s \in S, \tag{4.8}$$

where the parameters $\overline{p}(s'|s,a)$ are new values with $\overline{p}(s'|s,a) := p(s'|s,a)\gamma^{r(s,a,s')}$. The values $\overline{p}(s'|s,a)$ satisfy $p(s'|s,a) \leq \overline{p}(s'|s,a)$ according to our assumption that the exponential utility function is concave ($0 < \gamma < 1$) and all immediate rewards are negative ($r(s,a,s') < 0$). The values are nonnegative but their sum $\sum_{s' \in S} \overline{p}(s'|s,a)$ exceeds one – they are no longer probabilities. We commented on this already in the context of acyclic plans (Section 4.5.2). However, cyclic plans pose additional problems, some of which make utility theory break down and others make the multiplicative planning task transformation break down. We next discuss these problems. It remains future work to address the problem of planning with concave exponential utility functions when utility theory breaks down. When the multiplicative planning task transformation breaks down but utility theory does not, operations research has studied methods that are still able to find plans with maximal average utility. Thus, planning with exponential utility functions remains possible but it is future work to investigate how one can apply search and planning methods from artificial intelligence in this case.

We first describe a situation where utility theory breaks down.

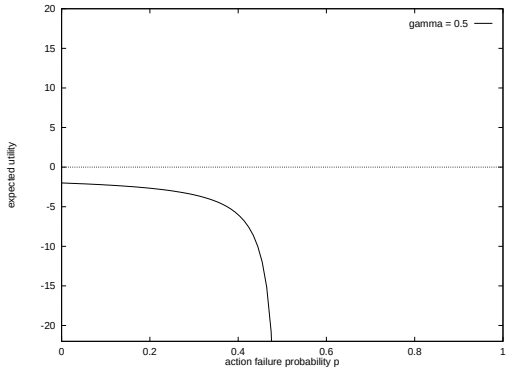


Figure 4.25: Average Utility of the Plan for Stacking Two Blocks

Scared Agents: For values of γ that are sufficiently close to zero, the average utilities of plans can be minus infinity even if the plans achieve the goal and therefore have a finite average reward. In this case, the average utilities of two plans that achieve the goal can both be minus infinity even if one plan dominates the other and thus should be preferred. (Therefore, for sufficiently risk-averse attitudes in high-stake one-shot planning domains, agents can become so "scared" of the risk that they no longer differentiate among plans of different goodness.) As an example, consider again the plan for stacking two blocks with a move action that takes one minute to execute and fails with probability p (Figure 4.19). The execution of this plan leads with probability $p^i(1-p)$ to a chronicle with reward $-i-1$ for all integers $i \geq 0$. The average utility of the plan is

$$\sum_{i=0}^{\infty}[p^i(1-p)u(-i-1)] = \sum_{i=0}^{\infty}[p^i(1-p)(-\gamma^{-i-1})] = -\frac{1-p}{\gamma}\sum_{i=0}^{\infty}\left(\frac{p}{\gamma}\right)^i = \begin{cases} -\frac{1-p}{\gamma-p} & \text{for } 0 \leq p < \gamma \\ -\infty & \text{for } \gamma \leq p \leq 1. \end{cases}$$

Figure 4.25 shows how the average utility of the plan depends on p for the concave exponential utility function $u(r) = -(1/2)^r$, where r is the negative plan-execution time, measured in minutes. The average utility of the plan is minus infinity for $\gamma \leq p$. If an agent can choose between a move action with failure probability p_1 and a different move action with failure probability p_2 where $\gamma \leq p_2 < p_1$, it cannot decide which one to prefer although it should clearly choose the latter one, since the probability distribution over the rewards of the latter move action dominates the probability distribution over the rewards of the former move action. The reason for this problem is that rewards can be unbounded for cyclic plans. We do not offer a solution for this problem in this thesis.

We now describe a situation where the multiplicative planning-task transformation breaks down.

Inadmissible Fixed Points: It is no longer guaranteed that a solution of Bellman's equation (Formula 4.8) corresponds to the average utility of the plan, because a finite solution should not always be preferred over an infinite solution. We give two examples:

First, consider a plan that achieves the goal with probability zero. Its reward is minus infinity (since all immediate rewards are negative) and its utility is minus infinity (since the exponential utility function is concave). However, the *finite* solution of Bellman's equation is zero for $eu(s_{start})$ of this plan.

Second, consider a plan that achieves the goal but has utility minus infinity ("scared agent"). As an example, consider again the plan for stacking two blocks with a move action that takes one minute to execute and fails with probability p. s_0 denotes the start state (both blocks on the table) and s_1 denotes the goal state (both blocks stacked). Bellman's equation for this plan is

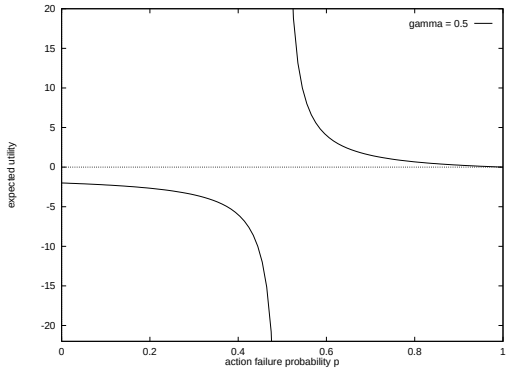


Figure 4.26: Solution of Bellman's Equation for the Plan for Stacking Two Blocks

$$\begin{aligned} eu(s_0) &= p\gamma^{-1}eu(s_0) + (1-p)\gamma^{-1}eu(s_1) \\ eu(s_1) &= -1. \end{aligned}$$

Its solutions are

$$\begin{aligned} eu(s_0) &= -\frac{1-p}{\gamma-p} \\ eu(s_1) &= -1. \end{aligned}$$

Figure 4.26 shows the solution $eu(s_0)$ of Bellman's Equation for the average utility of the plan, and Figure 4.25 showed its correct average utility. They differ for $\gamma < p$, when the *finite* solution of Bellman's Equation becomes erroneously nonnegative. (The point on the graph at $p = 1$ corresponds to the first example above.) Thus, if a planner used the finite solutions of Bellman's Equation unchanged to choose among several move actions with different failure probabilities, then it would pick a move action with failure probability $p > 0.5$ over a move action with failure probability $p = 0$.

Thus, not all planners with preference model C3 can be used unchanged on the transformed planning task. This does not mean that the planning task can no longer be solved. Operations research has investigated properties of these planning tasks and possible solution methods in the context of "risk-sensitive Markov decision process models" [Denardo and Rothblum, 1979, Hernandez-Hernandez and Marcus, 1997, Marcus *et al.*, 1997]. In this context, operations research has also investigated how to maximize the average utility for *partially observable* Markov decision process models if the utility functions are exponential. Examples include [Bensoussan and Schuppen, 1985, Whittle, 1990, Coraluppi and Marcus, 1996, Baras and James, 1997, Fernandez-Gaucherand and Marcus, 1997]. This means that planning with exponential utility functions is possible in these cases but it is future work to investigate how one can use search and planning methods from artificial intelligence.

4.7 Future Work

In this chapter, we studied preference models for acting with incomplete information where the consumption of only one limited resource needs to be considered. We studied exponential utility functions and described how to find plans efficiently that achieve the goal with maximal average utility (preference model C5) for this

class of nonlinear utility functions. While exponential utility functions are expressive, they cannot be used to solve all planning tasks since the number of their parameters, and thus their shapes, and the number of their variables are restricted.

Shape: Exponential utility functions can model immediate soft deadlines. However, they cannot model hard deadlines or deadlines in the future [Dean *et al.*, 1988, Haddawy and Hanks, 1990]. As an example of a hard deadline in the future is the delivery of a conference submission to a UPS stop. In this case, the utility of the delivery is u_0 if the delivery is completed before the last UPS pickup that meets the submission deadline and u_1 (with $u_1 < u_0$) afterwards. Similarly, exponential utility functions can model a continuum of risk attitudes in high-stake one-shot planning domains but not all risk attitudes. For example, they can model only agents that have pure risk attitudes but not agents who play the lottery *and* have insurance. Furthermore, they cannot model agents whose decisions depend on their wealth (that is, where their choices depend on how wealthy they are), such as agents who become less risk-averse as they become wealthier. The restricted shape of exponential utility functions appears to be a larger problem for modeling deadlines than risk attitudes because the range of relevant rewards often seems to be small enough so that only one risk attitude is present.

Number of Variables: Exponential utility functions depend on only one variable, namely, the consumption of one limited resource. Often, however, if more than one resource is scarce, agents have to reason about the trade-off between the consumption of these different resources. An outdoor delivery robot, for example, might have to worry about the delivery time *and* the rate of energy consumption, which depends on the ruggedness of the terrain and other factors.

These limitations of exponential utility functions demonstrate that it is important to integrate our planning methods with planning methods for other nonlinear utility functions and multi-variable utility functions. This would expand the applicability of our planning methods and, at the same time, increase the efficiency of the other planning methods on the parts of the planning tasks that can be modeled with exponential utility functions. This would also solve the problem that our planning methods only preserve the decomposability of planning tasks but do not create decomposability. Thus, they do not apply to planning tasks that are not decomposable (that is, do not satisfy the Markov assumption).

Another interesting problem for planning with nonlinear utility functions is how to schedule several tasks, where each task has its own utility function. This problem is a special case of having one utility function with more than one variable. It is relevant for delivery robots that have several delivery requests pending, each with its own deadline function.

Another interesting problem is how to combine utility models other than the one studied here with search and planning methods from artificial intelligence. One possible alternative is prospect theory [Kahneman and Tversky, 1979], that is able to explain some empirical findings about human decision making that utility theory cannot explain.

Another interesting problem is how to identify other opportunities for combining utility theory with search and planning methods from artificial intelligence. One such area appears to be multi-attribute utility theory [Keeney and Raiffa, 1976], which studies utility functions with more than one variable. PYRRHUS [Williamson and Hanks, 1994, Williamson and Hanks, 1996] is a first step in this direction. This planner uses a subset of the nonlinear, multi-attribute utility functions in [Haddawy and Hanks, 1992, Haddawy and Hanks, 1993] for planning in deterministic domains. Other steps in this direction are [Wellman and Doyle, 1992], which exploits the hierarchical structure of multi-attribute utility functions, and DRIPS [Haddawy *et al.*, 1995], a decision-theoretic refinement planner that exploits information provided in an abstraction hierarchy.

4.8 Conclusions

Many existing search and planning methods from artificial intelligence attempt to find plans with maximal probability of goal achievement or plans that achieve the goal with minimal average execution cost, but often one wants to find plans that achieve the goal with maximal average utility for nonlinear utility functions.

We have shown that planning tasks with nonlinear utility functions often are not decomposable, even if the corresponding planning tasks with linear utility functions (that is, for minimizing average cost) are decomposable. To maintain decomposability, and thus efficiency, we use exponential utility functions. Our planning methods combine constructive approaches from artificial intelligence with more descriptive approaches from utility theory. Planning is done via representation changes: The additive planning-task transformation applies to planning tasks that can be solved with sequential plans (more precisely: planning tasks whose actions have deterministic outcomes but whose rewards can be nondeterministic) and transforms them to deterministic planning tasks that require the minimization of cost. The multiplicative planning task transformation applies to planning tasks that can be solved with conditional plans and transforms them to planning tasks that require the maximization of the probability of goal achievement or minimization of the average cost. In both cases, the better the plan for the transformed planning tasks, the better it is for the original planning tasks as well. Thus, optimal or near-optimal plans for the original planning tasks can be found by solving the transformed planning tasks with standard deterministic or probabilistic planning methods, including those that do not reason about costs at all.

To summarize the properties of our planning-task transformations, we describe their advantages and disadvantages. Disadvantages are that the planning-task transformations are restricted to exponential utility functions. Thus, they cannot handle nonlinear utility functions of arbitrary shape or with multiple variables. Also, special cases of planning tasks cause problems for the multiplicative planning-task transformation: If the exponential utility functions are convex and not all costs are positive, or the exponential utility functions are concave and not all costs are negative, then the multiplicative planning-task transformation creates "probabilities" whose sum can be larger than one. Furthermore, concave exponential utility functions can pose problems for the multiplicative planning-task transformation in cyclic domains: not all planners can solve the transformed planning tasks, and in some cases the average utility of a plan is not even well defined.

We classify the advantages of our planning-task transformations into three categories.

Advantages of exponential utility functions are that they maintain the delta property (Markov property) and are expressive. For example, they can model immediate soft deadlines and model a continuum of risk attitudes in high-stake one-shot planning domains. Side benefits of studying exponential utility functions include that they help us to better understand the role of discounting and the relationship between minimizing worst-case (minimax principle), average, and best-case costs.

Advantages of the planning-task transformations are that they are simple context-insensitive representation changes of the planning tasks. They are fast and scale well since their running time is linear in the sizes of the planning-task representations. They can be performed on a variety of planning-task representations without changing their kind or size. Finally, the planning-task transformations can be used as black-box methods (that is, they can be applied without an understanding of how or why they work).

Advantages of using traditional planners to solve the transformed planning tasks are that the functionality of existing planners is extended to planning with exponential utility functions. This makes planning with nonlinear utility functions as fast as planning for traditional preference models and enables one to participate in performance improvements achieved by other researchers in the currently very active field of deterministic and probabilistic planning. This also makes it possible to integrate the planning-task transformations easily into existing agent architectures, such as robot architectures.

Chapter 5

Conclusions

In this thesis, we developed efficient general-purpose search and planning methods that solve one-shot (that is, single-instance) planning tasks for goal-directed acting in the presence of incomplete information. We did this by combining search and planning methods from artificial intelligence with methods from other disciplines, namely, operations research and utility theory.

In Chapter 2, we studied **Acting with Agent-Centered Search**. Agent-centered search methods interleave planning and plan execution to find suboptimal plans fast and plan only in the part of the domain around the current states of the agents. This is the part of the domain that is immediately relevant for the agents in their current situation. We developed real-time search methods, those agent-centered search methods that search forward from the current states of the agents and associate information with the states to prevent cycling. We assumed that probabilities were not available and developed methods that attempt to minimize the worst-case plan-execution cost.

