Single Agent and Multi-agent Path Planning in Unknown and Dynamic Environments

Dave Ferguson

CMU-RI-TR-06-54

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

September, 2006

Thesis Committee:
Anthony Stentz, Co-chair
Sebastian Thrun, Co-chair
James Kuffner
Steve LaValle, University of Illinois
Wolfram Burgard, University of Freiburg

Carnegie Mellon

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Thesis

Single Agent and Multi-agent Path Planning in Unknown and Dynamic Environments

Dave Ferguson

Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the field of Robotics

ACCEPTED:

Anthony Stentz, Thesis Committee Co-chair	9-25-06 Date
Sebastian Thrun, Thesis Committee Co-chair	9-26-06 Date
Reid G. Simmons, Program Chair	October 18, 2006 Date

APPROVED:

Randal E. Bryant, Dean	10/18/06 Date

Contents

List of Figures

List of Tables

Abstract

As autonomous agents make the transition from solving simple, well-behaved problems to being useful entities in the real world, they must deal with the added complexity and uncertainty inherent in real environments. In particular, agents navigating through the real world can be confronted with imperfect information (e.g. when prior maps are absent or incomplete), limited deliberation time (e.g. when agents need to act quickly), and dynamic elements (e.g. when there are humans or other agents in the environment). This thesis addresses the problem of path planning and replanning in realistic scenarios involving these three challenges.

For single agent planning we present a set of discrete search algorithms that efficiently repair the current solution when new information is received concerning the environment. We also introduce an approach that is able to provide better solutions through reasoning about the uncertainty in the initial information held by the agent. To cope with both imperfect information and limited deliberation time, we provide an additional algorithm that is able to improve the solution while time allows and repair the solution when new information is received. We further show how this algorithm can be used to plan and replan paths in dynamic environments.

For multi-agent planning we present a set of sampling-based search algorithms that provide similar behavior to the above approaches but that can handle much higher dimensional search spaces. These sampling-based algorithms extend current approaches to perform efficient repair when new information is received and to provide higher quality solutions given limited deliberation time. We show how our culminating algorithm, which is able to both improve and repair its solution over time, can be used for multi-agent planning and replanning in dynamic environments.

Together, the collection of planning algorithms introduced in this thesis enable single agents and multi-agent teams to navigate and coordinate in a wide range of realistic scenarios.

Funding Sources

This work was sponsored by a National Science Foundation Graduate Research Fellowship and the U.S. Army Research Laboratory contract "Robotics Collaborative Technology Alliance" (contract number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the U.S. Government.

For Sance, who will always be my fellow Chumpion

Acknowledgements

First of all, I would like to thank my advisors, Tony Stentz and Sebastian Thrun, for their support and guidance throughout this work. You have both been great teachers and friends. Sebastian, your enthusiasm is contagious. Thank you for letting me send my first year's worth of research into a hole in the ground and for having the confidence in me that it would come back out. Tony, you are a fantastic mentor. Thank you for bearing with me through all my crazy ideas and for your excellent advice on all topics. Together, you have had a huge influence on my life – thanks for making the last four years so enjoyable and rewarding.

I would also like to thank my committee members James Kuffner, Steve LaValle, and Wolfram Burgard, and all the other people that have contributed to this work and collaborated with me on various parts of it: Nidhi Kalra (Dynamic RRTs and Constrained Exploration), James Kuffner (Anytime D* with PRMs), Jur van den Berg (Anytime D* with PRMs), Geoff Gordon (Anytime D*), Joseph Carsten (3D Field D* and Field D* on the Mars rovers), Art Rankin and Mark Maimone (Field D* on the Mars rovers), Sascha Kolski and Roland Siegwart (autonomous navigation with the Smart vehicles). I owe special thanks to Max Likhachev for all the algorithms we developed together (in particular, Anytime D*) and the lengthy, stimulating discussions around various whiteboards. The system implementations would not have been possible without the help of Bryan Nagy, the PerceptOR and UPI teams, Marc Zinck, Juan Pablo Gonzalez, Boris Sofman, and Freddie Dias.

My undergraduate advisors Margaret Jefferies and Willem Labuschagne played a large part in my decision to pursue a Ph.D. in robotics, and I am extremely grateful for their encouragement and the opportunities they provided for me when I was eager but clueless. Margaret, you will be sorely missed.

The subterranean robotics group provided amazing challenges and excitement the first couple years of my time at CMU. Thank you to all the members for having me along for quite a ride: Red Whittaker, Scott Thayer, Sebastian Thrun, Chuck Whittaker, Chris Baker, Dirk Hähnel, Mike Montemerlo, Aaron Morris, Zachary Omohundro, Charlie Reverte, Dave Silver, and Rudi Triebel. Thanks especially to Aaron, Dave, and Chuck for all those magic

moments in Bruceton.

I have enjoyed the last four years immensely, and I have several people to thank for this. Firstly, my various office mates, for putting up with me and my chaos over the years (particularly Zachary Omohundro, who has suffered me for our entire PhDs). My housemates, Nick Patronik, Dave Silver, Yuriy Nevmyvaka, Sanjeev Koppal, Paul Tompkins, Marius Leordeanu, Andrew Stein, Clark Haynes, Caroline Pantofaru, and Sarah Stein, for all the great times and patience in the face of dog hair mountains. Also, all the other great friends the RI, CMU, and Pittsburgh has provided for me—you are too many to list here but I appreciate you all. Thanks for making this such a fantastic place to work and live.

Thanks also to my family for being so incredibly supportive. Mum and Dad, you have been amazing role models. Thanks for letting me climb trees and run off to Pittsburgh. Pete, thanks for being my tag-team all along and for all the laughs (a shark-eating fish?). You guys are the best.

Thanks most of all to Nidhi and Oscar, for making life so bright.

Chapter 1

Introduction

The goal of artificial intelligence is to develop agents that are capable of making smart decisions on their own. When dealing with embodied agents such as robots, there is the exciting additional step of the agents *acting* on these decisions and physically interacting with the environment. A decision of paramount importance is how the agent should move to perform this interaction. The field of path planning addresses this decision-making process; it provides approaches that generate sequences of actions an agent can take from its current location to get to some desired goal location.

This thesis is concerned with improving single agent and multi-agent path planning in real environments. In particular, two motivating example problems are used throughout this work to apply and compare various planning approaches: outdoor mobile robot navigation and multirobot constrained exploration.

Our first example is outdoor mobile robot navigation, in which a robotic agent is tasked with navigating from some initial position in the environment to a desired goal position. Typically, the robot will have onboard sensors that it can use to observe the environment in its vicinity, and it may be equipped with a prior map of the environment. An example outdoor robotic vehicle is shown in Figure 1.1 along with an overhead view of this vehicle navigating across an outdoor environment.

Secondly, we consider the multirobot constrained exploration problem, in which a team of robots is tasked with exploring some key locations in an environment while maintaining some team constraints during their traverse [Kalra et al., 2006]. Specifically, we are concerned with the scenario where the agents maintain line-of-sight communication across the team at all times, for safety or information passing. In essence, the team must maintain an ad-hoc line-of-sight network so that any member of the team could pass a message to any other member by using the rest of the team as relay nodes. Figure 1.2 shows an example scenario involving three agents and sets of locations to be explored by each agent.

Figure 1.1: The Spinner robot developed at the National Robotics Engineering Center navigating through an outdoor environment. On the left is an overhead view of the position of the vehicle, along with the local actions being evaluated (in green and blue), the global path (in yellow), and the safe (black) and obstacle (white) areas of the environment in the vicinity of the vehicle. On the right is an image of the vehicle.

1.1 Planning in the Real World

Planning paths for agents navigating through the real world is significantly more challenging than planning paths for agents operating in nice, well-behaved domains, as it involves dealing with the complexity and uncertainty inherent in real environments. In particular, path planning for agents navigating through the real world can be challenging in any of the following ways.

Incomplete or imperfect information. Often the initial information available to an agent or a team of agents concerning real environments is very limited: there may be no map of the environment or only a low-resolution or partial map. Further, even when agents are given prior map information, it is likely that this information will be imperfect. We would thus like the agents to be able to generate effective initial plans given incomplete information and then update these plans as new information is received.