In Chapter 3, we studied **Acting with POMDPs**. Partially observable Markov decision process models (POMDPs) from operations research represent uncertainty with probabilities and reason with them, including planning and learning improved models. We assumed that probabilities were available or could be learned and developed methods that attempt to minimize the average plan-execution cost.

In Chapter 4, we studied **Acting with Nonlinear Utility Functions**. Nonlinear utility functions from utility theory can represent immediate soft deadlines and risk attitudes in high-stake one-shot planning domains. We assumed that probabilities were available and developed methods that attempt to maximize the average utility of the plan-execution cost, where the utility is an exponential function of the cost.

The ideas discussed in the three chapters apply to a variety of real-world agents and tasks. We illustrated them using goal-directed robot-navigation tasks.

Results of Chapter 3 on Acting with POMDPs: Robots have to exhibit robust navigation behavior in the presence of various kinds of uncertainty. This includes uncertainty in actuation, uncertainty in sensing and sensor data interpretation, uncertainty in the initial pose of the robot, and uncertainty about their environment, such as the lengths of corridors. POMDPs address this problem. We illustrated how they can be integrated into robot architectures to provide a theoretically grounded framework for pose estimation, path planning, control during navigation, and learning. The robot maintains a probability distribution over its current pose instead of a single estimate of its current pose. Thus, the robot always has some belief as to what its true pose is, and is never completely lost. Different from Kalman filters, POMDPs discretize the poses which allows them to represent arbitrary probability distributions.

We developed efficient methods for POMDP planning and learning. Our application of these methods to office-navigation tasks showed that they provide a robust alternative to metric-based and landmark-based navigation methods, resulting in both in a new robot navigation architecture and a novel application area for POMDPs. Our POMDP planning method neglects state uncertainty during planning and then accounts for it

greedily during navigation. Our POMDP learning method, the GROW-BW method, extends the Baum-Welch method with a hill-climbing method that enables it to change the structure of the POMDP. It learns improved POMDPs without requiring a teacher or control of the robot. It decreases the memory requirements of the Baum-Welch method by using a sliding time window on the training data. It also decreases the training data requirements of the Baum-Welch method by imposing equality constraints between probabilities and using Bayesian estimates for the probabilities instead of maximum likelihood estimates.

It would be interesting to develop slightly more computationally intensive POMDP planning methods that produce plans of better quality. They could use the restrictive structure of the POMDP to make planning more efficient. It would also be interesting to develop POMDP learning methods that are able to detect and correct inaccuracies in the given topological maps or learn them from scratch.

Results of Chapter 4 on Acting with Nonlinear Utility Functions: Given a probabilistic navigation model, robots can solve goal-directed navigation tasks but still have to decide which one of all navigation plans that reach the destination to prefer. The preference model should take into account that the utility of navigation plans is often a nonlinear function of the plan-execution time. Exponential utility functions address this problem. We illustrated how they can be integrated into robot architectures to plan for immediate soft deadlines and a continuum of risk attitudes in high-stake one-shot planning domains, including risk-seeking behavior (such as gambling) and risk-averse behavior (such as holding insurance). Exponential utility functions preserve the modularity of planning tasks which allows for efficient planning. They can also trade-off between minimizing the worst-case plan-execution cost (as done in Chapter 2) and the average plan-execution cost (as done in Chapter 3).

We developed the additive and multiplicative planning-task transformations. They transform planning tasks with exponential utility functions to planning tasks that standard search and planning methods from artificial intelligence can solve, including those that do not reason about plan-execution costs at all. Thus, the transformations extend the functionality of these planners to maximizing average utility and make planning with exponential utility functions as fast as planning for traditional preference models. The transformations are simple context-insensitive representation changes that can be performed locally on various representations of planning tasks. The additive planning-task transformation applies to planning tasks that can be solved with sequential plans (more precisely: planning tasks whose actions have deterministic outcomes but whose rewards can be nondeterministic) and transforms them to deterministic planning tasks. The original planning task can then be solved by minimizing the plan-execution cost for the transformed planning task. The multiplicative planning-task transformation applies to planning tasks that can be solved with conditional plans and transforms them to probabilistic planning tasks. The original planning task can then be solved by either maximizing the probability of goal achievement or minimizing the average plan-execution cost for the transformed planning task.

It would be interesting to integrate the planning-task transformations with planning methods for arbitrary nonlinear or multi-variable utility functions, because exponential utility functions are single-variable utility functions with a restricted shape. For example, they cannot handle planning tasks with hard deadlines or deadlines in the future, risk-attitudes of decision makers whose choices depend on how wealthy they are, and more than one scarce resource. Finally, concave exponential utility functions can pose problems for cyclic plans.

Results of Chapter 2 on Acting with Agent-Centered Search: Given a probabilistic navigation model, robots can solve goal-directed navigation tasks but still have to decide which one of all navigation plans that reach the destination to prefer. The preference model should take the planning cost into account, especially for one-shot planning tasks, since planning can be time-consuming in the presence of incomplete information. Agent-centered search methods address this problem. We illustrated how they can be integrated into robot architectures to gather information early and use it to resolve uncertainty caused by nondeterminism, which can decrease the planning cost.

We developed the Min-Max LRTA* method, an efficient domain-independent real-time search method for nondeterministic domains. Our application of Min-Max LRTA* to maze-navigation tasks showed that it

provides an efficient alternative to first planning and then executing the resulting plan. Min-Max LRTA* allows for fine-grained control over how much planning to do between plan executions and uses heuristic knowledge to guide planning. It produces initially suboptimal plans that keep the sum of planning and plan-execution cost small and is able to improve the plans over time as it solves similar planning tasks.

It would be interesting to develop real-time search methods that can guarantee that their memory requirements are smaller than the size of the state space and real-time search methods that are able to use their problem-solving experience to improve their parameters, such as their look-ahead.

Other Results: The results of this thesis also led to a number of additional insights.

First, the results of this thesis provide a better understanding of why, when, and how well real-time search methods work in deterministic domains. We analyzed, both theoretically and experimentally, how heuristic knowledge and domain properties influence the performance of real-time search methods. Insights include that better approximations of the goal distances can degrade the performance of real-time search methods, that Eulerian domains can be searched efficiently even with inefficient real-time search methods, and that the performance of uninformed real-time search methods is often proportional to the number of states (or actions) of a domain times its maximal goal distance.

Second, the results of this thesis also provide guidelines for selecting test-beds for real-time search methods. We explained why sliding-tile puzzles, grid-worlds, and other traditional test-beds from artificial intelligence can be searched efficiently with a variety of real-time search methods, and introduced reset state spaces, quicksand state spaces, and "complex state spaces" that cannot be searched as easily.

Third, the results of this thesis also provide guidelines for selecting representations of reinforcement-learning tasks that allow them to be solved quickly by reinforcement-learning methods. We showed that representations of reinforcement-learning tasks can have a large impact on the performance of reinforcement-learning methods, and that the performance of reinforcement learning methods can be improved by making the reward structure dense or initializing the reinforcement-learning values optimistically.

Finally, the results of this thesis also provide a novel interpretation of discounting. We showed that discounting can be viewed as a special case of planning with convex exponential utility functions. This provides an alternative to the common interpretation of discounting as accounting for either interest accumulated on unused resources, an unrealistic interpretation for most agents, or the possibility of their death during each action execution.

Future Work: This thesis developed component technologies for goal-directed acting with incomplete information and illustrated their usefulness. We developed methods for acting with agent-centered search, acting with POMDPs, and acting with nonlinear utility functions. In the following, we list possibilities for future work.

First, it is future work to integrate our methods into a single agent architecture. The central ideas behind the three main chapters are mostly orthogonal and could be combined into a single agent architecture that uses POMDPs to reason with uncertainty, attempts to find plans that maximize average utility, and uses agent-centered search methods to approximate these plans efficiently.

Second, it is also future work to investigate further properties of our methods, both formally and empirically, and address their limitations by either extending them or developing alternative methods. The limitations of our methods are discussed in detail in their respective chapters.

Third, it is also future work to study additional aspects of goal-directed acting in the presence of incomplete information. For example, an agent can have incomplete information because there are other agents in the environment that interfere with its actions. The agent then has to learn to predict their behavior, cooperate with them, or compete with them.

Finally, it is also future work to investigate additional applications of goal-directed acting in the presence of incomplete information (besides goal-directed robot navigation) since a multitude of other tasks fit this framework as well. After all, all agents have to act in the world and they usually have to do so in a

goal-directed and efficient manner despite incomplete knowledge, imperfect actuation capabilities, limited or noisy perception, and insufficient reasoning speed. Other possible applications include highway navigation, package routing in computer networks, on-line scheduling, and web navigation.

To summarize, we developed efficient general-purpose search and planning methods that solve one-shot planning tasks for goal-directed acting in the presence of incomplete information. We addressed three problems that are important for real-world agents: how to act reliably despite a substantial amount of uncertainty, which one of several plans that all achieve the goal to choose, and how to decrease the sum of planning and plan execution cost. Our solutions to these problems combine search and planning methods from artificial intelligence with methods from operations research and utility theory. We believe that the cross fertilization among these and other decision-making disciplines is an important step towards building agents for goal-directed acting with incomplete information.

Chapter 6

Appendices

6.1 Complexity of Edge Counting

In this appendix, we complete the proof of Theorem 17 from page 53. We analyze the complexity of the variant of Edge Counting in Figure 6.1 [Koenig and Simmons, 1996c]. Different from the variant used in the main text (Section 2.5.2.1), its q-values approximate the positive (not negative) number of times the actions have been executed. The time superscript t refers to the values of the variables immediately before the $(t+1)$st value-update step of Edge Counting (Line 4 in Figure 6.1).

Theorem 25 *For all times $t = 0, 1, 2, \ldots$ (until termination), $s \in S$, and $a \in A(s)$, $q^t(s,a) \leq \min_{a' \in A(s)} q^t(s,a') + 1$.*

Thus, the q-values of any two actions leaving a state differ by at most one.

Proof by induction on t: The theorem holds at time $t = 0$, since $q^0(s,a) = 0$ for all $s \in S$ and $a \in A(s)$. Assume that the theorem holds at an arbitrary time t, and consider arbitrary $s \in S$ and $a \in A(s)$. The only q-value that changes between t and $t+1$ is $q(s^t, a^t)$. We distinguish two cases: First, $s \neq s^t$ or $a \neq a^t$. Then, $q^{t+1}(s,a) = q^t(s,a) \overset{\text{Assumption}}{\leq} \min_{a' \in A(s)} q^t(s,a') + 1 \overset{\text{Monotonicity}}{\leq} \min_{a' \in A(s)} q^{t+1}(s,a') + 1$. Second, $s = s^t$ and $a = a^t$. Then, $q^{t+1}(s,a) = q^t(s,a) + 1 \overset{\text{Action Selection}}{=} \min_{a' \in A(s)} q^t(s,a') + 1 \overset{\text{Monotonicity}}{\leq} \min_{a' \in A(s)} q^{t+1}(s,a') + 1$. Put together, $q^{t+1}(s,a) \leq \min_{a' \in A(s)} q^{t+1}(s,a') + 1$ for all $s \in S$ and $a \in A(s)$, and the theorem holds at time $t+1$ as well. ■

Theorem 26 *For all times $t = 0, 1, 2, \ldots$ (until termination), $\max_{a \in A(s^t)} q^{t+1}(s^t, a) = q^{t+1}(s^t, a^t)$.*

Thus, the action executed last in a state has a largest q-value of all actions leaving the state (after the action execution).

Proof: Consider an arbitrary time t and an arbitrary action $a \in A(s^t)$. We distinguish two cases: First, $a \neq a^t$. Then, $q^{t+1}(s^t,a) = q^t(s^t,a) \overset{\text{Theorem 25}}{\leq} \min_{a' \in A(s^t)} q^t(s^t,a') + 1 \overset{\text{Action Selection}}{=} q^t(s^t,a^t) + 1 = q^{t+1}(s^t,a^t)$. Second, $a = a^t$. Then, $q^{t+1}(s^t,a) = q^{t+1}(s^t,a^t)$ (trivially). Put together, $q^{t+1}(s^t,a) \leq q^{t+1}(s^t,a^t)$ for all $a \in A(s^t)$, but equality holds for at least one action (namely a^t), and the theorem follows. ■

In the following, we use sets that can contain duplicate elements (bags). To distinguish operators on bags from operators on sets, we use an additional dot. We use the following operators: construction (description) of bags $\dot{\{\}}$, membership of an element in a bag $\dot{\in}$ (or $\dot{\ni}$), equality of bags $\doteq$, nonstrict inclusion of bags $\dot{\subseteq}$

Initially, $q(s,a) = 0$ for all $s \in S$ and $a \in A(s)$.

1. $s := s_{start}$.
2. If $s \in G$, then stop successfully.
3. $a :=$ one-of $\arg\min_{a' \in A(s)} q(s,a')$.
4. $q(s,a) := q(s,a) + 1$.
5. Execute action a, that is, change the current state to $succ(s,a)$.
6. $s :=$ the current state.
7. Go to 2.

Figure 6.1: Edge Counting

(or $\dot{\supseteq}$), union of bags $\dot{\cup}$, intersection of bags $\dot{\cap}$, and difference of bags $\dot{\setminus}$. The operators $\dot{\cup}$, $\dot{\cap}$, and $\dot{\setminus}$ have the same precedence and are left-associative. Furthermore, we always denote a bag with one element by the element itself.

We define the bags $IN1^t(s)$, $OUT1^t(s)$, $IN2^t(s)$, and $OUT2^t(s)$ inductively as follows for all times $t = 0, 1, 2, \ldots$ (until termination) and $s \in S$:

$$
\begin{array}{rcl}
IN1^0(s) & := & \dot{\{}\, q^0(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s \,\dot{\}} \\
OUT1^0(s) & := & \dot{\{}\, q^0(s,a) : a \in A(s) \,\dot{\}} \\
IN2^t(s) & := & IN1^t(s) \\
OUT2^t(s) & := & \begin{cases} OUT1^t(s) \mathbin{\dot{\setminus}} q^t(s^t,a^t) \mathbin{\dot{\cup}} q^{t+1}(s^t,a^t) & \text{for } s = s^t \\ OUT1^t(s) & \text{otherwise} \end{cases} \\
IN1^{t+1}(s) & := & \begin{cases} IN2^t(s) \mathbin{\dot{\setminus}} q^t(s^t,a^t) \mathbin{\dot{\cup}} q^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ IN2^t(s) & \text{otherwise} \end{cases} \\
OUT1^{t+1}(s) & := & OUT2^t(s)
\end{array}
$$

Theorem 27 *For all times $t = 0, 1, 2, \ldots$ (until termination) and $s \in S$,*

$$
\begin{array}{rcl}
IN1^t(s) & \doteq & \dot{\{}\, q^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s \,\dot{\}} \\
OUT1^t(s) & \doteq & \dot{\{}\, q^t(s,a) : a \in A(s) \,\dot{\}}
\end{array}
$$

Thus, $IN1^t(s)$ is the bag of q-values of all incoming actions into state s at time t, and $OUT1^t(s)$ is the bag of q-values of all outgoing actions from state s at time t. Similarly, $IN2^t(s) \doteq IN1^t(s)$ and $OUT2^t(s) \doteq OUT1^{t+1}(s)$ for all times $t = 0, 1, 2, \ldots$ (until termination) and $s \in S$.

Proof by induction on t: The theorem holds at time $t = 0$ (by definition). Assume that it holds at an arbitrary time t. Then, for all $s \in S$,

$$
\begin{array}{rcl}
IN1^{t+1}(s) & \doteq & \begin{cases} IN2^t(s) \mathbin{\dot{\setminus}} q^t(s^t,a^t) \mathbin{\dot{\cup}} q^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ IN2^t(s) & \text{otherwise} \end{cases} \\
 & \doteq & \begin{cases} IN1^t(s) \mathbin{\dot{\setminus}} q^t(s^t,a^t) \mathbin{\dot{\cup}} q^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ IN1^t(s) & \text{otherwise} \end{cases}
\end{array}
$$

$$\begin{aligned}
&\doteq \begin{cases} \dot\{\, q^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s \,\dot\} \ldots \\ \dot\{\, q^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s \,\dot\} \ldots \end{cases} \\
&\qquad \begin{array}{ll} \ldots \mathbin{\dot\setminus} q^t(s^t,a^t) \mathbin{\dot\cup} q^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ \ldots & \text{otherwise} \end{array} \\
&\doteq \dot\{\, q^{t+1}(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s \,\dot\} \\
OUT1^{t+1}(s) &\doteq OUT2^t(s) \\
&\doteq \begin{cases} OUT1^t(s) \mathbin{\dot\setminus} q^t(s^t,a^t) \mathbin{\dot\cup} q^{t+1}(s^t,a^t) & \text{for } s = s^t \\ OUT1^t(s) & \text{otherwise} \end{cases} \\
&\doteq \begin{cases} \dot\{\, q^t(s,a) : a \in A(s) \,\dot\} \mathbin{\dot\setminus} q^t(s^t,a^t) \mathbin{\dot\cup} q^{t+1}(s^t,a^t) & \text{for } s = s^t \\ \dot\{\, q^t(s,a) : a \in A(s) \,\dot\} & \text{otherwise} \end{cases} \\
&\doteq \dot\{\, q^{t+1}(s,a) : a \in A(s) \,\dot\}
\end{aligned}$$

Thus, the theorem holds at time $t+1$ as well. ■

To keep our notation concise, we also define the bags $IND1^t(s)$, $OUTD1^t(s)$, $IND2^t(s)$, and $OUTD2^t(s)$ for all times $t = 0, 1, 2, \ldots$ (until termination) and $s \in S$:

$$\begin{aligned}
IND1^t(s) &:= IN1^t(s) \mathbin{\dot\setminus} OUT1^t(s) \\
OUTD1^t(s) &:= OUT1^t(s) \mathbin{\dot\setminus} IN1^t(s) \\
IND2^t(s) &:= IN2^t(s) \mathbin{\dot\setminus} OUT2^t(s) \\
OUTD2^t(s) &:= OUT2^t(s) \mathbin{\dot\setminus} IN2^t(s)
\end{aligned}$$

Note that $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ if and only if $IN1^t(s) \doteq OUT1^t(s)$, and $IND2^t(s) \doteq OUTD2^t(s) \doteq \emptyset$ if and only if $IN2^t(s) \doteq OUT2^t(s)$. We call a state $s \in S$ balanced at time t if $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$. This means that, for every state, the number of incoming actions with q-value n equals the number of outgoing actions with q-value n for all n.

Theorem 28 *For all times $t = 0, 1, 2, \ldots$ (until termination), (A) and (B) hold, where*

(A) either

(a) $s^t = s_{start}$, and $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ for all $s \in S$ (that is, all states are balanced)

or

(b) there exist states $s_1, s_2, \ldots, s_k \in S$ and an integer y with $y \geq k-1 \geq 1$, $s_1 = s_{start}$, $s_k = s^t$, $IND1^t(s_1) \doteq y-1$ and $OUTD1^t(s_1) \doteq y$, $IND1^t(s_i) \doteq \dot\{\, y-i, y-i+2 \,\dot\}$ and $OUTD1^t(s_i) \doteq \dot\{\, y-i+1, y-i+1 \,\dot\}$ for $i = 2, 3, \ldots, k-1$, $IND1^t(s_k) \doteq y-k+2$ and $OUTD1^t(s_k) \doteq y-k+1$, and finally $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ for $s \notin \{s_1, s_2, \ldots, s_k\}$. (Note that case (b) implies that $k \geq 2$ and $s_1, s_2, \ldots, s_k$ are pairwise different, for example, $s^t = s_k \neq s_{start}$.)

(B) there exist states $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}} \in S$ and an integer $\bar{y}$ with $\bar{y} \geq \bar{k} \geq 1$, $q^t(s^t, a^t) = \bar{y} - \bar{k}$, $\bar{s}_1 = s_{start}$, $IND2^t(\bar{s}_1) \doteq \bar{y} - 1$ and $OUTD2^t(\bar{s}_1) \doteq \bar{y}$, $IND2^t(\bar{s}_i) \doteq \dot\{\, \bar{y}-i, \bar{y}-i+2 \,\dot\}$ and $OUTD2^t(\bar{s}_i) \doteq \dot\{\, \bar{y}-i+1, \bar{y}-i+1 \,\dot\}$ for $i = 2, 3, \ldots, \bar{k}$, and finally $IND2^t(s) \doteq OUTD2^t(s) \doteq \emptyset$ for $s \notin \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}}\}$. (Note that case (B) implies that $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}}$ are pairwise different.)

Proof by induction on t:

- (A) holds at time $t = 0$: $s^0 = s_{start}$, and $q^0(s,a) = 0$ for all $s \in S$ and $a \in A(s)$. Since the graph is Eulerian, (a) holds.
- Assume that (A) holds at an arbitrary time t. We show that (B) holds at time t as well. Note that, if $s \neq s^t$, then $IN2^t(s) \doteq IN1^t(s)$ (this equality holds for all $s \in S$) and $OUT2^t(s) \doteq OUT1^t(s)$ and therefore $IND2^t(s) \doteq IN2^t(s) \setminus OUT2^t(s) \doteq IN1^t(s) \setminus OUT1^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUT2^t(s) \setminus IN2^t(s) \doteq OUT1^t(s) \setminus IN1^t(s) \doteq OUTD1^t(s)$. We distinguish two cases to determine $IND2^t(s)$ and $OUTD2^t(s)$ for $s = s^t$:
  - First, (a) holds at time t. We show that (B) holds at time t with $\bar{k} = 1$, $\bar{y} = q(s^t, a^t) + 1$, and $\bar{s}_1 = s_{start} = s^t$. Obviously, $\bar{y} \geq \bar{k} \geq 1$ and $q^t(s^t, a^t) = \bar{y} - \bar{k}$. Define $X := IN1^t(s^t) \cap OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \cup IND1^t(s^t) \doteq X \cup \emptyset \doteq X$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t) \doteq X \cup OUTD1^t(s^t) \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t) \doteq X \cup \emptyset \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t) \doteq X \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t)$. Since $q^t(s^t, a^t) \in OUT1^t(s^t) \doteq X \cup OUTD1^t(s^t) \doteq X \cup \emptyset \doteq X$ and $q^{t+1}(s^t, a^t) \neq q^t(s^t, a^t)$, it follows that $IND2^t(\bar{s}_1) \doteq IND2^t(s^t) \doteq IN2^t(s^t) \setminus OUT2^t(s^t) \doteq X \setminus (X \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t)) \doteq q^t(s^t, a^t) \doteq \bar{y} - \bar{k} \doteq \bar{y} - 1$. Also, $OUTD2^t(\bar{s}_1) \doteq OUTD2^t(s^t) \doteq OUT2^t(s^t) \setminus IN2^t(s^t) \doteq (X \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t)) \setminus X \doteq q^{t+1}(s^t, a^t) \doteq q^t(s^t, a^t) + 1 \doteq \bar{y} - \bar{k} + 1 \doteq \bar{y}$. For $s \neq \bar{s}_1$ (that is, $s \neq s^t$), $IND2^t(s) \doteq IND1^t(s) \doteq \emptyset$ and $OUTD2^t(s) \doteq OUTD1^t(s) \doteq \emptyset$.
  - Second, (b) holds at time t. Then, there exists an action $a \in A(s^t)$ with $q^t(s^t, a) = y - k + 1$, because $OUT1^t(s^t) \doteq OUT1^t(s_k) \supseteq OUTD1^t(s_k) \doteq y - k + 1$. Then, $y - k = q^t(s^t, a) - 1 \overset{\text{Theorem 25}}{\leq} \min_{a' \in A(s^t)} q^t(s^t, a') \overset{\text{Action Selection}}{=} q^t(s^t, a^t) \overset{\text{Action Selection}}{=} \min_{a' \in A(s^t)} q^t(s^t, a') \leq q^t(s^t, a) = y - k + 1$. Thus, either $q^t(s^t, a^t) = y - k$ or $q^t(s^t, a^t) = y - k + 1$. Consequently, we distinguish two sub-cases:
    - First, $q^t(s^t, a^t) = y - k$. We show that (B) holds at time t with $\bar{k} = k$, $\bar{y} = y$, and $\bar{s}_i = s_i$ for $1 \leq i \leq \bar{k} = k$. Obviously, $0 \leq q^t(s^t, a^t) \overset{\text{Assumption}}{=} y - k = \bar{y} - \bar{k}$ and thus $\bar{y} \geq \bar{k} = k \geq 2 \geq 1$, which implies $\bar{y} \geq \bar{k} \geq 1$. Furthermore, $\bar{s}_1 = s_1 = s_{start}$. Define $X := IN1^t(s^t) \cap OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \cup IND1^t(s^t) \doteq X \cup IND1^t(s_k) \doteq X \cup y - k + 2$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t) \doteq X \cup OUTD1^t(s^t) \setminus y - k \cup y - k + 1 \doteq X \cup OUTD1^t(s_k) \setminus y - k \cup y - k + 1 \doteq X \cup y - k + 1 \setminus y - k \cup y - k + 1 \doteq X \setminus y - k \cup \{ y - k + 1, y - k + 1 \}$. Note that $y - k \in X$, because $q^t(s^t, a^t) \overset{\text{Assumption}}{=} y - k$ and thus $y - k \in OUT1^t(s^t) \doteq X \cup OUTD1^t(s^t) \doteq X \cup OUTD1^t(s_k) \doteq X \cup y - k + 1$. Consequently, $IND2^t(s^t) \doteq IN2^t(s^t) \setminus OUT2^t(s^t) \doteq \{ y - k, y - k + 2 \} \doteq \{ \bar{y} - \bar{k}, \bar{y} - \bar{k} + 2 \}$ and $OUTD2^t(s^t) \doteq OUT2^t(s^t) \setminus IN2^t(s^t) \doteq \{ y - k + 1, y - k + 1 \} \doteq \{ \bar{y} - \bar{k} + 1, \bar{y} - \bar{k} + 1 \}$ for $s^t = s_k = \bar{s}_{\bar{k}}$. It is easy to show that (B) also holds for $s \neq s^t$, using $IND2^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUTD1^t(s)$ together with $\bar{y} = y$.
    - Second, $q^t(s^t, a^t) = y - k + 1$. We show that (B) holds at time t with $\bar{k} = k - 1$, $\bar{y} = y$, and $\bar{s}_i = s_i$ for $1 \leq i \leq \bar{k} = k - 1$. Obviously, $\bar{y} \geq \bar{k} \geq 1$ (since $y \geq k - 1 \geq 1$), $q^t(s^t, a^t) \overset{\text{Assumption}}{=} y - k + 1 = \bar{y} - \bar{k}$, and $\bar{s}_1 = s_1 = s_{start}$. Define $X := IN1^t(s^t) \cap OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \cup IND1^t(s^t) \doteq X \cup IND1^t(s_k) \doteq X \cup y - k + 2$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \setminus q^t(s^t, a^t) \cup q^{t+1}(s^t, a^t) \doteq OUT1^t(s_k) \setminus y - k + 1 \cup y - k + 2 \doteq X \cup OUTD1^t(s_k) \setminus y - k + 1 \cup y - k + 2 \doteq X \cup y - k + 1 \setminus y - k + 1 \cup y - k + 2 \doteq X \cup y - k + 2$. Consequently, $IND2^t(s^t) \doteq IN2^t(s^t) \setminus OUT2^t(s^t) \doteq \emptyset$ and $OUTD2^t(s^t) \doteq OUT2^t(s^t) \setminus IN2^t(s^t) \doteq \emptyset$ for $s^t = s_k \notin \{s_1, \ldots, s_{k-1}\} =$

$\{\bar{s}_1, \ldots, \bar{s}_k\}$. It is easy to show that (B) also holds for $s \neq s^t$, using $IND2^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUTD1^t(s)$ together with $\bar{y} = y$.