For instance, imagine a robot navigating an outdoor environment with a low-resolution satellite map of the area. From this map, the robot has some idea of the environment and can plan a path to its destination. However, as the robot traverses this path and acquires more information concerning the environment, it will almost certainly have to modify its path to account for new terrain features. Generating a new path from scratch in such situations can be computationally expensive and, if the robot is constantly receiving new information, may not even be possible. Thus, to efficiently produce a valid path the robot may should intelligently repair its previous solution.

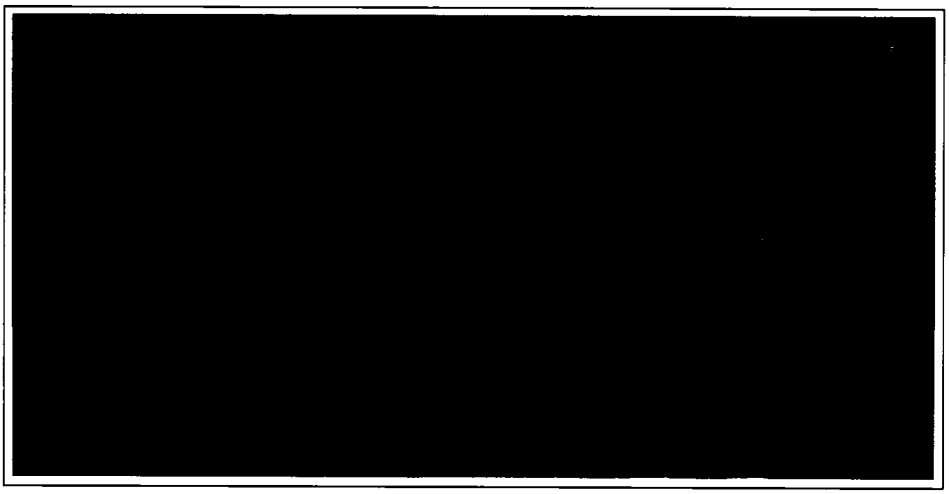


Figure 1.2: An example of constrained exploration with three robots and six key locations to be visited by each robot in an outdoor environment. Each team member must be able to reach any other member through line-of-sight communication, if necessary through using other team members as relay nodes.

Limited deliberation time. When operating in the real world, agents do not have the luxury of unlimited planning time. Rather, they must plan and respond quickly as they interact with the environment. If we are dealing with a single complex agent or a team of agents, performing this planning quickly can be particularly difficult because it involves searching through very large, high-dimensional spaces.

For instance, in our constrained exploration scenario each robot must take into account the actions of each teammate during the planning process to ensure the team line-of-sight constraint is not violated. As a result, planning for the team involves reasoning about the actions of all of the robots together. Solving such complex planning problems can be difficult, and solving them when the time available for planning is limited is particularly challenging.

Dynamic environments. Things can change in real environments over time: objects can be moved and people or other agents can move themselves. Ideally, our agents should take into account any known dynamics during the initial planning process, and they should certainly update their plans based on dynamic changes as they observe them.

For instance, imagine a robot trying to cross a road on which there are cars approaching. If this agent ignores that the cars are dynamic objects and treats them as static objects, then it will plan a path across the road that avoids the cars in their current positions, but obviously executing such a path may be fatal. To safely cross the road, the robot needs to plan a path that takes into account the dynamic nature of the cars.

Each of the challenges listed above needs to be addressed to enable agents to navigate reliably and effectively in the real world. This thesis is aimed at equipping agents with path planning techniques that address these challenges. The contributions of this work are a series of planning algorithms that deal with all the above challenges and provide greatly improved single agent and multi-agent decision making in complex, natural environments.

1.2 Outline of Thesis

This thesis is split into two parts. In the first part of the thesis, we present a number of algorithms effective for solving single agent planning problems such as our mobile robot navigation example. We begin by introducing the range of existing path planning approaches and how they can be used for robot navigation. We then deal with imperfect initial information and present novel approaches that provide improved performance both in terms of solution quality and computational requirements. Next, we discuss algorithms that can be used to plan when deliberation time is limited, and provide a novel algorithm that can cope with both limited deliberation time and imperfect information. Finally, we describe how our new approaches can be used for planning in dynamic environments.

In the second part of the thesis, we present algorithms appropriate for solving multi-agent planning problems, such as our constrained exploration example. We first describe how multi-agent planning relates to single agent planning and describe algorithms suited to the multi-agent path planning problem. We then present novel extensions of these algorithms able to efficiently cope with imperfect initial information. Next, we explain how these approaches can be further modified to provide higher quality solutions and we describe a new algorithm based on these modifications that is able to provide good solutions when planning time is limited. We then combine all these extensions into a single algorithm that is able to both improve and repair its solution over time and is thus well-suited to planning with limited deliberation time and with imperfect initial information. We further show how this algorithm can be used for multi-agent planning in dynamic environments.

We conclude the thesis with a discussion of the various algorithms presented, along with some key concepts shared by the approaches and ideas for future extensions.

1.3 Publication Note

Parts of this thesis have appeared in previous publications. The replanning algorithms presented in Chapter 3 were first introduced in [Ferguson and Stentz, 2005b, Ferguson and Stentz, 2004a, Ferguson and Stentz, 2005a, Ferguson and Stentz, 2006c, Ferguson and Stentz, 2006d]; the uncertainty-based planning algorithms presented in that chapter were introduced in [Ferguson et al., 2004, Ferguson and Stentz, 2004b, Ferguson and Stentz, 2004c].

The anytime replanning algorithm presented in Chapter 4 was introduced in [Likhachev et al., 2005a, Likhachev et al., 2005b] and a guide to the usage of this algorithm and related heuristic-based search algorithms in various applications was provided in [Ferguson et al., 2005]. The extension of this algorithm presented in Chapter 5 used to plan in dynamic environments was introduced in [van den Berg et al., 2006]. Finally, the sampling-based analogs to these algorithms, presented in Chapters 7 and 8, were introduced and applied to the constrained exploration problem in [Ferguson et al., 2006, Kalra et al., 2006, Ferguson and Stentz, 2006a, Ferguson and Stentz, 2006b].

Part I

Single-agent Planning

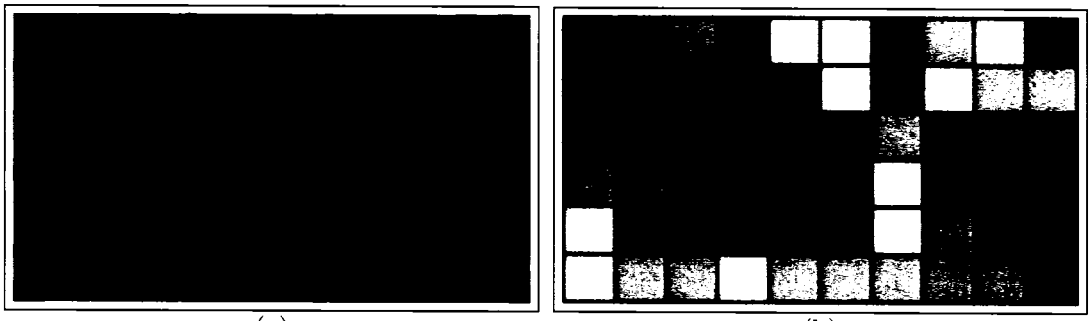
Chapter 2

Single Agent Path Planning

Path planning for mobile robots consists of finding a sequence of state transitions that leads a robot from its initial state to some desired goal state. Typically, the states are robot locations and the transitions represent actions the robot can take, each of which has an associated cost. Since there may be several paths that take the robot to its goal, and several different algorithms that can generate these paths, we tend to distinguish these solutions and algorithms according to certain key properties. A path is said to be *optimal* if the sum of its transition costs is minimal across all possible paths leading from the initial state to the goal state. A planning algorithm is said to be *complete* if it will always find a path in finite time when one exists, and if it will let us know in finite time if no path exists. Similarly, a planning algorithm is optimal if it will always find an optimal path.