- Assume that (B) holds at an arbitrary time t. We show that (A) holds at time $t+1$. Note that, if $s \neq s^{t+1}$, then $IN1^{t+1}(s) \doteq IN2^t(s)$ and $OUT1^{t+1}(s) \doteq OUT2^t(s)$ (this equality holds for all $s \in S$) and therefore $IND1^{t+1}(s) \doteq IN1^{t+1}(s) \setminus OUT1^{t+1}(s) \doteq IN2^t(s) \setminus OUT2^t(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUT1^{t+1}(s) \setminus IN1^{t+1}(s) \doteq OUT2^t(s) \setminus IN2^t(s) \doteq OUTD2^t(s)$. We distinguish two cases to determine $IND1^{t+1}(s)$ and $OUTD1^{t+1}(s)$ for $s = s^{t+1}$:

  - First, $s^{t+1} \notin \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_k\}$. We show that (b) holds at time $t+1$ with $k = \bar{k} + 1$, $y = \bar{y}$, $s_i = \bar{s}_i$ for $1 \leq i \leq k-1 = \bar{k}$, and $s_k = s^{t+1}$. Obviously, $y \geq k-1 \geq 1$ (since $y = \bar{y} \geq \bar{k} = k-1 = \bar{k} \geq 1$) and $s_1 = \bar{s}_1 = s_{start}$. Define $X := IN2^t(s^{t+1}) \cap OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \setminus q^t(s^t, a^t) \dot\cup q^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup IND2^t(s^{t+1}) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup \emptyset \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \dot\cup OUTD2^t(s^{t+1}) \doteq X \dot\cup \emptyset \doteq X$. Note that $\bar{y}-\bar{k} \in X$, because $q^t(s^t, a^t) = \bar{y}-\bar{k}$ and thus $\bar{y}-\bar{k} \in IN1^t(s^{t+1}) \doteq IN2^t(s^{t+1}) \doteq X \dot\cup IND2^t(s^{t+1}) \doteq X \dot\cup \emptyset \doteq X$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \setminus OUT1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k}+1 \doteq y-k+2$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \setminus IN1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k} \doteq y-k+1$ for $s^{t+1} = s_k$. It is easy to show that (b) also holds for $s \neq s^{t+1}$, using $IND1^{t+1}(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s)$ together with $y = \bar{y}$.
  - Second, $s^{t+1} \in \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_k\}$. We show that $s^{t+1} = \bar{s}_k$ by contradiction. Suppose that $s^{t+1} = \bar{s}_i$ for some $i < k$ and define $X := IN2^t(s^{t+1}) \cap OUT2^t(s^{t+1})$. Then, there exists an action $a \in A(s^{t+1})$ with $q^{t+1}(s^{t+1}, a) = \bar{y}-k$, because $q^t(s^t, a^t) = \bar{y}-\bar{k}$ and thus $\bar{y}-\bar{k} \in IN1^t(s^{t+1}) \doteq IN2^t(s^{t+1}) \doteq X \dot\cup IND2^t(s^{t+1})$; it must be that $\bar{y}-\bar{k} \in X \subseteq OUT2^t(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1})$, since $\bar{y}-\bar{k} \notin IND2^t(\bar{s}_i) = IND2^t(s^{t+1})$. There also exists an action $a' \in A(s^{t+1})$ with $q^{t+1}(s^{t+1}, a') = \bar{y}-i+1$, because $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \supseteq OUTD2^t(s^{t+1}) \doteq OUTD2^t(\bar{s}_i) \ni \bar{y}-i+1$. Put together, $q^{t+1}(s^{t+1}, a') = \bar{y}-i+1 > \bar{y}-\bar{k}+1 = q^{t+1}(s^{t+1}, a)+1$ (since $i < \bar{k}$). This, however, is a contradiction to Theorem 25, which asserts that $q^{t+1}(s^{t+1}, a') \leq \min_{a'' \in A(s^{t+1})} q^{t+1}(s^{t+1}, a'') + 1 \leq q^{t+1}(s^{t+1}, a) + 1$. It follows that $s^{t+1} = \bar{s}_k$. We distinguish two sub-cases:

    * First, $\bar{k} = 1$. We show that (a) holds at time $t+1$. Obviously, $s^{t+1} = \bar{s}_k = \bar{s}_1 = s_{start}$. Define $X := IN2^t(s^{t+1}) \cap OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \setminus q^t(s^t, a^t) \dot\cup q^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup IND2^t(s^{t+1}) \setminus \bar{y}-1 \dot\cup \bar{y} \doteq X \dot\cup IND2^t(\bar{s}_1) \setminus \bar{y}-1 \dot\cup \bar{y} \doteq X \dot\cup \bar{y}-1 \setminus \bar{y}-1 \dot\cup \bar{y} \doteq X \dot\cup \bar{y}$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \dot\cup OUTD2^t(s^{t+1}) \doteq X \dot\cup OUTD2^t(\bar{s}_1) \doteq X \dot\cup \bar{y}$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \setminus OUT1^{t+1}(s^{t+1}) \doteq \emptyset$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \setminus IN1^{t+1}(s^{t+1}) \doteq \emptyset$ for $s^{t+1} = \bar{s}_1$. For $s \neq s^{t+1}$, $IND1^{t+1}(s) \doteq IND2^t(s) \doteq \emptyset$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s) \doteq \emptyset$.
    * Second, $\bar{k} > 1$. We show that (b) holds at time $t+1$ with $k = \bar{k}$, $y = \bar{y}$, and $s_i = \bar{s}_i$ for $1 \leq i \leq k = \bar{k}$. Obviously, $y \geq k-1 \geq 1$ (since $y = \bar{y} \geq \bar{k} = k = \bar{k} > 1$), $s_1 = \bar{s}_1 = s_{start}$, and $s_k = \bar{s}_k = s^{t+1}$. Define $X := IN2^t(s^{t+1}) \cap OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \setminus q^t(s^t, a^t) \dot\cup q^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup IND2^t(s^{t+1}) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup IND2^t(\bar{s}_k) \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup \{\bar{y}-\bar{k}, \bar{y}-\bar{k}+2\} \setminus \bar{y}-\bar{k} \dot\cup \bar{y}-\bar{k}+1 \doteq X \dot\cup \{\bar{y}-\bar{k}+1, \bar{y}-\bar{k}+2\}$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \dot\cup OUTD2^t(s^{t+1}) \doteq X \dot\cup OUTD2^t(\bar{s}_k) \doteq X \dot\cup \{\bar{y}-\bar{k}+1, \bar{y}-\bar{k}+1\}$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \setminus OUT1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k}-2 \doteq y-k+2$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \setminus IN1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k}+1 \doteq y-k+1$ for

$s^{t+1} = \bar{s}$ (note that $s^{t+1} \neq \bar{s}$). It is easy to show that (b) also holds for $s \neq s^{t+1}$, using $IND1^{t+1}(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s)$ together with $y = \bar{y}$. ■

Theorem 29 *For all times $t = 0, 1, 2, \ldots$ (until termination), $s \in S$, and $a \in A(s)$, $q^t(s,a) \leq \max_{a' \in A(s_{start})} q^t(s_{start}, a')$.*

Thus, there is always an action that leaves the start state and has a largest q-value of all actions.

Proof by induction on t: The theorem holds at time $t = 0$, since $q^0(s,a) = 0$ for all $s \in S$ and $a \in A(s)$. Assume that the theorem holds at an arbitrary time t, and consider arbitrary $s \in S$ and $a \in A(s)$. We distinguish two cases: First, $s \neq s^t$ or $a \neq a^t$. Then, $q^{t+1}(s,a) = q^t(s,a) \overset{\text{Assumption}}{\leq} \max_{a' \in A(s_{start})} q^t(s_{start}, a') \overset{\text{Monotonicity}}{\leq} \max_{a' \in A(s_{start})} q^{t+1}(s_{start}, a')$. Second, $s = s^t$ and $a = a^t$. We distinguish two sub-cases: First, $s = s_{start}$. Then, $q^{t+1}(s,a) = q^{t+1}(s_{start}, a) \leq \max_{a' \in A(s_{start})} q^{t+1}(s_{start}, a')$ (trivially). Second, $s \neq s_{start}$. Then, consider the variables from Theorem 28(B) at time t. There exists an action $a' \in A(s_{start})$ with $q^{t+1}(s_{start}, a') = \bar{y}$, because $OUTD1^{t+1}(s_{start}) \doteq OUTD2^t(s_{start}) \doteq OUTD2^t(\bar{s}) \geq OUTD2^t(\bar{s}) \doteq \bar{y}$. Then, $q^{t+1}(s,a) = q^{t+1}(s^t, a^t) = q^t(s^t, a^t) + 1 = (\bar{y} - \bar{k}) + 1 \overset{\bar{k} \geq 1}{\leq} \bar{y} = q^{t+1}(s_{start}, a') \leq \max_{a'' \in A(s_{start})} q^{t+1}(s_{start}, a'')$, and the theorem holds at time $t+1$ as well. ■

Theorem 30 *The complexity of Edge Counting is at most $e \times gd(s_{start}) - gd(s_{start})^2$ action executions over safely explorable, Eulerian domains.*

The proof of Theorem 17 shows that this bound is tight.

Proof: If $s_{start} \in G$, then $gd(s_{start}) = 0$ and the theorem holds since the goal state is reached without any action executions. Assume that $s_{start} \notin G$. Edge Counting reaches a goal state eventually (Theorem 10). In the following, we analyze how many action executions it needs at most.

Consider the latest time t with $s^t = s_{start}$. All states, including s_{start}, are balanced at time t (Theorem 28(a)). Now consider a shortest path from s_{start} to a closest goal state. The last action on that path has never been executed and thus has q-value 0 at time t. Consequently, the largest q-value of any action that leaves the last nongoal state s on the path is 0 if no other action leaves s, otherwise the largest q-value of any action leaving s is at most 1 (Theorem 25). Since s is balanced at time t, the largest q-value of any action entering s is 0 in the former case and at most 1 in the latter case. Thus, the q-value of the action that precedes the last action on the path is 0 or at most 1, etc. Finally, the q-value $q^t(s_{start}, a)$ of the first action a on the path is at most x where x is the number of intermediate states on the path (that is, not including the start and goal state) that have two or more outgoing actions. $q^t(s_{start}, a) \leq x \leq gd(s_{start}) - 1$, because the path has length $gd(s_{start})$ and therefore $gd(s_{start}) - 1$ intermediate states. $q^t(s_{start}, a) \leq x \leq e - gd(s_{start}) - 1$, because each of the $gd(s_{start}) + 1$ states on the path (including the start and goal state) has at least one outgoing action (all states on the path but the goal state have at least one outgoing action to the next state on the path; the goal state has at least one incoming action from the previous state on the path and, since the domain is Eulerian, also at least one outgoing action). Each state with two or more outgoing actions needs at least one more action of the remaining $e - gd(s_{start}) - 1$ actions. Put together, $\min_{a' \in A(s_{start})} q^t(s_{start}, a') \leq q^t(s_{start}, a) \leq x \leq \min(gd(s_{start}) - 1, e - gd(s_{start}) - 1)$.

Now consider the time $t' > t$ when Edge Counting reaches a goal state for the first time. There exist integers $y \geq k - 1 \geq 1$ and pairwise different states $s_1, s_2, \ldots, s_k \in S$ with $s_1 = s_{start}$, $s_k = s^{t'}$, $OUTD1^{t'}(s_1) \doteq y$, $OUTD1^{t'}(s_i) \doteq \{ y - i + 1, y - i + 1 \}$ for $i = 2, 3, \ldots, k-1$, and $OUTD1^{t'}(s_k) \doteq y - k + 1$ (Theorem 28(b)). The values k (with $k > 1$) and y are calculated as follows: At time t', all actions entering and leaving the goal state $s^{t'}$ have q-value 0, with the exception of the action with which it was entered, which has q-value 1. Thus, $0 \doteq OUTD1^{t'}(s^{t'}) \doteq OUTD1^{t'}(s_k) \doteq y - k + 1$ and consequently $k = y + 1$. Similarly, $y \doteq OUTD1^{t'}(s_1) \doteq OUTD1^{t'}(s_{start}) \doteq OUTD1^{t+1}(s_{start}) \doteq q^{t+1}(s_{start}, a^t)$

and consequently $y = q^{t+1}(s_{start}, a^t)$, since s_{start} was balanced at time t and Edge Counting never entered s_{start} again after time t. $y = q^{t+1}(s_{start}, a^t) = q^{t+1}(s^t, a^t) = q^t(s^t, a^t) + 1 = q^t(s_{start}, a^t) + 1 \overset{\text{Action Selection}}{=} \min_{a' \in A(s_{start})} q^t(s_{start}, a') + 1 \overset{\text{Previous Paragraph}}{\leq} \min(gd(s_{start}) - 1, e - gd(s_{start}) - 1) + 1 = \min(gd(s_{start}), e - gd(s_{start}))$.

To summarize, at time t', there is at least one action (in state s_1) with q-value y, there are at least two actions (in state s_i) with q-value $y - i + 1$ for $i = 2, 3, \ldots, k - 1 = y$, and there is at least one action (in state $s_k = s_{y+1}$) with q-value y-k+1=0. Since the states $s_1, s_2, \ldots, s_k$ are pairwise different, this accounts for $2(y - 1) + 2 = 2y$ actions. The q-values of the remaining $e - 2y$ actions can be at most (Theorem 29) $\max_{a' \in A(s_{start})} q^{t'}(s_{start}, a') = \max_{a' \in A(s_{start})} q^{t+1}(s_{start}, a') = \max_{a' \in A(s^t)} q^{t+1}(s^t, a') \overset{\text{Theorem 26}}{=} q^{t+1}(s^t, a^t) = q^{t+1}(s_{start}, a^t) = y$ at time t', since Edge Counting never entered s_{start} again after time t. It follows that the sum of the q-values of all actions at time t' is at most

$$y + 2\sum_{i=2}^{y}(y - i + 1) + 0 + (e - 2y)y = ey - y^2$$

This expression is maximized for $y = e/2$. However, there is the restriction on y that $y \leq \min(gd(s_{start}), e - gd(s_{start}))$. If $gd(s_{start}) \leq e/2$, then $gd(s_{start}) \leq e/2 \leq e - gd(s_{start})$ and $y = gd(s_{start})$ is optimal. If $gd(s_{start}) \geq e/2$, then $e - gd(s_{start}) \leq e/2 \leq gd(s_{start})$ and $y = e - gd(s_{start})$ is optimal. In both cases, the sum of the q-values of all actions at time t' is at most $ey - y^2 \leq e \times gd(s_{start}) - gd(s_{start})^2$. The theorem follows because the total number of action executions corresponds to the sum of the q-values of all actions at time t'. ■

6.2 Dirichlet Distributions

In this appendix, we explain how we use Dirichlet distributions to change the re-estimation formulas to use Bayes rule instead of maximum likelihood estimates, as outlined in Section 3.3.4.2.2 on page 119.

Overview of Dirichlet Distributions: Assume a finite sequence of independent events, where an event of type e_i is generated with probability $p_i > 0$ and $\sum_i p_i = 1$. Let e denote the event that the sequence contains n_i events of type e_i. Then, e is distributed according to a *multinomial distribution* (the multivariate counterpart to the binomial distribution) with parameter $p = (p_i)$. Its distribution $f(e|p)$ is

$$f(e|p) = \frac{(\sum_i n_i)!}{\prod_i n_i!} \prod_i p_i^{n_i}.$$

Thus, $f(e|p)$ is proportional to $\prod_i p_i^{n_i}$ if everything but p is given.

Assume that an observer observes e but does not know p. To obtain a Bayesian estimate for p, we maintain a probability distribution over all values of this random variable under the assumption that it is distributed according to a *Dirichlet distribution* (the multivariate counterpart to the beta distribution) with parameter $a = (a_i)$ where $a_i > 0$. Then, the density $f(p)$ of p is

$$f(p) = \frac{\Gamma(\sum a_i)}{\prod_i \Gamma(a_i)} \prod_i p_i^{a_i - 1},$$

where the gamma function $\Gamma(x)$ is the continuous counterpart to the factorial (that is, $x!$). Thus, $f(p)$ is proportional to $\prod_i p_i^{a_i - 1}$ if everything but p is given.

Ultimately, we are interested in the probabilities p_i. We need to estimate them from the Dirichlet distribution. In general, it is standard to minimize the mean squared difference between the estimate and the true parameter value (mean squared error). The Bayesian estimate $\widehat{p_i}$ for p_i for this criterion (loss function) is

$$E[(p_i - \widehat{p_i})^2] = E[(p_i)^2] - 2E(p_i)\widehat{p_i} + (\widehat{p_i})^2. \tag{6.1}$$

This quantity is minimized by $\widehat{p_i} = E(p_i)$. Thus, the Bayesian estimate for p_i is its mean $E(p_i)$. For Dirichlet distributions with parameter $a = (a_i)$, it holds that $E(p_i) = a_i / \sum_i a_i$.

We now use e to update p. According to Bayes rule,

$$f(p|e) = \frac{f(e|p)f(p)}{f(e)},$$

which is proportional to $\prod_i p_i^{n_i} \prod_i p_i^{a_i - 1}$ and thus also to $\prod_i p_i^{n_i + a_i - 1}$. Consequently, the distribution of p after observing e (short: $p|e$) is again a Dirichlet distribution, this time with parameter $a' = (a'_i)$ where $a'_i = n_i + a_i$. It follows that the Dirichlet distribution is a conjugate family for samples from a multinomial distribution. This is the reason for our assumption that p is distributed according to a Dirichlet distribution. The estimate for $p_i|e$ is

$$\widehat{p_i|e} = E(p_i|e) = \frac{a'_i}{\sum_i a'_i} = \frac{n_i + a_i}{\sum_i [n_i + a_i]}.$$

A longer introduction to Dirichlet distributions in the context of part manipulation with grippers is given in [Goldberg, 1990].

Using Dirichlet Distributions: We utilize the above properties of Dirichlet distributions as follows: Given an initial estimate p' for p, we use $a_i = k\ p'_i$ for some constant $k > 0$. Then, the estimate for p_i (the prior estimate) is, as expected,

$$\widehat{p_i} = \frac{a_i}{\sum_i a_i} = \frac{k\ p'_i}{\sum_i [k\ p'_i]} = p'_i.$$

The estimate for $p_i|e$ (the posterior estimate) is

$$\widehat{p_i|e} = \frac{n_i + a_i}{\sum_i [n_i + a_i]} = \frac{n_i + k\ p'_i}{\sum_i [n_i + k\ p'_i]} = \frac{n_i + k\ p'_i}{\sum_i [n_i] + k}.$$

The larger k, the more evidence is needed to change the prior estimate significantly.

In the POMDP-based navigation architecture, for example, the re-estimation formula A9 (from page 91) for the transition probabilities becomes (probability classes are not shown):

A9'. Set $\bar{p}(\hat{s}|s,a) := (\sum_{t=1\ldots T-1|a_t=a} \gamma_t(s,s') + k\ p(s'|s,a))/(\sum_{t=1\ldots T-1|a_t=a} \gamma_t(s) + k)$ for all $s, s' \in S$ and $a \in A(s)$.

In this case, e is not really distributed according to a multinomial distribution, but the modified re-estimation formulas seem to work well in practice.

6.3 Plans with Large Approximation Errors

In this appendix, we show that approximating exponential utility functions with linear utility functions can result in large approximation errors, as stated in Section 4.3 on page 142. We first present two theorems that are needed for the subsequent example.

Assume that the interval $[r_{min}, r_{max}]$ contains all possible rewards for a given planning task, where r_{min} and r_{max} are finite with $r_{min} \neq r_{max}$. For example, for navigation tasks, if the length of the shortest (and longest) path from the start location to the goal location is d_{min} (and d_{max}, respectively) then all rewards are contained in $[-d_{max}/v, -d_{min}/v]$, where v is the travel speed of the robot.

Theorem 31 *There exists a plan ps that maximizes $err(ps) := ce(ps) - er(ps)$ for $u(r) = \gamma^r$ with $\gamma > 1$ and is of the following form: it leads with probability $p_i > 0$ to a chronicle with reward r_i, where $r_i = r_{min}$ or $r_i = r_{max}$ for all i.*

Proof by contradiction: Consider a plan ps that leads with probability $p_j > 0$ to a chronicle with reward r_j, where $r_j \neq r_{min}$ and $r_j \neq r_{max}$. Note that

$$\frac{d^2}{(dr_j)^2}\left[ce(ps) - er(ps)\right] = \frac{d^2}{(dr_j)^2}\left[\log_\gamma \sum_i [p_i \gamma^{r_i}] - \sum_i [p_i r_i]\right] = \frac{\ln\gamma\, p_j \gamma^{r_j} \sum_{i\neq j}[p_i\gamma^{r_i}]}{(\sum_i [p_i\gamma^{r_i}])^2} \geq 0.$$

Consequently, the function is convex or linear and at least one of its global maxima is at either $r_j = r_{min}$ or $r_j = r_{max}$. Thus, changing r_j from its present value to either r_{min} or r_{max} does not decrease $err(ps)$. ■

Theorem 32 *The following plan maximizes $err(ps) := ce(ps) - er(ps)$ for $u(r) = \gamma^r$ with $\gamma > 1$: plan ps_{bad} leads with probability p_{bad} to a chronicle with reward r_{min} and with probability $1 - p_{bad}$ to a chronicle with reward r_{max}, where*

$$p_{bad} := -\frac{1}{\ln\gamma}\frac{1}{r_{max} - r_{min}} - \frac{\gamma^{r_{max}-r_{min}}}{1-\gamma^{r_{max}-r_{min}}}.$$

Proof: According to Theorem 31, there exists a plan ps_{bad} that maximizes $err(ps)$ and is of the following form: it leads with probability p_{bad} to a chronicle with reward r_{min} and with probability $1 - p_{bad}$ to reward r_{max}. We have to determine only the probability p_{bad} that maximizes $err(ps)$. Note that

$$\begin{aligned}
\frac{d}{dp_{bad}} err(ps_{bad}) &= \frac{d}{dp_{bad}}\left[ce(ps_{bad}) - er(ps_{bad})\right] \\
&= \frac{d}{dp_{bad}}\left[\log_\gamma[p_{bad}\gamma^{r_{min}} + (1-p_{bad})\gamma^{r_{max}}] - p_{bad}r_{min} - (1-p_{bad})r_{max}\right] \\
&= \frac{1}{\ln\gamma}\frac{\gamma^{r_{min}} - \gamma^{r_{max}}}{p_{bad}\gamma^{r_{min}} + (1-p_{bad})\gamma^{r_{max}}} - r_{min} + r_{max}.
\end{aligned}$$

Thus,

$$\begin{aligned}
\frac{d}{dp_{bad}} err(ps_{bad}) &= 0 \\
p_{bad} &= -\frac{1}{\ln\gamma}\frac{1}{r_{max} - r_{min}} - \frac{\gamma^{r_{max}-r_{min}}}{1-\gamma^{r_{max}-r_{min}}}.
\end{aligned}$$

This is indeed a maximum since

$$\frac{d^2}{(dp_{bad})^2} err(ps_{bad}) = -\frac{1}{\ln\gamma} \frac{(\gamma^{r_{min}} - \gamma^{r_{max}})^2}{(p_{bad}\gamma^{r_{min}} + (1-p_{bad})\gamma^{r_{max}})^2} < 0. \blacksquare$$

Now consider again the navigation task in Figure 4.2, this time in the presence of an immediate soft deadline that can be modeled with the utility function $u(r) = \gamma^r$ for $\gamma > 1$, where r is the negative travel time of the robot, measured in seconds. To determine the possible approximation error, let ps_{eu} denote the plan with maximal average utility and ps_{er} the plan with maximal average reward. The approximation error err of choosing ps_{er} over ps_{eu} is the difference in certainty equivalents between the two plans. Since the utility function is convex, $ce(ps) \geq er(ps)$ for all plans ps (Section 4.2.2). Thus,

$$\begin{aligned} err &= ce(ps_{eu}) - ce(ps_{er}) \\ &\leq ce(ps_{eu}) - er(ps_{er}) \\ &\leq ce(ps_{eu}) - er(ps_{eu}) \\ &\leq \max_{ps}[ce(ps) - er(ps)]. \end{aligned}$$

Theorem 32 shows that the following plan maximizes err: Plan ps_{bad} leads with probability p_{bad} (not necessarily equal to 0.50) to a chronicle with reward r_{min} and with probability $1 - p_{bad}$ to a chronicle with reward r_{max}, where

$$p_{bad} := -\frac{1}{\ln\gamma}\frac{1}{r_{max} - r_{min}} - \frac{\gamma^{r_{max}-r_{min}}}{1-\gamma^{r_{max}-r_{min}}}.$$

For ps_{bad}, it holds that

$$\begin{aligned} err &\leq \max_{ps}[ce(ps) - er(ps)] \\ &= ce(ps_{bad}) - er(ps_{bad}) \\ &= \frac{r_{max} - r_{min}}{\gamma^{r_{max}-r_{min}} - 1} - \log_\gamma \frac{e \ln \gamma^{r_{max}-r_{min}}}{\gamma^{r_{max}-r_{min}} - 1}. \end{aligned}$$

That this error bound can be tight can be seen as follows: Assume that a planner that maximizes average reward has to choose between plan ps_{bad} and a plan ps_{worse} that leads with probability one to a chronicle with reward $er(ps_{bad})$. Since both plans have the same average reward, the planner can choose ps_{worse}. Then, $ce(ps_{bad}) \geq er(ps_{bad}) = er(ps_{worse}) = ce(ps_{worse})$ and the approximation error is

$$\begin{aligned} err &= ce(ps_{bad}) - ce(ps_{worse}) \\ &= ce(ps_{bad}) - er(ps_{worse}) \\ &= ce(ps_{bad}) - er(ps_{bad}). \end{aligned}$$

To construct an example of the worst case, assume that $d_{min} = 20$ meters, $d_{max} = 200$ meters, and $v = 0.25$ meters per second. Furthermore, the utility function is $u(r) = (2^{1/300})^r$ (Figure 4.3 (right)). Then, similarly to Example 1 from Figure 4.2, ps_{bad} leads with probability 0.37 to a travel time of 80.00 seconds and with the complementary probability to a travel time of 800.00 seconds. The average utility of ps_{bad} is $eu(r_{bad}) = 0.41$, its certainty equivalent is $ce(r_{bad}) = -391.20$ seconds, and its average reward is $er(r_{bad}) = -535.49$ seconds. ps_{worse} leads with probability 1.00 to a travel time of 535.49 seconds. The average utility of ps_{worse} is $eu(r_{worse}) = 0.29$, its certainty equivalent is $ce(r_{worse}) = -535.49$ seconds, and its average reward is $er(r_{worse}) = -535.49$ seconds as well. The approximation error of choosing ps_{worse} over ps_{bad} calculates to roughly 2 1/2 minutes (144.30 seconds) for this delivery task whose average travel time is only about 9 minutes (535.49 seconds) in both cases.

6.4 Continuum of Risk Attitudes

Proposition 1 from page 144 showed that exponential utility functions cover a continuum of risk attitudes in high-stake one-shot planning domains, ranging from being strongly risk-averse over being risk-neutral to being strongly risk-seeking. In this appendix, we prove the four cases of Proposition 1 separately.

Theorem 33 shows that the certainty equivalent of any plan approaches its best-case reward as γ approaches infinity for a risk-seeking agent.