In this chapter, we introduce the field of path planning and describe several widely-used path planning algorithms, including both optimal and suboptimal, complete and incomplete algorithms. We begin by introducing discrete algorithms, which work by transforming the planning problem into a search over some discrete representation, such as a graph. Next, we present sampling-based algorithms, which avoid this discretization step and instead plan directly in the original problem space, using sampling to efficiently explore this space and look for solution paths. Finally, we present a number of alternative approaches that abandon the idea of computing paths that will lead an agent all the way to the goal, and instead focus on short range planning that only takes into account the next few actions of the agent.

Throughout the chapter, we use our motivating example of robotic navigation in outdoor environments to illustrate each of the different approaches. We conclude the chapter by discussing which of the general approaches is most applicable to the problems addressed in this thesis.



(a) (b)

Figure 2.1: An outdoor environment and a sample traversability grid for a small section of such an environment. The darker the cell, the more difficult it is to navigate the corresponding area of the environment. The traversability costs of the grid cells are expanded to reflect the dimensions of the vehicle, so that the vehicle can be treated as a single point within the resulting grid.

2.1 Discrete Algorithms

A very common approach in robotic path planning is to take the original planning problem, which may involve continuous environments, continuous actions, and complex kinematic and dynamic constraints, and simplify this problem by discretizing the continuous elements or ignoring the complex constraints.

For instance, consider our example of a robotic vehicle navigating an outdoor environment. One way to represent this environment is as a two-dimensional (2D) traversability grid. This involves splitting the environment up into a grid of cells and assigning a traversal value or label to each cell based on the nature of the environment at the corresponding location: grid cells that correspond to areas obstructed by obstacles or otherwise unnavigable are labeled as obstacles, and the rest are given a cost per unit of traverse (traversal cost) reflecting the difficulty of navigating the corresponding areas of the environment. The basic idea behind such representations is to reduce the continuous nature of the true environment into manageable, discretized chunks. Typically, the grids used are uniform, although several other approaches exist, including exact cell decomposition [Latombe, 1991], quadtrees [Samet, 1982], framed quadtrees [Yahja et al., 1998], and parti-game representations [Moore and Atkeson, 1995, Likhachev and Koenig, 2002]. There are also several other, non-grid based approaches for discretizing the environment, such as visibility graphs, which create a graph connecting all of the obstacle corners and the initial and final locations of the robot, where edges exist between all of these locations that can be reached via a straight-line path from one another [de Berg et al., 2000], and voronoi diagrams, which create a graph that consists of all points that are equidistant from their furthest two obstacles, resulting

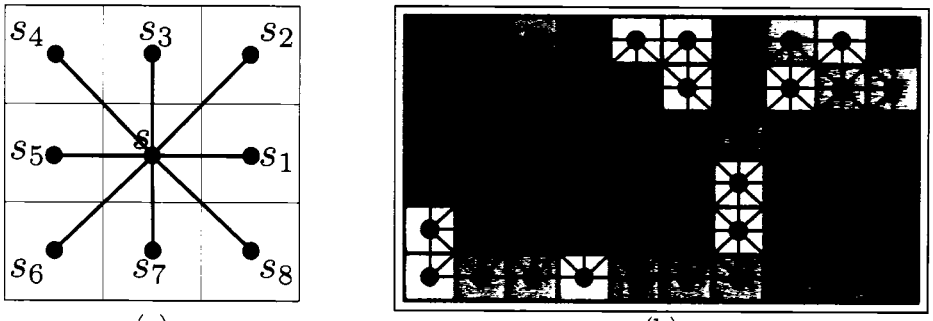


Figure 2.2: (a) A standard 2D grid used for global path planning in which states reside at the centers of the grid cells. The arcs emanating from the center state represent all the possible actions that can be taken from this state. (b) The resulting graph constructed from the example grid in Figure 2.1(b). States are shown as red circles and edges are shown in blue.

in graphs with maximal clearance from the closest obstacles [Choset and Burdick, 1995]. Although the following discussion focuses on grid-based representations, almost all of the above approaches use their representation to construct some sort of graph, which is then searched for a path using the algorithms discussed in the following subsection.

Traversability grids commonly encode the *configuration space* costs, where the configuration space is the space of all possible positions that the agent can take in the environment. This is done by expanding the obstacles and traversability costs to reflect the actual dimensions of the robot. Often conservative estimates are taken (for instance if the robot is larger in some dimension than another [e.g. length vs width], the larger of the dimensions is used to expand obstacles and costs) so that the robot can be treated as a single point in the resulting grid, and planning a path for the robot can be accomplished by generating a trajectory through this grid for a single point. Figure 2.1 illustrates the grid extraction process in action.

Although planning paths through grids is much easier than planning through the original environment, it is still computationally expensive to generate paths that are optimal with respect to the traversability costs within the grid. In Chapter 3 we discuss a number of algorithms that attempt to produce such paths, but it is much more common in robotics to instead focus on basic approximation algorithms that are extremely fast. The most popular such approach is to approximate the traversability grid as a discrete graph, then generate paths over the graph. A common way to do this is to assign a state to each cell center, with edges connecting this state to states in adjacent cell centers. The cost of each edge is a combination of the traversal costs of the two cells it transitions through and the length of the edge. Figure 2.2(a) shows this state and edge extraction process for one cell in a uniform resolution 2D grid, and Figure 2.2(b) shows the resulting graph computed for the

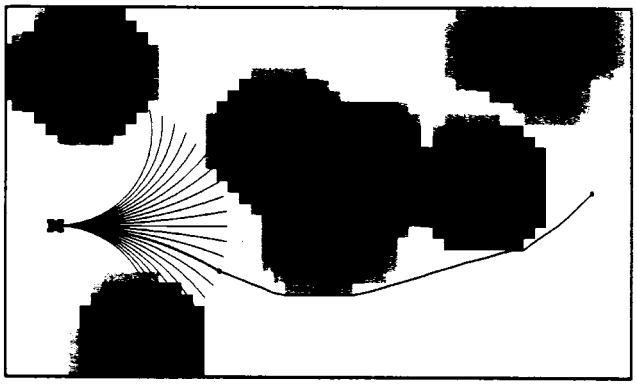


Figure 2.3: Local Planning Combined with Global Planning. The vehicle projects a set of feasible trajectories through the local map from its current position and orientation (arcs for a single speed are shown in red). The cost of each of these trajectories is computed, based on the cost of the cells the arc travels through (darker areas are more expensive, with black areas representing obstacles). A global path is planned from the end of each arc to the goal (shown as a filled yellow and red circle on the right side of the map) and the cost of this path is added to the cost of the arc. The best arc is shown in blue, along with the global path from the end of this arc to the goal. Note that this map represents the configuration space of the vehicle.

example grid shown in Figure 2.1.

Since the paths produced by these approximation techniques do not necessarily encode the kinematic and dynamic constraints of the robot, it may be not be possible for a robot to directly traverse them. To cope with this limitation, in robotics these approximate *global* path planning techniques are typically combined with high-fidelity *local* path planners [Kelly, 1995, Stentz and Hebert, 1995, Brock and Khatib, 1999, Singh et al., 2000, Nagy and Kelly, 2001, Howard and Kelly, 2005, Howard and Kelly, 2006]. The global planners generate paths through a grid or graph that ignore the kinematic and dynamic constraints of the vehicle. Then, the local planner takes into account the constraints of the vehicle and generates a set of feasible local trajectories that can be directly executed from its current position and velocity. To decide which of these trajectories to execute, one common approach is to have the robot evaluate both the cost of each local trajectory and the cost of a global path from the end of each trajectory to the robot's desired goal location. Figure 2.3 illustrates this general idea.

Graph Search

Once we have extracted a graph representing our planning problem, planning a path for the robot consists of searching this graph for a solution path. A number of classical graph search algorithms have been developed for calculating least-cost paths on a weighted graph, with

two popular ones being Dijkstra's algorithm [Dijkstra, 1959], and A* search [Hart et al., 1968, Nilsson, 1980]. Both algorithms return an optimal path given the graph [Gelperin, 1977] and can be considered special forms of Dynamic Programming [Bellman, 1957].

Before describing these algorithms in more detail, it is worth introducing some notation that we will use throughout the first part of this thesis. We use S to denote the finite set of states in the graph, $Succ(s)$ to denote the set of successor states of state $s \in S$, and $Pred(s)$ to denote the set of predecessor states of s. For any pair of states $s, s' \in S$ such that $s' \in Succ(s)$ we denote the cost of transitioning from s to s' (the edge cost) as $c(s, s')$. We also require this cost to be greater than zero.

We assume we are given some initial state s_{start} and we wish to generate a path to a desired goal state s_{goal}. This path can be represented as a sequence of states $P = \{s_0, s_1, \ldots, s_n\}$ such that $s_0 = s_{start}$, $s_n = s_{goal}$, and $s_i \in Succ(s_{i-1})$ for all $0 < i \leq n$. The cost of this path is the cumulative cost of all the transitions it is composed of, i.e., $\sum_{i=0}^{n-1} c(s_i, s_{i+1})$. The cost of a least-cost, or optimal, path from state s to state s' is denoted $c^*(s, s')$.

Optimal algorithms such as Dijkstra's search or A* search outwards from the initial state s_{start} and try to compute a least-cost path from s_{start} to the goal state s_{goal}. They do this by maintaining, for each state s, the cost of the best path found from s_{start} to s. This cost is referred to as $g(s)$. These values are useful because, when they correspond to the optimal values $c^*(s_{start}, s)$ for every state s, an optimal path can be traced backwards from s_{goal} to s_{start} by repeatedly moving from the current state to the predecessor that provides the minimum overall path cost: begin at s_{goal}, and at any state s pick a state $s'' = \arg\min_{s' \in Pred(s)}(g(s') + c(s', s))$ until $s'' = s_{start}$.

It is the task of the search algorithm to ensure that these g-values do in fact correspond to the optimal values, which we refer to as g^*-values. Dijkstra's algorithm works by updating g-values until these values are optimal for all states s. A* search worlds by updating g-values until these values are optimal for all states s that reside upon any least-cost path from s_{start} to s_{goal}.

Dijkstra's algorithm is presented in Figure 2.4. In this pseudocode, *OPEN* is a list of states whose g-values have changed since they were last *expanded*, where we say a state is expanded when it is removed from the list and its successors have their g-values updated (lines 2 through 8). The algorithm begins by inserting the initial state s_{start} into the list *OPEN* as its only element. Next, it continues to expand states s from *OPEN* in increasing order of their path costs $g(s)$ until there are no longer any states left in this list.

When a state s is expanded, each successor s' of s is examined. If s' has not been examined before, its g-value is initialized to infinity. Next, the current path cost of state s', $g(s')$, is compared to the cost of a path through state s, $g(s) + c(s, s')$. If the current path cost of s' is greater than the cost of a path through s, this path cost is updated to reflect

```
ComputePath()
 1 while (OPEN is not empty)
 2   remove s with the smallest g(s) from OPEN;
 3   for each successor s' of s
 4     if s' was never visited before then
 5       g(s') = ∞;
 6     if g(s') > g(s) + c(s, s')
 7       g(s') = g(s) + c(s, s');
 8       insert/update s' in OPEN;
Main()
 9 g(s_start) = 0; OPEN = ∅;
10 insert s_start into OPEN;
11 ComputePath();
```

Figure 2.4: Dijkstra's Algorithm

the cheaper path through s and the state s' is inserted into *OPEN* so that it can propagate this new, lower path cost to its successors.

By expanding states in order of increasing g-values, Dijkstra's algorithm is guaranteed to only expand each state once. Its operation is analogous to a wavefront that propagates outwards from the initial location through the graph. It is thus very efficient for computing optimal paths from state s_{start} to *every* other state in the graph. However, if we are only interested in computing an optimal path to a particular goal state s_{goal}, then the algorithm can be made more efficient by replacing the termination condition (line 1) with the following

1(a) while (s_{goal} is not expanded)

This has the effect of stopping the wavefront as soon as it reaches the goal state. This can make a huge difference in runtime if the graph is large and there exists a relatively cheap path between the initial state and the goal state.

A* search attempts to be even more efficient, by using a heuristic to focus its computation on the most promising areas of the graph. Instead of propagating a wave out in all directions, it biases the movement of the wave so that it concentrates towards states that could reside on least-cost paths from the initial state to the goal state. The basic idea is that, when examining whether a state s could reside on a least-cost path from s_{start} to s_{goal}, it is important to consider not only the cost of a path from s_{start} to s but also the cost of a path from s to s_{goal}. Thus, states s that have low g-values but are very far from s_{goal} (e.g. further than s_{start}) can be ruled out of contention.

To do this, A* employs a heuristic $h(s)$ that estimates, for each state s, the cost of a least-cost path from s to s_{goal}. It then processes states from *OPEN* in increasing order

```
ComputePath()
 1 while (s_goal is not expanded)
 2   remove s with the smallest f(s) from OPEN;
 3   for each successor s' of s
 4     if s' was never visited before then
 5       g(s') = ∞;
 6     if g(s') > g(s) + c(s, s')
 7       g(s') = g(s) + c(s, s');
 8       insert/update s' in OPEN with f(s') = g(s') + h(s');
Main()
 9 g(s_start) = 0; OPEN = ∅;
10 insert s_start into OPEN with f(s_start) = g(s_start) + h(s_start);
11 ComputePath();
```

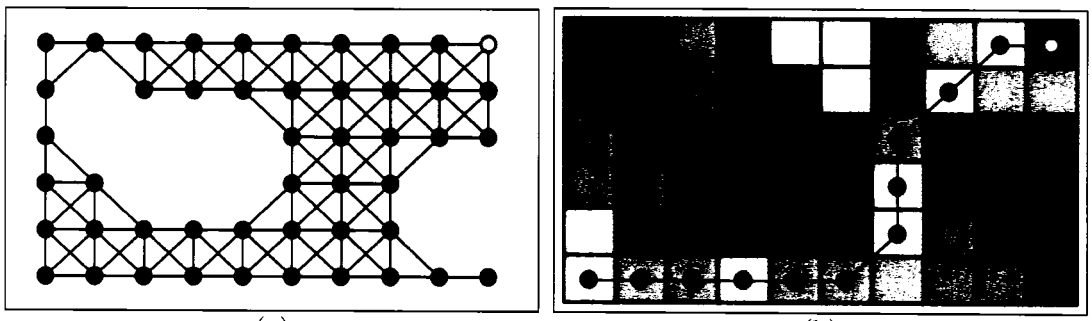
Figure 2.5: A* Search

of their f-values, where $f(s) = g(s) + h(s)$. The algorithm is otherwise identical to the modified version of Dijkstra's algorithm discussed above, and is presented in Figure 2.5.

Figure 2.6 illustrates an example solution path planned through the graph in Figure 2.2, as well as this path overlaid back onto the original grid. Figure 2.7 shows a path planned through a larger, non-uniform cost grid using the graph extraction method described earlier along with graph-based search (in this example, A* search was used).

As long as the heuristic used by A* is *admissible*, that is, $h(s) \leq c^*(s, s_{goal})$ for all states s, the solution returned by A* is guaranteed to be optimal [Gelperin, 1977]. To see this, imagine some other lower-cost path existed. Then, since we know state s_{start} is on this path, we know at least one of the states on this path has an optimal path cost when A* terminates. Consider the state furthest along this path from s_{start} whose path cost is optimal when A* terminates. Call this state s_l. Either s_l is on *OPEN* when A* terminates or it is not. If it is, then $f(s_l) = g(s_l) + h(s_l) \leq g(s_l) + c^*(s_l, s_{goal}) < g(s_{goal}) + h(s_{goal}) = f(s_{goal})$, so s_l would be expanded instead of s_{goal}. This is a contradiction. If s_l is not on *OPEN* when A* terminates, then it has already been expanded, in which case its successor along the lower-cost path would have had its g-value updated to reflect its optimal path through s_l. But this is also a contradiction since s_l is the last along the path with an optimal value. Hence no such lower-cost path exists.