Theorem 33 *If the execution of plan ps leads with probability $p_i > 0$ to a chronicle with reward r_i, then $\lim_{\gamma\to\infty} ce(ps) = \max_i r_i$ for any utility function $u(r) = \gamma^r$ with $\gamma > 1$.*

Proof: It holds that

$$\begin{aligned}
\max_i r_i &= \lim_{\gamma\to\infty} \max_i r_i \\
&= \lim_{\gamma\to\infty} \log_\gamma \gamma^{\max_i r_i} \\
&= \lim_{\gamma\to\infty} \log_\gamma \sum_i [p_i \gamma^{\max_i r_i}] \\
&\geq \lim_{\gamma\to\infty} \log_\gamma \sum_i [p_i \gamma^{r_i}] \\
&\geq \lim_{\gamma\to\infty} \max_i \log_\gamma [p_i \gamma^{r_i}] \\
&= \lim_{\gamma\to\infty} \max_i [\log_\gamma p_i + r_i] \\
&= \max_i r_i,
\end{aligned}$$

and thus

$$\begin{aligned}
\lim_{\gamma\to\infty} ce(ps) &= \lim_{\gamma\to\infty} u^{-1}(\sum_i [p_i u(r_i)]) \\
&= \lim_{\gamma\to\infty} \log_\gamma \sum_i [p_i \gamma^{r_i}] \\
&\overset{\text{see above}}{=} \max_i r_i. \ \blacksquare
\end{aligned}$$

Theorem 34 shows that the certainty equivalent of any plan approaches its average reward as γ approaches one for a risk-seeking agent.

Theorem 34 *If the execution of plan ps leads with probability p_i to a chronicle with reward r_i, then $\lim_{\gamma \to 1} ce(ps) = er(ps)$ for any utility function $u(r) = \gamma^r$ with $\gamma > 1$.*

Proof: It holds that

$$\begin{aligned}
\lim_{\gamma \to 1} ce(ps) &= \lim_{\gamma \to 1} u^{-1}(\sum_i [p_i u(r_i)]) \\
&= \lim_{\gamma \to 1} \log_\gamma \sum_i [p_i \gamma^{r_i}] \\
&= \lim_{\gamma \to 1} \frac{\ln \sum_i [p_i \gamma^{r_i}]}{\ln \gamma} \\
&\stackrel{L'Hôpital}{=} \lim_{\gamma \to 1} \frac{\sum_i [p_i r_i \gamma^{r_i - 1}] / \sum_i [p_i \gamma^{r_i}]}{1/\gamma} \\
&= \lim_{\gamma \to 1} \frac{\sum_i [p_i r_i \gamma^{r_i}]}{\sum_i [p_i \gamma^{r_i}]} \\
&= \frac{\lim_{\gamma \to 1} \sum_i [p_i r_i \gamma^{r_i}]}{\lim_{\gamma \to 1} \sum_i [p_i \gamma^{r_i}]} \\
&= \frac{\sum_i [p_i r_i]}{1} \\
&= er(ps). \ \blacksquare
\end{aligned}$$

Similar theorems also hold for risk-averse agents. They can be proved by transforming the task to one for risk-seeking agents. Theorem 35 shows that the certainty equivalent of any plan approaches its average reward as γ approaches one for a risk-averse agent.

Theorem 35 *If the execution of plan ps leads with probability p_i to a chronicle with reward r_i, then $\lim_{\gamma \to 1} ce(ps) = er(ps)$ for any utility function $u(r) = -\gamma^r$ with $0 < \gamma < 1$.*

Proof: It holds that

$$\begin{aligned}
\lim_{\gamma \to 1} ce(ps) &= \lim_{\gamma \to 1} u^{-1}(\sum_i [p_i u(r_i)]) \\
&= \lim_{\gamma \to 1} \log_\gamma [-\sum_i [p_i (-\gamma^{r_i})]] \\
&= -\lim_{\gamma \to 1} [-\log_\gamma \sum_i [p_i \gamma^{r_i}]] \\
&= -\lim_{\gamma \to 1} \log_{1/\gamma} \sum_i [p_i (1/\gamma)^{-r_i}] \\
&\stackrel{Theorem\ 34}{=} -\sum_i [p_i (-r_i)] \\
&= \sum_i [p_i r_i] \\
&= er(ps). \ \blacksquare
\end{aligned}$$

Finally, Theorem 36 shows that the certainty equivalent of any plan approaches its worst-case reward as γ approaches zero for a risk-averse agent.

Theorem 36 *If the execution of plan ps leads with probability $p_i > 0$ to a chronicle with reward r_i, then $\lim_{\gamma \to 0} ce(ps) = \min_i r_i$ for any utility function $u(r) = -\gamma^r$ with $0 < \gamma < 1$.*

Proof: It holds that

$$\begin{aligned}
\lim_{\gamma \to 0} ce(ps) &= \lim_{\gamma \to 0} u^{-1}(\sum_i [p_i u(r_i)]) \\
&= \lim_{\gamma \to 0} \log_\gamma [- \sum_i [p_i (-\gamma^{r_i})]] \\
&= - \lim_{\gamma \to 0} [- \log_\gamma \sum_i [p_i \gamma^{r_i}]] \\
&= - \lim_{\gamma \to 0} \log_{1/\gamma} \sum_i [p_i (1/\gamma)^{-r_i}] \\
&\overset{Theorem\ 33}{=} - \max_i [-r_i] \\
&= \min_i r_i. \ \blacksquare
\end{aligned}$$

Bibliography

[Aleliunas *et al.*, 1979] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[Amarel, 1968] S. Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Elsevier, 1968.

[Ambros-Ingerson and Steel, 1988] J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of the National Conference on Artificial Intelligence*, pages 83–88, 1988.

[Baeza-Yates *et al.*, 1993] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106:234–252, 1993.

[Balch and Arkin, 1993] T. Balch and R. Arkin. Avoiding the past: A simple, but effective strategy for reactive navigation. In *International Conference on Robotics and Automation*, pages 678–685, 1993.

[Baras and James, 1997] J. Baras and M. James. Robust and risk-sensitive output feedback control for finite state machines and hidden Markov models. *Journal of Mathematical Systems, Estimation, and Control*, in press, 1997.

[Barto *et al.*, 1989] A. Barto, R. Sutton, and C. Watkins. Learning and sequential decision making. Technical Report 89–95, Department of Computer Science, University of Massachusetts at Amherst, Amherst (Massachusetts), 1989.

[Barto *et al.*, 1995] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 73(1):81–138, 1995.

[Basye *et al.*, 1989] K. Basye, T. Dean, and J. Vitter. Coping with uncertainty in map learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 663–668, 1989.

[Bellman, 1957] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Benson and Prieditis, 1992] G. Benson and A. Prieditis. Learning continuous-space navigation heuristics in real-time. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1992.

[Bensoussan and Schuppen, 1985] A. Bensoussan and J. Van Schuppen. Optimal control of partially observable stochastic systems with an exponential-of-integral performance index. *SIAM Journal on Control and Optimization*, 23(4):599–613, 1985.

[Bernoulli, 1738] D. Bernoulli. Specimen theoriae novae de mensura sortis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 5, 1738. Translated by L. Sommer, **Econometrica**, 22; 23–36, 1954.

[Bertsekas, 1987] D. Bertsekas. *Dynamic Programming, Deterministic and Stochastic Models*. Prentice Hall, 1987.

[Betke *et al.*, 1995] M. Betke, R. Rivest, and M. Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2/3), 1995.

[Blum *et al.*, 1991] A. Blum, P. Raghavan, and B. Schieber. Navigation in unfamiliar terrain. In *Proceedings of the Symposium on Theory of Computing*, pages 494–504, 1991.

[Blythe, 1994] J. Blythe. Planning with external events. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 94–101, 1994.

[Blythe, 1996] J. Blythe. Event-based decompositions for reasoning about external change in planners. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 27–34, 1996.

[Boddy and Dean, 1989] M. Boddy and T. Dean. Solving time-dependent planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 979–984, 1989.

[Bonet *et al.*, 1997] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[Borenstein *et al.*, 1996] J. Borenstein, B. Everett, and L. Feng. *Navigating Mobile Robots: Systems and Techniques*. Peters, 1996.

[Boutilier and Dearden, 1994] C. Boutilier and R. Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1016–1022, 1994.

[Boutilier and Poole, 1996] C. Boutilier and D. Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1168–1175, 1996.

[Boutilier *et al.*, 1995a] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. In *Proceedings of the European Workshop on Planning*, 1995.

[Boutilier *et al.*, 1995b] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1104–1111, 1995.

[Bresina and Drummond, 1990] J. Bresina and M. Drummond. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the National Conference on Artificial Intelligence*, pages 138–144, 1990.

[Brost and Christiansen, 1993] R. Brost and A. Christiansen. Probabilistic analysis of manipulation tasks: A research agenda. In *Proceedings of the International Conference on Robotics and Automation*, pages 549–556, 1993.

[Bunke *et al.*, 1995] H. Bunke, M. Roth, and E. Schukat-Talamazzini. Off-line cursive handwriting recognition using hidden Markov models. *Pattern Recognition*, 28(9):1399–1413, 1995.

[Burgard *et al.*, 1996] W. Burgard, D. Fox, D. Hennig, and T. Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the National Conference on Artificial Intelligence*, pages 896–901, 1996.

[Cassandra *et al.*, 1994] A. Cassandra, L. Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1023–1028, 1994.

[Cassandra *et al.*, 1996] A. Cassandra, L. Kaelbling, and J. Kurien. Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 963–972, 1996.

[Cassandra, 1997] A. Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Department of Computer Science, Brown University, Providence (Rhode Island), 1997.

[Chartrand and Lesniak, 1986] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Wadsworth and Brooks/Cole, second edition, 1986.

[Choset and Burdick, 1994] H. Choset and J. Burdick. Sensor based planning and nonsmooth analysis. In *Proceedings of the International Conference on Robotics and Automation*, pages 3034–3041, 1994.

[Chrisman, 1992] L. Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the National Conference on Artificial Intelligence*, pages 183–188, 1992.

[Christiansen and Goldberg, 1990] A. Christiansen and K. Goldberg. Robotic manipulation planning with stochastic actions. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Coraluppi and Marcus, 1996] S. Coraluppi and S. Marcus. Risk-sensitive control of Markov decision processes. In *Proceedings of the Conference on Information Science and Systems*, pages 934–939, 1996.

[Cox, 1994] I. Cox. Modeling a dynamic environment using a Bayesian multiple hypothesis approach. *Artificial Intelligence*, 66:311–344, 1994.

[Dasgupta *et al.*, 1994] P. Dasgupta, P. Chakrabarti, and S. DeSarkar. Agent searching in a tree and the optimality of iterative deepening. *Artificial Intelligence*, 71:195–208, 1994.

[Dean *et al.*, 1988] T. Dean, J. Firby, and D. Miller. Hierarchical planning involving deadlines, travel times, and resources. *Computational Intelligence*, 4(4):381–398, 1988.

[Dean *et al.*, 1990] T. Dean, K. Basye, R. Chekaluk, S. Hyun, M. Lejter, and M. Randazza. Coping with uncertainty in a control system for navigation and exploration. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1010–1015, 1990.

[Dean *et al.*, 1992] T. Dean, D. Angluin, K. Basye, S. Engelson, L. Kaelbling, E. Kokkevis, and O. Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 208–214, 1992.

[Dean *et al.*, 1993] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 574–579, 1993.

[Dean *et al.*, 1995] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, 1995.

[Dearden and Boutilier, 1994] R. Dearden and C. Boutilier. Integrating planning and execution in stochastic domains. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1994.

[Denardo and Rothblum, 1979] E. Denardo and U. Rothblum. Optimal stopping, exponential utility, and linear programming. *Mathematical Programming*, 16:228–244, 1979.

[Deng and Papadimitriou, 1990] X. Deng and C. Papadimitriou. Exploring an unknown graph. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 355–361, 1990.

[Devijver, 1985] P. Devijver. Baum's forward backward algorithm revisited. *Pattern Recognition Letters*, 3:369–373, 1985.

[Dickson, 1978] P. Dickson. *The Official Rules*. Dell, 1978.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 31–37, 1994.

[Elfes, 1989] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, pages 46–57, 6 1989.

[Engelson and McDermott, 1992] S. Engelson and D. McDermott. Error correction in mobile robot map learning. In *Proceedings of the International Conference on Robotics and Automation*, pages 2555–2560, 1992.

[Erdmann, 1984] M. Erdmann. On motion planning with uncertainty. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge (Massachusetts), 1984.

[Erdmann, 1989] M. Erdmann. *On Probabilistic Strategies for Robot Tasks*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge (Massachusetts), 1989.

[Erdmann, 1992] M. Erdmann. Randomization in robot tasks. *The International Journal of Robotics Research*, 11(5):399–436, 1992.

[Etzioni, 1991] O. Etzioni. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence*, 49(1-3):129–159, 1991.

[Farquhar and Nakamura, 1988] P. Farquhar and Y. Nakamura. Utility assessment procedures for polynomial-exponential functions. *Naval Research Logistics*, 35:597–613, 1988.

[Farquhar, 1984] P. Farquhar. Utility assessment methods. *Management Science*, 30(11):1283–1300, 1984.

[Feldman and Sproull, 1977] J. Feldman and R. Sproull. Decision theory and Artificial Intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1977.

[Feller, 1966] W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, second edition, 1966.

[Fernandez-Gaucherand and Marcus, 1997] E. Fernandez-Gaucherand and S. Marcus. Risk-sensitive optimal control of hidden Markov models: Structural results. *IEEE Transactions on Automatic Control*, in press, 1997.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Filar *et al.*, 1989] J. Filar, L. Kallenberg, and H.-M. Lee. Variance-penalized Markov decision processes. *Mathematics of Operations Research*, 14(1):147–161, 1989.

[Fink, 1995] E. Fink. Design of representation-changing algorithms. Technical Report CMU–CS–95–120, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1995.

[Foux *et al.*, 1993] G. Foux, M. Heymann, and A. Bruckstein. Two-dimensional robot navigation among unknown stationary polygonal obstacles. *IEEE Transactions on Robotics and Automation*, 9(1):96–102, 1993.