By using a heuristic that provides an accurate estimate of the cost of a least-cost path from each state to the goal state, A* can expand significantly fewer states than Dijkstra's algorithm. In particular, if the heuristic is perfect, so that $h(s) = c^*(s, s_{goal})$ for all states s, then it is possible for it to expand *only* the states along one least-cost path from s_{start} to



(a) (b)

Figure 2.6: (a) A path planned from the lower-left state (in blue) to the top-right state (in yellow) in the graph from Figure 2.1. (b) The path overlaid on the original grid from Figure 2.1.

s_{goal}[1]. In fact, it can also be shown that, for a given heuristic function h, A* expands the minimum number of states possible in guaranteeing an optimal solution [Gelperin, 1977]. A* can also search in a backwards fashion, beginning at the goal and following the predecessors of states rather than the successors. This modified version is known as Backwards A* and will be useful to us when discussing algorithms that are able to repair solutions in Chapter 3.

The use of a heuristic function by A* causes the algorithm to expand more promising states first and thus potentially saves a significant amount of computation. Using a heuristic function alone to prioritize states for expansion (i.e. ordering states s in *OPEN* based only on their $h(s)$ values) results in best-first search, which is a greedy algorithm that can sometimes vastly reduce computation times compared to Dijkstra's algorithm or A*. However, path optimality is no longer guaranteed.

Bounds on the optimality of the path can be achieved by using an admissible heuristic function and an *inflation factor* that overemphasizes the value returned by the heuristic function. Specifically, if the value used for inserting a state s into *OPEN* is the sum of its path cost plus a heuristic estimate of its cost to the goal state multiplied by an inflation factor ϵ, then the cost of the path returned by the search will be within ϵ times the cost of an optimal path. In other words, if states s are expanded in increasing order of $g(s)+\epsilon \cdot h(s)$, it can be shown that $g(s_{goal}) \leq \epsilon \cdot g^*(s_{goal})$. The resulting algorithm is known as either Weighted A* or Inflated A* and has been used to produce a number of efficient approximation algorithms, many of which we will discuss in later chapters.

Combining these graph searches with efficient discretizations of the planning problem has been a popular approach in mobile robotics. These discrete approaches have a number of nice properties, such as solution optimality or bounds on the suboptimality of the solutions

[1]This depends upon the tie-breaking criteria used.

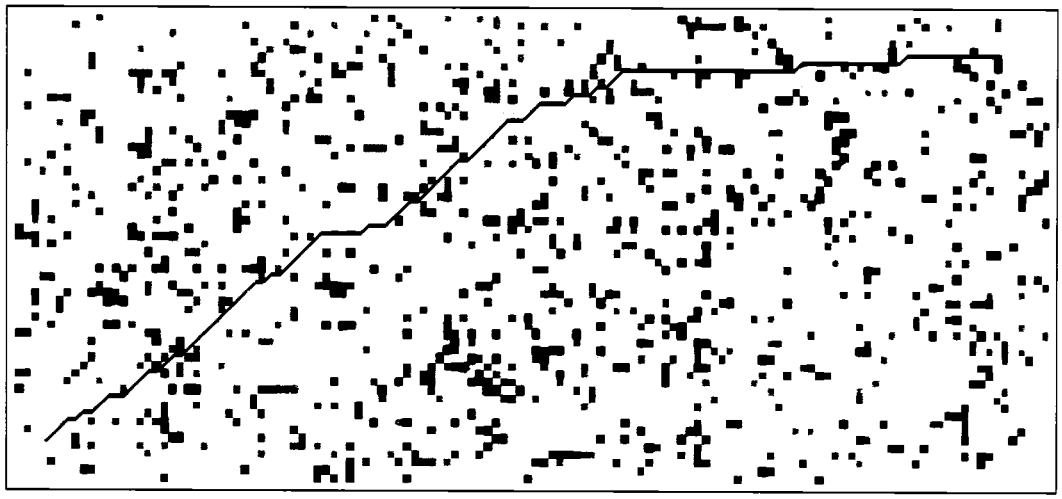


Figure 2.7: A path planned from the lower-left to the top-right of a non-uniform cost grid.

returned on the graph. They also proceed in deterministic, predictable manners, which is not true of some of the algorithms we will discuss next. However, because they create and plan over explicit discretizations of the planning problem, they are limited to problems for which this discretization and planning is feasible. In particular, when dealing with a very complex single agent with several degrees of freedom, or a team of agents, the dimensionality of the planning problem (i.e. the number of independent coordinates required to describe each state in the problem space) may be so large that an explicit discretization of this space is intractable. Consequently, discrete algorithms are widely used for reasonably complex single agent path planning, but fall out of favor when the number of agents we wish to plan for, or the complexity of the problem, increases significantly.

2.2 Sampling-based Algorithms

A popular alternative to discrete planning is to plan in the original problem space and use sampling to efficiently explore this space for a solution path. The basic idea behind sampling-based algorithms is to generate a set of sample configurations and to try and connect these configurations to each other through sequences of actions. They avoid explicitly constructing a discrete representation of the entire configuration space and can thus be much better than discrete approaches at coping with high-dimensional search spaces.

For example, consider our constrained exploration task, in which a team of robots explores an environment while maintaining line-of-sight connectivity across the team. Planning a path for the team in such a scenario can be formulated as a joint planning problem where the number of states in the joint configuration space is exponential in the number

of robots. Even for simple instantiations of this problem involving small teams, the corresponding state space can be extremely large. For example, imagine three robots performing constrained exploration in a 100×100 meter environment, where the position of each robot is specified in one meter resolution and their headings are ignored. The dimensionality of this planning problem is 6 (to specify a particular state, we need to describe the $\{x, y\}$ position for each of the three robots) and the number of possible states is $(100 \times 100)^3 = 1 \times 10^{12}$. Further, even if we constrain the actions available to each agent to be moves into the 8 adjacent $\{x, y\}$ positions (as in our 8-connected grid planning presented earlier), the number of joint actions from any particular state, or *branching factor*, is $(9^3 - 1) = 729$ (some of the robots can stay in the same position). The size of this state space and branching factor is prohibitively large for most discrete algorithms. However, sampling-based algorithms are quite effective when dealing with high-dimensional planning problems such as this one, since they are not nearly as encumbered by the large state spaces and branching factors.

A number of sampling-based algorithms exist. The Ariadne's Clew algorithm grows a search tree out from an initial configuration by iteratively selecting random nodes in the search tree and then growing outwards from these nodes to new configurations that are as far away as possible from the nodes currently in the search tree [Mazer et al., 1992, Mazer et al., 1998]. The Expansive Space Trees algorithm operates similarly to Ariadne's Clew, except that nodes are selected from the tree with probabilities inversely proportional to the number of other tree nodes that reside within some fixed neighborhood [Hsu et al., 1999]. This approach uses information regarding the current nodes in the search tree to bias the growth of the tree. More recent algorithms extend this idea by using information about the configuration space itself to direct the tree's growth. Probabilistic Roadmaps create a compact graph representation of the entire configuration space by sampling the space itself and connecting nearby samples together into graphs [Kavraki et al., 1996]. Rapidly-exploring Random Trees grow a search tree from an initial configuration by iteratively sampling the configuration space and then selecting the node in the tree closest to this sample and extending the tree outwards from this node [LaValle, 1998, LaValle and Kuffner, 1999, Kuffner and LaValle, 2000, LaValle and Kuffner, 2001]. Several similar approaches and variants on these approaches have also been developed.

A major benefit of all of these approaches is that they do not need a discrete state space in which to operate: they can plan within the continuous configuration space. Moreover, they do not even need to explicitly extract the configuration space, as long as a collision detection routine is available that can return whether a collision exists when transitioning between two particular configurations. It is this latter property that makes them well-suited to solving very high-dimensional problems.

These algorithms do not guarantee optimal solutions, and the quality of the solutions generated can often be quite poor. However, most of them are complete provided that

```
InitializeRRT(rrt T)
 1  T.add(q_start);
GrowRRT(rrt T)
 2  q_new = q_start;
 3  while (Distance(q_new, q_goal) > distance-threshold)
 4    q_target = ChooseTarget();
 5    q_nearest = NearestNeighbor(q_target, T);
 6    q_new = Extend(q_nearest, q_target, T);
 7    if (q_new ≠null)
 8      T.add(q_new)
ChooseTarget()
 9    p = RandomReal([0.0, 1.0]);
10    if (p < goal-sampling-prob)
11      return q_goal;
12    else
13      return RandomConfiguration();
Main()
14  InitializeRRT(tree);
15  GrowRRT(tree);
```

Figure 2.8: The Rapidly-exploring Random Tree Algorithm. The modifications required for goal-biasing are shown highlighted in red.

they are allowed to run for an infinite amount of time. Out of all these sampling-based algorithms, the two most common in robotic path planning are Rapidly-exploring Random Trees (RRTs) and Probabilistic Roadmaps (PRMs). RRTs are used for planning a path between a single start and goal pair, while PRMs are most commonly used to efficiently generate paths between any two locations. We now describe each of these algorithms in more detail.

Rapidly-exploring Random Trees

RRTs grow a search tree out from the initial state or configuration through the search space towards the goal configuration. They do this by generating random sample configurations in the space and attempting to connect the current search tree to these sample configurations. The basic algorithm is presented in Figure 2.8.

To begin with, the search tree is initialized to contain only the initial configuration q_{start}. Next, the tree is grown until the goal configuration has been added to the tree[2].

[2]Typically, the algorithm will terminate when the distance between the goal and the closest node in the tree is less than some threshold distance, as shown in line 3

To grow the tree, (lines 2 to 8 in function *GrowRRT*), first a target configuration q_{target} is randomly selected from the configuration space using the function *ChooseTarget*. Then, a *NearestNeighbor* function selects the node $q_{nearest}$ in the tree closest to q_{target}. Finally, a new node q_{new} is created in an *Extend* function by growing the tree some distance from $q_{nearest}$ towards q_{target}. If extending the tree towards q_{target} requires growing through an obstacle, no extension occurs. Otherwise, q_{new} is added as a node to the tree and the extension from $q_{nearest}$ to q_{new} is added as a branch (both of these additions take place when q_{new} is added on line 8). When a new node q_{new} is added to the tree, the node $q_{nearest}$ that was used to extend the tree to q_{new} is known as its parent. This entire sampling, selection, and extension process is repeated until the tree connects to the goal, at which point a path can be retraced from the goal back to the initial configuration by starting at the node closest to the goal and then repeatedly following the parent branch of each node encountered.

A very nice property that follows from this sampling-based method of construction is that the tree growth is strongly biased towards unexplored areas of the configuration space. Consequently, exploration occurs very quickly. To generate solutions even more efficiently, the original RRT algorithm has also been modified to bias the search towards the goal configuration [LaValle and Kuffner, 2001]. To do this, instead of always selecting a random sample configuration to grow towards, with some probability the goal configuration q_{goal} is chosen as the sample configuration. This has the effect of focusing the tree in the direction of the goal, much as the use of a heuristic does for A* search. This modification is also included in Figure 2.8, as the lines shaded in red (lines 9 to 12).

Figure 2.9 shows an example application of RRTs to path planning for a single agent navigating an environment containing several obstacles. The initial location of the agent is shown as the small blue robot on the left side of the environment, and the goal is the blue square on the right. We have included four different images showing the state of the tree at various stages during its growth. In this example, the goal bias threshold (goal-sampling-prob on line 9 was set to 0.05 so that the goal was chosen as the next sample point 5% of the time).

The RRT algorithm has been widely-used in robotics. For example, it has been used to control vehicles operating in indoor environments [Oriolo et al., 2004], vehicles operating in rugged, outdoor environments [Urmson and Simmons, 2003, Kobilarov and Sukhatme, 2004], robocup teams [Bruce and Veloso, 2002], aerial vehicles [Kim and Ostrowski, 2003], humanoids [Kuffner et al., 2003], and in a host of other applications [Branicky and Curtiss, 2002, Cheng and LaValle, 2002, Choudhury and Lynch, 2002, Frazzoli et al., 2002, Kagami et al., 2003, Kallmann et al., 2003, Yakey et al., 2001]. RRTs have also been extended in various ways. In particular, bi-directional and multi-directional variants now exist, which maintain two or more trees that are simultaneously grown and eventually connected together [Kuffner and LaValle, 2000, Bekris et al., 2003, Strandberg, 2004a, Strandberg, 2004b].

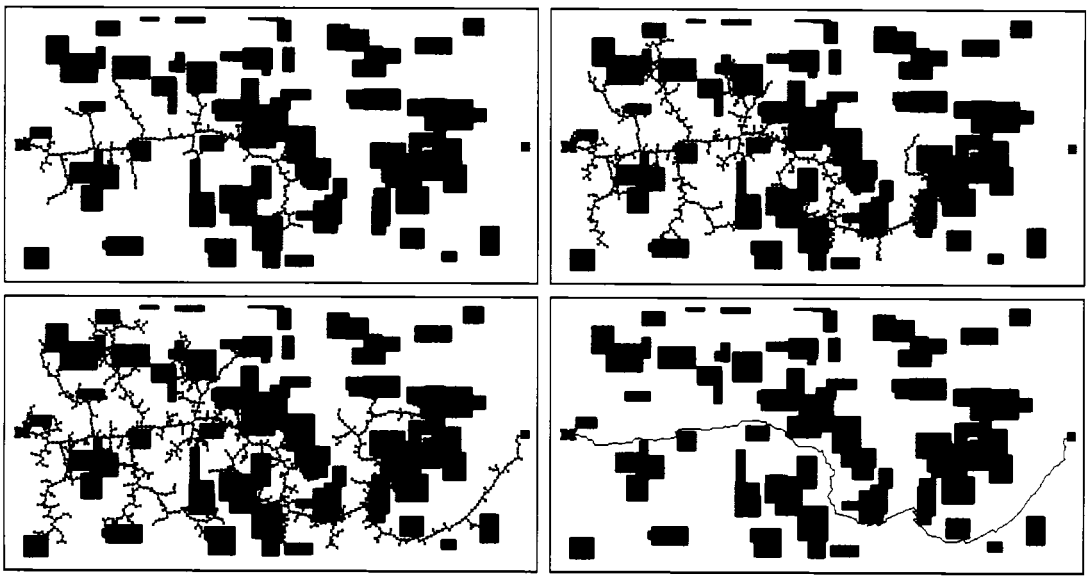


Figure 2.9: An RRT created for planning a path from one end to the other of a binary environment. Obstacles are shown in black. The agent begins at the location represented by the blue vehicle (on the left side of the environment) and the goal location is shown as a large blue square (on the right side of the environment). Nodes of the tree are shown as small blue squares. The top two images show the RRT at two stages during its growth. The bottom two images show the final tree and path constructed, respectively.

Also, researchers have looked at improving the efficiency of the growth of the tree in difficult configuration spaces [Yershova et al., 2005, Jaillet et al., 2005] and using information learned during the search to improve its future performance [Burns and Brock, 2005], as well as improving the quality of the solutions generated by RRTs [Urmson and Simmons, 2003] and using the current tree to help grow a new one when the configuration space changes [Bruce and Veloso, 2002]. We will discuss some of these extensions in depth in later chapters.

Probabilistic Roadmaps

While the RRT algorithm grows a search tree to produce a solution path from a single initial configuration to a single goal configuration, Probabilistic Roadmaps (PRMs) construct a graph representation of the entire configuration space that is dense enough that it can be used to plan paths between *any* two configurations. They are thus well suited to problems involving more than one start-goal pair.

The PRM algorithm is usually split into two distinct phases: a preprocessing phase and a query phase. The preprocessing phase is where the roadmap is built. The basic roadmap construction algorithm is shown in Figure 2.10 (adapted from [LaValle, 2006]), where G is the graph being created, C_{free} specifies the non-obstacle portion of the configuration space,

```
Main()
 1  G = ∅;
 2  size = 0;
 3  while (size < min-size)
 4    q_new = RandomConfiguration();
 5    if (q_new ∈ C_free)
 6      G.addVertex(q_new);
 7      size = size + 1;
 8      for each q_nbr ∈ Neighbors(q_new, G)
 9        if (not (G.sameComponent(q_new, q_nbr)) and Connect(q_new, q_nbr))
10          G.addEdge(q_new, q_nbr);
```

Figure 2.10: The Probabilistic Roadmap Construction Algorithm

Neighbors(q, G) returns a set of nodes in G close to q, G.sameComponent(q_{new}, q) returns whether a path in the graph already exists between q_{new} and q_{nbr}, and *Connect*(q_{new}, q_{nbr}) returns whether it is possible to plan a local path through the configuration space from q_{new} to q_{nbr}. Both the *Neighbors* and *Connect* functions can be implemented in several different ways, leading to various construction techniques.