[Gardner, 1973] M. Gardner. Mathematical games. *Scientific American*, 228(1):108–115, 1 1973.

[Genesereth and Nilsson, 1986] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1986.

[Genesereth and Nourbakhsh, 1993] M. Genesereth and I. Nourbakhsh. Time-saving tips for problem solving with incomplete information. In *Proceedings of the National Conference on Artificial Intelligence*, pages 724–730, 1993.

[Goldberg, 1990] K. Goldberg. *Stochastic Plans for Robotic Manipulation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1990. Available as Technical Report CMU-CS-90-161.

[Goldman and Boddy, 1994] R. Goldman and M. Boddy. Epsilon-safe planning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 253–261, 1994.

[Good, 1971] I. Good. Twenty-seven principles of rationality. In V. Godambe and D. Sprott, editors, *Foundations of Statistical Inference*. Holt, Rinehart, Winston, 1971.

[Goodwin and Simmons, 1992] R. Goodwin and R.G. Simmons. Rational handling of multiple goals for mobile robots. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 70–77, 1992.

[Goodwin, 1994] R. Goodwin. Reasoning about when to start acting. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 86–91, 1994.

[Goodwin, 1997] R. Goodwin. *Meta-Level Control for Decision-Theoretic Planners*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1997. Available as Technical Report CS-96-186.

[Haddawy and Hanks, 1990] P. Haddawy and S. Hanks. Issues in decision-theoretic planning: Symbolic goals and numeric utilities. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Haddawy and Hanks, 1992] P. Haddawy and S. Hanks. Representation for decision-theoretic planning: Utility functions for deadline goals. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[Haddawy and Hanks, 1993] P. Haddawy and S. Hanks. Utility models for goal-directed decision-theoretic planners. Technical Report 93–06–04, Department of Computer Science and Engineering, University of Washington, Washington (Seattle), 1993.

[Haddawy *et al.*, 1995] P. Haddawy, A. Doan, and R. Goodwin. Efficient decision-theoretic planning: Techniques and empirical analysis. In *Proceedings of the Annual Conference on Uncertainty in Artificial Intelligence*, pages 229–236, 1995.

[Haigh and Veloso, 1996] K. Haigh and M. Veloso. Interleaving planning and robot execution for asynchronous user requests. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 148–155, 1996.

[Haigh, 1995] K. Haigh. Using planning and execution experience for high-level robot learning. Phd thesis proposal, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1995.

[Hamidzadeh and Shekhar, 1991] B. Hamidzadeh and S. Shekhar. Dynora: A real-time planning algorithm to meet response-time constraints in dynamic environments. In *Proceedings of the International Conference on Tools for Artificial Intelligence*, pages 228–235, 1991.

[Hamidzadeh and Shekhar, 1993] B. Hamidzadeh and S. Shekhar. Specification and analysis of real-time problem solvers. *IEEE Transactions on Software Engineering*, 19(8):788–803, 1993.

[Hamidzadeh, 1992] B. Hamidzadeh. Can real-time search algorithms meet deadlines? In *Proceedings of the National Conference on Artificial Intelligence*, pages 486–491, 1992.

[Hanks, 1990] S. Hanks. *Projecting Plans for Uncertain Worlds*. PhD thesis, Department of Computer Science, Yale University, New Haven (Connecticut), 1990. Available as Technical Report YALE/DCS/TR756.

[Hannaford and Lee, 1991] B. Hannaford and P. Lee. Hidden Markov model analysis of force/torque information in telemanipulation. *The International Journal of Robotics Research*, 10(5):528–539, 1991.

[Hansson *et al.*, 1990] O. Hansson, A. Mayer, and S. Russell. Decision-theoretic planning in BPS. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, 1990.

[Hauskrecht, 1997] M. Hauskrecht. *Planning and Control in Stochasic Domains with Imperfect Information*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Boston (Massachusetts), 1997.

[Hayes and Simon, 1976] J. Hayes and H. Simon. The understanding process: Problem isomorphs. *Cognitive Psychology*, 8:165–190, 1976.

[Heger, 1994] M. Heger. Consideration of risk in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 105–111, 1994.

[Heger, 1996] M. Heger. The loss from imperfect value functions in expectation-based and minimax-based tasks. *Machine Learning*, 22(1–3):197–225, 1996.

[Hernandez-Hernandez and Marcus, 1997] D. Hernandez-Hernandez and S. Marcus. Risk-sensitive control of Markov processes in countable state space. *Systems and Control Letters*, in press, 1997.

[Hierholzer, 1873] C. Hierholzer. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6:30–32, 1873.

[Horvitz *et al.*, 1989] E. Horvitz, G. Cooper, and D. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1121–1127, 1989.

[Howard and Matheson, 1972] R. Howard and J. Matheson. Risk-sensitive Markov decision processes. *Management Science*, 18(7):356–369, 1972.

[Howard, 1964] R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, third edition, 1964.

[Huang *et al.*, 1990] X. Huang, Y. Ariki, and M. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

[Ishida and Korf, 1991] T. Ishida and R. Korf. Moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210, 1991.

[Ishida and Shimbo, 1996] T. Ishida and M. Shimbo. Improving the learning efficiencies of real-time search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 305–310, 1996.

[Ishida, 1992] T. Ishida. Moving target search with intelligence. In *Proceedings of the National Conference on Artificial Intelligence*, pages 525–532, 1992.

[Ishida, 1995] T. Ishida. Two is not always better than one: Experiences in real-time bidirectional search. In *Proceedings of the International Conference on Multi-Agent Systems*, pages 185–192, 1995.

[Ishida, 1997] T. Ishida. *Real-Time Search for Learning Autonomous Agents*. Kluwer Academic Publishers, 1997.

[Iyengar *et al.*, 1986] S. Iyengar, C. Jorgensen, S. Rao, and C. Weisbin. Robot navigation algorithms using learned spatial graphs. *Robotica*, 4:93–100, 1986.

[Jacobson, 1973] D. Jacobson. Optimal stochastic linear systems with exponential performance criteria and their relation to deterministic differential games. *IEEE Transactions on Automatic Control*, 18:124–131, 1973.

[Kaelbling *et al.*, 1986] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1986.

[Kaelbling, 1990] L. Kaelbling. *Learning in Embedded Systems*. MIT Press, 1990.

[Kahneman and Tversky, 1979] D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. *Econometrica*, 47:263–291, 1979.

[Kalman, 1960] R. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the American Society of Mechanical Engineers: Journal of Basic Engineering*, 82:35–45, 1960.

[Kanazawa and Dean, 1989] K. Kanazawa and T. Dean. A model for projection and action. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 985–990, 1989.

[Kaplan and Simon, 1990] C. Kaplan and H. Simon. In search of insight. *Cognitive Psychology*, 22:374–419, 1990.

[Karakoulas, 1993] G. Karakoulas. A machine learning approach to planning for economic systems. In *Proceedings of the Third International Workshop on Artificial Intelligence in Economics and Management*, 1993.

[Keeney and Raiffa, 1976] R. Keeney and H. Raiffa. *Decisions with Multiple Objectives : Preferences and Value Tradeoffs*. Wiley, 1976.

[Knight, 1993] K. Knight. Are many reactive agents better than a few deliberative ones? In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 432–437, 1993.

[Koenig and Simmons, 1992] S. Koenig and R.G. Simmons. Complexity analysis of real-time reinforcement learning applied to finding shortest paths in deterministic domains. Technical Report CMU–CS–93–106, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1992.

[Koenig and Simmons, 1993a] S. Koenig and R.G. Simmons. Complexity analysis of real-time reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 99–105, 1993.

[Koenig and Simmons, 1993b] S. Koenig and R.G. Simmons. Exploration with and without a map. In *Proceedings of the AAAI Spring Symposium on Learning Action Models*, pages 28–32, 1993. Available as AAAI Technical Report WS-93-06.

[Koenig and Simmons, 1994a] S. Koenig and R.G. Simmons. How to make reactive planners risk-sensitive. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 293–298, 1994.

[Koenig and Simmons, 1994b] S. Koenig and R.G. Simmons. Risk-sensitive planning with probabilistic decision graphs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 2301–2308, 1994.

[Koenig and Simmons, 1995a] S. Koenig and R.G. Simmons. The effect of representation and knowledge on goal-directed exploration with reinforcement learning algorithms: The proofs. Technical Report CMU–CS–95–177, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1995.

[Koenig and Simmons, 1995b] S. Koenig and R.G. Simmons. Real-time search in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1660–1667, 1995.

[Koenig and Simmons, 1996a] S. Koenig and R.G. Simmons. Easy and hard testbeds for real-time search algorithms. In *Proceedings of the National Conference on Artificial Intelligence*, pages 279–285, 1996.

[Koenig and Simmons, 1996b] S. Koenig and R.G. Simmons. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, 22(1/3):227–250, 1996.

[Koenig and Simmons, 1996c] S. Koenig and R.G. Simmons. The influence of domain properties on the performance of real-time search algorithms. Technical Report CMU–CS–96–115, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1996.

[Koenig and Simmons, 1996d] S. Koenig and R.G. Simmons. Modeling risk and soft deadlines for robot navigation. In *Proceedings of the AAAI Spring Symposium Series, Symposium on Planning with Incomplete Information for Robot Problems*, pages 57–61, 1996. Available as AAAI Technical Report SS-96-04.

[Koenig and Simmons, 1996e] S. Koenig and R.G. Simmons. Passive distance learning for robot navigation. In *Proceedings of the International Conference on Machine Learning*, pages 266–274, 1996.

[Koenig and Simmons, 1996f] S. Koenig and R.G. Simmons. Unsupervised learning of probabilistic models for robot navigation. In *Proceedings of the International Conference on Robotics and Automation*, pages 2301–2308, 1996.

[Koenig and Simmons, 1997] S. Koenig and R.G. Simmons. A robot navigation architecture based on partially observable Markov decision process models. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*. MIT Press, 1997.

[Koenig and Smirnov, 1996] S. Koenig and Y. Smirnov. Graph learning with a nearest neighbor approach. In *Proceedings of the Conference on Computational Learning Theory*, pages 19–28, 1996.

[Koenig *et al.*, 1996] S. Koenig, R. Goodwin, and R.G. Simmons. Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. In L. Dorst, M. van Lambalgen, and R. Voorbraak, editors, *Reasoning with Uncertainty in Robotics*, volume 1093 of *Lecture Notes in Artificial Intelligence*, pages 322–337. Springer, 1996.

[Koenig, 1991] S. Koenig. Optimal probabilistic and decision-theoretic planning using Markovian decision theory. Master's thesis, Computer Science Department, University of California at Berkeley, Berkeley (California), 1991. Available as Technical Report UCB/CSD 92/685.

[Koenig, 1992] S. Koenig. The complexity of real-time search. Technical Report CMU–CS–92–145, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1992.

[Koenig, 1995] S. Koenig. Agent-centered search: Situated search with small look-ahead. Phd thesis proposal, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1995.

[Koenig, 1996] S. Koenig. Agent-centered search: Situated search with small look-ahead. In *Proceedings of the National Conference on Artificial Intelligence*, page 1365, 1996.

[Kohavi, 1978] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, second edition, 1978.

[Korach *et al.*, 1990] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Transactions on Programming Languages and Systems*, 12(1):84–101, 1990.

[Korf, 1987] R. Korf. Real-time heuristic search: First results. In *Proceedings of the National Conference on Artificial Intelligence*, pages 133–138, 1987.

[Korf, 1988] R. Korf. Real-time heuristic search: New results. In *Proceedings of the National Conference on Artificial Intelligence*, pages 139–144, 1988.

[Korf, 1990] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[Korf, 1993] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[Kortenkamp and Weymouth, 1994] D. Kortenkamp and T. Weymouth. Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the National Conference on Artificial Intelligence*, pages 979–984, 1994.

[Kosaka and Kak, 1992] A. Kosaka and A. Kak. Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 2177–2186, 1992.

[Kuipers and Byun, 1988] B. Kuipers and Y. Byun. A robust, qualitative method for robot spatial learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 774–779, 1988.

[Kuipers and Levitt, 1988] B. Kuipers and T. Levitt. Navigation and mapping in large-scale space. *AI Magazine*, 9(2):25–43, 1988.

[Kushmerick *et al.*, 1995] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.

[Landay, 1995] J. Landay. Interactive sketching for user interface design. In *Proceedings of the CHI (Computer-Human Interaction)*, pages 63–64, 1995.

[Latombe *et al.*, 1991] J.-C. Latombe, A. Lazanas, and S. Shekhar. Robot motion planning with uncertainty in control and sensing. *Artificial Intelligence*, 52:1–47, 1991.

[Latombe, 1991] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[LaValle and Hutchinson, 1994] S. LaValle and S. Hutchinson. An objective-based stochastic framework for manipulation planning. In *Proceedings of the International Conference on Robotics and Automation*, pages 1772–1779, 1994.

[Leonard *et al.*, 1992] J. Leonard, H. Durrant-Whyte, and I. Cox. Dynamic map building for an autonomous mobile robot. *International Journal of Robotics Research*, 11(4):286–298, 1992.

[Lin, 1992] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8(3/4):293–321, 1992.

[Lin, 1993] L.-J. Lin. *Reinforcement Learning for Robots using Neural Networks*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1993. Available as Technical Report CMU-CS-93-103.

[Littman and Boyan, 1993] M. Littman and J. Boyan. A distributed reinforcement learning scheme for network routing. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*, pages 45–51, 1993.

[Littman *et al.*, 1995a] M. Littman, A. Cassandra, and L. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the International Conference on Machine Learning*, pages 362–370, 1995.

[Littman *et al.*, 1995b] M. Littman, T. Dean, and L. Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Annual Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

[Littman, 1996] M. Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, Providence (Rhode Island), 1996. Available as Technical Report CS-96-09.

[Loui, 1983] R. Loui. Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, 26:670–676, 1983.