The construction algorithm begins by initializing the graph to be empty. It then expands the graph by selecting random samples in the configuration space, adding these sample points to the graph, and then attempting to connect these samples to other nodes in the graph. To do this, for a given sample q_{new} it finds the closest nodes *Neighbors*(q_{new}, G) in the graph, then tries to generate edges from q_{new} to each of these neighboring nodes q_{nbr}. If the two nodes are already connected, i.e. some path already exists in the graph between q_{new} and q_{nbr}, then no additional edge will be added. However, if such a path does not already exist and a new edge has been successfully generated, the edge will be added to the graph. As mentioned in the previous paragraph, local planners are used to generate these new edges and the efficacy of these planners can have a huge impact on the efficiency of the overall algorithm and the quality of the resulting roadmap. This process is continued until the roadmap contains a minimum number of nodes (denoted min-size in our pseudocode).

Once the PRM has been constructed, the query phase plans a path between a specified q_{start} and q_{goal} pair. First, the two configurations are connected to the roadmap using the same process as for new sample nodes in the construction phase (lines 6 - 10 in Figure 2.10). If this is successful, then a path is planned between the two vertices in G using any graph-based planning algorithm (such as those described in Section 2.1). As with RRTs, several modifications and extensions have been made to PRMs for various applications and purposes [Amato et al., 1998, Bohlin and Kavraki, 2000, Leven and Hutchinson, 2002, Pisula et al., 2000, Simeon et al., 2000, Wilmarth et al., 1999, Yakey et al., 2001]. For instance,

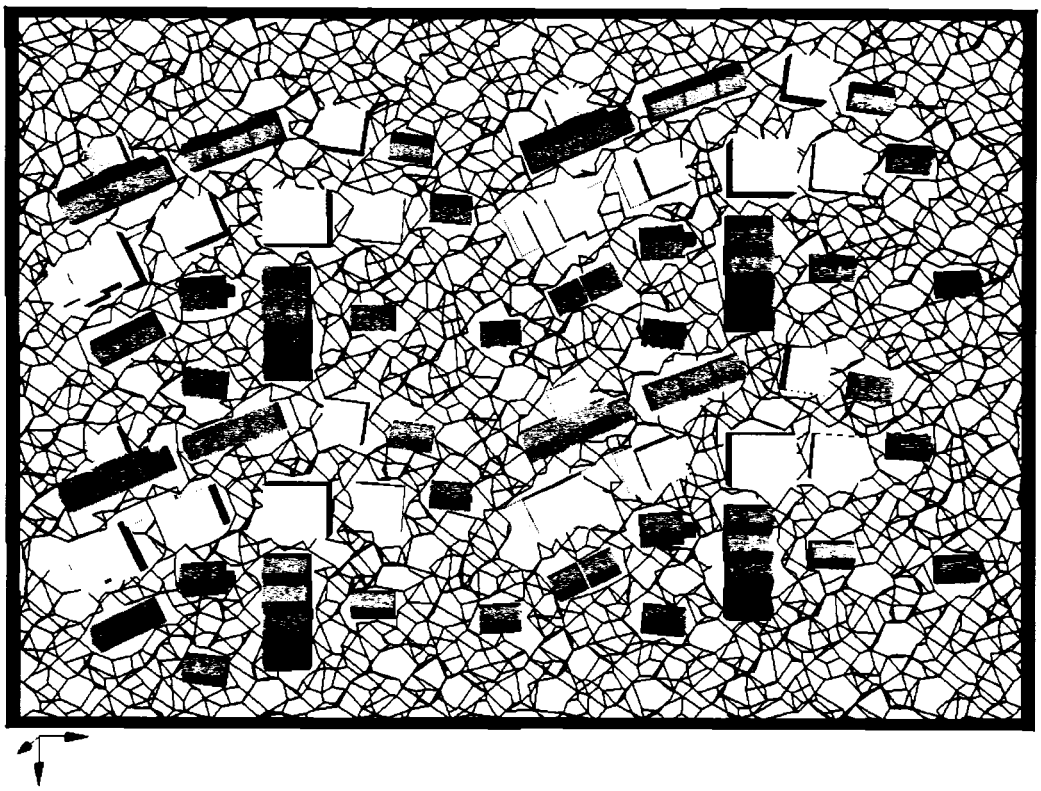


Figure 2.11: An example Probabilistic Roadmap created for a ground vehicle navigating through an outdoor environment. The roadmap has been strengthened to allow for multiple paths between locations. Figure courtesy of Jur van den Berg.

lazy versions have been developed that create the PRM without checking the validity of the edges in the roadmap, then do this checking during the query phase for just those edges that could contribute to a solution path [Bohlin and Kavraki, 2000, Sanchez and Latombe, 2001]. Modifications have also been made to the sampling process to focus computation on areas of the configuration space that may be difficult to connect, such as narrow passages [Kavraki et al., 1996, Amato and Wu, 1996, Amato et al., 1998, Boor et al., 1999, Hsu et al., 2003, Holleman and Kavraki, 2000]. See [LaValle, 2006] for a very good overview of these approaches.

An example PRM is illustrated in Figure 2.11. In this figure, the grey areas represent buildings taken from a three-dimensional model of an outdoor area. A PRM was constructed for a ground vehicle traveling through this environment and the edges of the PRM are shown in black. This PRM was strengthened to provide multiple alternative paths between any two locations using an algorithm from [Nieuwenhuisen and Overmars, 2004].

The beauty of PRMs is their ability to efficiently generate compact representations of

the configuration space. And, as with RRTs, all they require to create these representations is a collision checker that will return whether or not a particular edge is obstacle-free. In complex configuration spaces such as our constrained exploration domain, where explicitly constructing C_{free} is very difficult, PRMs present a very effective method for generating representations for planning. As discussed in the previous paragraph, these representations can then be searched using any of the discrete search algorithms we have discussed earlier. This combination of sampling-based representations and discrete planning algorithms can be extremely powerful and we will return to this idea in Chapter 5.

2.3 Other approaches

Thus far, we have focused on planning algorithms that provide paths from the initial location or configuration to a desired goal location or configuration. However, it is important to point out that this is not the only way to get an agent to its goal. In fact, several approaches exist that abandon the idea of path planning altogether, and instead reason purely locally to generate the next action for an agent. Many of these techniques use simple heuristics or behaviors to try and guide the agent towards the goal. For instance, the Bug algorithm and its variants direct the agent in a straight line towards the goal and when an obstacle is encountered, the agent follows the boundary of the obstacle until it can start moving towards the goal again [Lumelsky and Stepanov, 1987, Kamon et al., 1996]. Potential field methods treat the agent as a charged particle in a force field, where obstacles in the environment are assigned repulsive forces and the goal is assigned a large positive force [Barraquand et al., 1992, Khatib, 1986], and the agent travels along the gradient of this combined field. Several other related approaches also exist, such as vector field histograms [Borenstein and Koren, 1991], and motor schema-based navigation [Arkin, 1989]. All these algorithms are very computationally efficient, since no complex planning is performed and often no representation of the environment is stored or reasoned about. However, they often produce grossly suboptimal paths for the agent to traverse, particularly in cluttered environments or configuration spaces, and can lead the agent into dead-ends from which it cannot escape. For instance, U-shaped obstacles between an agent and the goal can often foil potential field methods, as the robot will travel towards the obstacle trying to get to the goal on the other side, and then become stuck behind the obstacle as it creates a local minima in the field. The other approaches described above suffer from similar problems. However, because of their efficiency they are often relied upon when there are not sufficient computational resources to perform more sophisticated planning.