[Lovejoy, 1991] W. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[Lozano-Perez *et al.*, 1984] T. Lozano-Perez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, 1984.

[Lumelsky *et al.*, 1990] V. Lumelsky, S. Mukhopadhyay, and K. Sun. Dynamic path planning in sensor-based terrain acquisition. *IEEE Transactions on Robotics and Automation*, 6(4):462–472, 1990.

[Lumelsky, 1987] V. Lumelsky. Algorithmic and complexity issues of robot motion in an uncertain environment. *Journal of Complexity*, 3:146–182, 1987.

[Marcus *et al.*, 1997] S. Marcus, E. Fernàndez-Gaucherand, D. Hernàndez-Hernàndez, S. Colaruppi, and P. Fard. Risk-sensitive Markov decision processes. In C. Byrnes et. al., editor, *Systems and Control in the Twenty-First Century*, pages 263–279. Birkhauser, 1997.

[Mataric, 1990] M. Mataric. Environment learning using a distributed representation. In *Proceedings of the International Conference on Robotics and Automation*, pages 402–406, 1990.

[Matsubara and Ishida, 1994] S. Matsubara and T. Ishida. Real-time planning by interleaving real-time search with subgoaling. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 122–127, 1994.

[McCallum, 1995a] A. McCallum. Instance-based state identification for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 377–384, 1995.

[McCallum, 1995b] R. McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the International Conference on Machine Learning*, pages 387–395, 1995.

[Mine and Osaki, 1970] H. Mine and S. Osaki. *Markovian Decision Processes*. Elsevier, 1970.

[Monahan, 1982] G. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[Moore and Atkeson, 1993] A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[Moore and Atkeson, 1995] A. Moore and C. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, 1995.

[Morgenstern, 1987] L. Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 83–88, 1987.

[Motwani and Raghavan, 1995] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Newell, 1965] A. Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. Tualbee, editors, *Electronic Information Handling*. Spartan, 1965.

[Newell, 1966] A. Newell. On the representations of problems. In *Computer Science Research Reviews*. Carnegie Institute of Technology, 1966.

[Newman, 1953] J. Newman. Leonhard Euler and the Königsberg bridges. *Scientific American*, 188(6):66–70, 1953.

[Nilsson, 1971] N. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[Nourbakhsh and Genesereth, 1996] I. Nourbakhsh and M. Genesereth. Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots Journal*, 3(1):49–67, 1996.

[Nourbakhsh and Genesereth, 1997] I. Nourbakhsh and M. Genesereth. Teaching AI with robots. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*. MIT Press, 1997.

[Nourbakhsh *et al.*, 1995] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: An office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[Nourbakhsh, 1996] I. Nourbakhsh. *Interleaving Planning and Execution*. PhD thesis, Department of Computer Science, Stanford University, Stanford (California), 1996.

[Nourbakhsh, 1997] I. Nourbakhsh. *Interleaving Planning and Execution for Autonomous Robots*. Kluwer Academic Publishers, 1997.

[Olawsky *et al.*, 1993] D. Olawsky, K. Krebsbach, and M. Gini. An analysis of sensor-based task planning. Technical Report 93-94, Computer Science Department, University of Minnesota, Minneapolis (Minnesota), 1993.

[Oommen *et al.*, 1987] J. Oommen, S. Iyengar, N. Rao, and R. Kashyap. Robot navigation in unknown terrains using learned visibility graphs. Part I: The disjoint convex obstacle case. *IEEE Journal of Robotics and Automation*, 3:672–681, 1987.

[Papadimitriou and Tsitsiklis, 1987] C. Papadimitriou and J. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[Papadimitriou and Yannakakis, 1991] C. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[Parr and Russell, 1995] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1088–1094, 1995.

[Pearl, 1985] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[Pemberton and Korf, 1992] J. Pemberton and R. Korf. Incremental path planning on graphs with cycles. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 179–188, 1992.

[Pemberton and Korf, 1994] J. Pemberton and R. Korf. Incremental search algorithms for real-time decision making. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 140–145, 1994.

[Pemberton, 1995] J. Pemberton. *Incremental Search Methods for Real-Time Decision Making*. PhD thesis, Computer Science Department, University of California at Los Angeles, Los Angeles (California), 1995.

[Peng and Williams, 1992] J. Peng and R. Williams. Efficient learning and planning within the DYNA framework. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 281–290, 1992.

[Pirzadeh and Snyder, 1990] A. Pirzadeh and W. Snyder. A unified solution to coverage and search in explored and unexplored terrains using indirect control. In *Proceedings of the International Conference on Robotics and Automation*, pages 2113–2119, 1990.

[Pratt, 1964] J. Pratt. Risk aversion in the small and in the large. *Econometrica*, 32(1-2):122–136, 1964.

[Rabiner, 1986] L. Rabiner. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–16, 1 1986.

[Rao *et al.*, 1991] N. Rao, N. Stoltzfus, and S. Iyengar. A "retraction" method for learned navigation in unknown terrains for a circular robot. *IEEE Transactions on Robotics and Automation*, 7(5):699–707, 1991.

[Reinefeld, 1993] A. Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in IDA*. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 248–253, 1993.

[Rencken, 1995] W. Rencken. Autonomous sonar navigation in indoor, unknown, and unstructured environments. In V. Graefe, editor, *Intelligent Robots and Systems*. Elsevier, 1995.

[Ring, 1992] M. Ring. Two methods for hierarchy learning in reinforcement environments. In *Proceedings of the International Conference on Simulation of Adaptive Behavior*, pages 148–155, 1992.

[Russell and Wefald, 1991] S. Russell and E. Wefald. *Do the Right Thing – Studies in Limited Rationality*. MIT Press, 1991.

[Russell and Zilberstein, 1991] S. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 212–217, 1991.

[Sanborn and Hendler, 1988] J. Sanborn and J. Hendler. Near-term event projection through dynamic simulation. In *Proceedings of the SCS Multiconference on Artificial Intelligence and Simulation: The Diversity of Applications*, pages 194–198, 1988.

[Schapire, 1992] R. Schapire. *The Design and Analysis of Efficient Learning Algorithms*. MIT Press, 1992.

[Schoppers, 1987] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Shakhar and Hamidzadeh, 1993] S. Shakhar and B. Hamidzadeh. Evaluation of real-time problem solvers in dynamic environments. *International Journal on Artificial Intelligence Tools (Architectures, Languages, Algorithms)*, 2(4):459–484, 1993.

[Shatkay and Kaelbling, 1997] H. Shatkay and L. Kaelbling. Learning topological maps with weak local odometric information. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.

[Shekhar and Dutta, 1989] S. Shekhar and S. Dutta. Minimizing response times in real-time planning and search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 238–242, 1989.

[Shekhar and Hamidzadeh, 1992] S. Shekhar and B. Hamidzadeh. Evaluation of real-time search algorithms in dynamic worlds (summary of results). In *Proceedings of the International Conference on Tools for Artificial Intelligence*, pages 6–13, 1992.

[Simmons and Koenig, 1995] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1080–1087, 1995.

[Simmons *et al.*, 1995] R. Simmons, E. Krotkov, L. Chrisman, F. Cozman, R. Goodwin, M. Hebert, L. Katragadda, S. Koenig, G. Krishnaswamy, Y. Shinoda, W. Whittaker, and P. Klarer. Experience with rover navigation for lunar-like terrains. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 441–446, 1995.

[Simmons *et al.*, 1996] R. Simmons, S. Thrun, G. Armstrong, R. Goodwin, K. Haigh, S. Koenig, S. Mahamud, D. Nikovski, and J. O'Sullivan. Amelia. In *Proceedings of the National Conference on Artificial Intelligence*, page 1358, 1996.

[Simmons *et al.*, 1997] R.G. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O'Sullivan. A layered architecture for office delivery robots. In *Proceedings of the International Conference on Autonomous Agents*, 1997.

[Simmons, 1994a] R. Simmons. Becoming increasingly reliable. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 152–157, 1994.

[Simmons, 1994b] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[Simmons, 1995] R. Simmons. The 1994 AAAI robot competition and exhibition. *AI Magazine*, 16(2):19–30, 1995.

[Simmons, 1996] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proceedings of the International Conference on Robotics and Automation*, pages 3375–3382, 1996.

[Singh, 1992] S. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the National Conference on Artificial Intelligence*, pages 202–207, 1992.

[Smirnov and Veloso, 1997] Y. Smirnov and M. Veloso. Gensat: A navigational approach. In *Proceedings of the Portuguese Conference on Artificial Intelligence*, 1997.

[Smirnov, 1997] Y. Smirnov. *Improving Search Efficiency through Cross-Fertilization among Artificial Intelligence, Theoretical Computer Science and Operations Research*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1997. Available as Technical Report CMU-CS-97-171.

[Smith and Cheeseman, 1986] R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5:56–68, 1986.

[Smith *et al.*, 1990] R. Smith, M. Self, and P. Cheeseman. Estimating uncertain spatial relationships in robotics. In I. Cox and G. Wilfong, editors, *Autonomous Robot Vehicles*, pages 167–193. Springer, 1990.

[Smith, 1988] D. Smith. A decision-theoretic approach to the control of planning search. Technical Report LOGIC-87-11, Department of Computer Science, Stanford University, Stanford (California), 1988.

[Sondik, 1978] E. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304, 1978.

[Stentz and Hebert, 1995] A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145, 1995.

[Stentz, 1995] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[Stolcke and Omohundro, 1993] A. Stolcke and S. Omohundro. Hidden Markov model induction by Bayesian model merging. In *Advances in Neural Information Processing Systems*, pages 11–18, 1993.

[Stone and Veloso, 1996] P. Stone and M. Veloso. User-guided interleaving of planning and execution. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning: Proceedings of the European Workshop on Planning*, pages 103–112. IOS-Press, 1996.

[Sutherland, 1969] I. Sutherland. A method for solving arbitrary-wall mazes by computer. *IEEE Transactions on Computers*, C–18(12):1092–1097, 1969.

[Sutton, 1990] R. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the International Conference on Machine Learning*, pages 216–224, 1990.

[Sutton, 1991] R. Sutton. DYNA, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2(4):160–163, 1991.

[Tash and Russell, 1994] J. Tash and S. Russell. Control strategies for a stochastic planner. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1079–1085, 1994.

[Tenenberg *et al.*, 1992] J. Tenenberg, J. Karlsson, and S. Whitehead. Learning via task decomposition. In *Proceedings of the Conference on "From Animals to Animats"*, pages 337–343, 1992.

[Thrun, 1992a] S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1992.

[Thrun, 1992b] S. Thrun. The role of exploration in learning control with neural networks. In D. White and D. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold, 1992.

[Thrun, 1993] S. Thrun. Exploration and model building in mobile robot domains. In *Proceedings of the International Conference on Neural Networks*, pages 175–80, 1993.

[Thrun, 1996] S. Thrun. A Bayesian approach to landmark discovery in mobile robot navigation. Technical Report CMU-CS-96-122, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania), 1996.

[Toulet, 1986] C. Toulet. An axiomatic model of unbounded utility functions. *Mathematics of Operations Research*, 11(1):81–94, 1986.

[Veloso *et al.*, 1995] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, 7(1):81–120, 1995.

[Viterbi, 1967] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, 1967.

[von Neumann and Morgenstern, 1947] J. von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, second edition, 1947.

[Wagner *et al.*, 1997] I. Wagner, M. Lindenbaum, and A. Bruckstein. On-line graph searching by a smell-oriented vertex process. In *Proceedings of the AAAI Workshop on On-Line Search*, 1997.

[Watkins and Dayan, 1992] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[Watkins, 1989] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge University, Cambridge (Great Britain), 1989.

[Watson and Buede, 1987] S. Watson and D. Buede. *Decision Synthesis*. Cambridge University Press, 1987.

[Wellman and Doyle, 1991] M. Wellman and J. Doyle. Preferential semantics for goals. In *Proceedings of the National Conference on Artificial Intelligence*, pages 698–703, 1991.

[Wellman and Doyle, 1992] M. Wellman and J. Doyle. Modular utility representation for decision theoretic planning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 236–242, 1992.

[Wellman *et al.*, 1995] M. Wellman, M. Ford, and K. Larson. Path planning under time-dependent uncertainty. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 523–539, 1995.

[Wellman, 1990] M. Wellman. *Formulation of Tradeoffs in Planning under Uncertainty*. Pitman, 1990.

[White, 1991] C. White. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 1991.

[Whitehead, 1991a] S. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 607–613, 1991.

[Whitehead, 1991b] S. Whitehead. A study of cooperative mechanisms for faster reinforcement learning. Technical Report 365, Department of Computer Science, University of Rochester, Rochester (New York), 1991.

[Whitehead, 1992] S. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD thesis, Department of Computer Science, University of Rochester, Rochester (New York), 1992.

[Whittle, 1990] P. Whittle. *Risk-Sensitive Optimal Control*. Wiley, 1990.

[Wiering and Schmidhuber, 1996] M. Wiering and J. Schmidhuber. Solving POMDPs with levin search and EIRA. In *Proceedings of the International Conference on Machine Learning*, pages 534–542, 1996.

[Williamson and Hanks, 1994] M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 176–181, 1994.

[Williamson and Hanks, 1996] M. Williamson and S. Hanks. Flaw selection strategies for value-directed planning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, 1996.

[Yang *et al.*, 1994] J. Yang, Y. Xu, and C. Chen. Hidden Markov model approach to skill learning and its application to telerobotics. *IEEE Transactions on Robotics and Automation*, 10(5):621–631, 1994.

[Young, 1994] S. Young. The HTK hidden Markov model toolkit: Design and philosophy. Technical Report CUED/F–INFENG/TR.152, Engineering Department, Cambridge University, Cambridge (Great Britain), 1994.

[Zelinsky, 1992] A. Zelinsky. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, 8(6):707–717, 1992.

[Zilberstein, 1993] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Department, University of California at Berkeley, Berkeley (California), 1993.

*** THE END ***

There are bound to be mistakes in a document of this size.
I will maintain a list of errata. Just ask for it.