In the remainder of this thesis, we focus on planning approaches that generate complete paths from the initial configuration of an agent or team of agents to the desired goal configuration. In the first part of the thesis, we concentrate on single agent planning prob-

lems involving fairly low-dimensional configuration spaces. For these problems, we rely on discrete approaches and present a series of algorithms that extend these techniques in a number of ways. First, we describe replanning extensions to these algorithms that are able to repair their solutions when new information is received concerning the environment or planning problem. We also provide techniques for improving the quality of the solutions generated by the classic grid-based mobile robot navigation strategy. We apply these algorithms to the problem of path planning in imperfectly-known environments. Next, we describe an algorithm that is able to plan and replan in an anytime fashion, so that solutions can be generated under strict time constraints and improved while deliberation time allows, then repaired when new information is received. We apply this algorithm to three different problems: smooth trajectory planning for a mobile robot moving at speed through a partially-known environment, trajectory planning for a kinematic arm moving in a changing environment, and mobile robot navigation in an environment containing dynamic obstacles.

In the second part of this thesis we discuss multi-agent path planning and return to sampling-based algorithms. In particular, we will apply the RRT algorithm to our constrained exploration problem, where the dimensionality and complexity of the configuration space precludes the discrete approaches used for single agent planning. We will also present a number of extensions to this algorithm that are analogous to those made to discrete algorithms. In particular, we will present RRT-based algorithms able to repair solutions when new information is received, improve the quality of the solutions returned over time, and both improve and repair solutions when time and the quality of the available information are limited. We will apply these algorithms to various instances of the constrained exploration problem, including the cases of imperfect initial information concerning the environment, areas of non-uniform traversal cost existing within the environment, and dynamic obstacles moving about the environment.

Chapter 3

Single Agent Planning with Imperfect Information

The approaches mentioned in the previous chapter work well for planning an initial path through a known environment. However, when operating in real environments, a mobile robot usually does not have complete map information. There are a number of different ways the information the robot has can be limited:

1. The agent has no map information or a partial map
2. The agent has an incorrect map
3. The agent has uncertain information regarding the environment

An example of case (3) would be an agent that starts out with a low-resolution map (such as that generated by a satellite), so that it has some information about each area of the environment, but this information provides only a general indication of what the true nature of the environment is at the area.

It turns out that case (1) can be made equivalent to case (2) from a planning point of view simply by filling any gaps in the agent's map with a default value (such as "traversable" or "untraversable"). And both case (1) and case (2) break down to case (3) if the agent realizes its initial information is imperfect.

There are two different general strategies for dealing with imperfect initial information. The first strategy is to take the initial information and use it to come up with a 'best guess' of what the environment is like at each location, then use this estimate to plan an initial path for the agent. As the agent executes this plan and observes new information concerning the environment it can update its path to reflect this new information. The second strategy is to reason about the uncertainty associated with the initial information and create a plan that takes into account this uncertainty. The first of these strategies is very computationally

efficient and works well when the connectivity of the environment is high (i.e., when errors in the agent's initial estimate of the environment do not have significant consequences); the second is much more computationally expensive but can produce shorter, more robust paths in certain environments.

In this chapter, we introduce algorithms for mobile robot navigation that fall under each of these two general strategies, along with some of their limitations. We begin by describing current approaches that fall under the former class, and introduce two new algorithms that are more efficient and produce better solutions than these existing approaches, respectively. We then describe approaches that fall under the latter class, and provide a new approach for efficiently reasoning about key areas of uncertainty in the environment. We conclude the chapter with a discussion of the new algorithms introduced and their applicability in various domains.

3.1 Assumptive Planning

Given an initial map that is partially-complete or otherwise imperfect, a common approach in robotics is to estimate, or 'assume', the nature of the environment at each location, then use this estimate to plan a path for an agent using the approaches described in the previous chapter. This is known as *assumptive planning* [Nourbakhsh and Genesereth, 1996]. As the agent navigates through the environment, its path may turn out to be invalid or suboptimal as it receives updated environmental information through, for example, an onboard sensor. It is thus important that the agent is able to update its map and replan new paths when new information arrives.

Current Approaches

Assumptive planning consists of three stages: constructing an estimate of the environment from initial information, planning an initial path for the agent, and repairing this path when new information is received during the agent's traverse.

Coming up with an initial estimate of the environment is fairly straightforward. Typically, the initial information is used to produce a most likely hypothesis of the terrain at each location, and unknown areas are usually assumed to be traversable. Depending on the mission, more conservative estimates can be produced, where all uncertain areas are marked as untraversable or as high cost areas so that the agent will avoid them.

Once this estimate has been produced, the approaches described in Chapter 2 can be used to plan an initial path for the agent. In particular, the discrete grid-based approaches discussed in Section 2.1 are very popular for mobile robot navigation, where the planning problem is transformed into a search over a traversability grid, and is then simplified further

to a search over a discrete graph extracted from this grid. However, since the resulting paths are planned based on the initial, imperfect estimate of the environment, they will need to be updated when new information arrives that is in conflict with this estimate.

Many planning algorithms exist for repairing solutions when new information is received that invalidates these solutions or reduces their effectiveness [Stentz, 1994, Stentz, 1995, Barbehenn and Hutchinson, 1995, Ramalingam and Reps, 1996, Trovato, 1990, Ersson and Hu, 2001, Huiming et al., 2001, Podsedkowski et al., 2001, Koenig and Likhachev, 2002]. Focussed Dynamic A* (D*) [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2002] are currently the most widely-used of these algorithms, due to their efficient use of heuristics and incremental updates. These graph-based algorithms operate similarly to A* search during initial planning, but if the costs associated with some of the edges later change, these algorithms are able to efficiently update the solution to account for these changes and maintain its optimality. D* has been shown to be one to two orders of magnitude more efficient than planning from scratch with A*, and it has been incorporated into a plethora of robotic systems [Stentz and Hebert, 1995, Hebert et al., 1999, Matthies et al., 2000, Thayer et al., 2000, Zlot et al., 2002]. D* Lite is a simplified version of D* that is slightly more efficient by some measures [Koenig and Likhachev, 2002]. It has been used to guide Segbots and ATRV vehicles in urban terrain [Likhachev, 2003]. Both algorithms are graph-based and optimal. Because these algorithms are so fundamental to robot navigation, we describe them in detail below.

Replanning: D* and D* Lite

D* and D* Lite maintain least-cost paths between a start state and a goal state as the costs of edges between states change. Both algorithms can handle increasing or decreasing edge costs and moving start states. Since the two algorithms are fundamentally very similar, we restrict our attention here to D* Lite for simplicity.

D* Lite maintains a least-cost path from a start state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is the set of states in some finite state space (as in Chapter 2). Instead of searching for this path from the start state towards the goal (as in A*), its search proceeds in a backwards fashion from the goal (as in Backwards A*)[1]. This is to facilitate efficient replanning and to allow for the start state s_{start} to change as the agent moves. We will describe how this works and why it is important in due course.

To perform this search, it stores an estimate $g(s)$ of the cost of the best path from each state s to s_{goal}, similarly to Backwards A*. It also stores an additional value, $v(s)$, that is equal to the cost of the best path from s the last time it was expanded[2].

[1] A forwards-searching version also exists, known as Lifelong Planning A* [Koenig et al., 2004], for when the agent is not moving and edge cost changes are not localized around the agent.

[2] In the original presentation of D* Lite, this $v(s)$ value was represented by $g(s)$ and the $g(s)$ value

Both the v-values and the g-values of each state are initialized to infinity, except for $g(s_{goal})$. The v-value of a state is only updated when the state is expanded. The g-value of a state is updated whenever one of its successors is expanded, and always satisfies the following relationship:

$$g(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in Succ(s)}(c(s,s') + v(s')) & \text{otherwise,} \end{cases}$$

where $Succ(s)$ denotes the set of successors of s and $c(s,s')$ is the cost of the edge from s to s', exactly as in Chapter 2. A state is called consistent iff its v-value equals its g-value, otherwise it is either overconsistent (if $v(s) > g(s)$) or underconsistent (if $v(s) < g(s)$).

Like A*, D* Lite uses a heuristic and a priority queue *OPEN* to focus its search and to order its cost updates efficiently. The heuristic $h(s,s')$ estimates the cost of moving from state s to s', and needs to be consistent [Pearl, 1984], that is, satisfy $h(s_{start}, s_{start}) = 0$ and $h(s_{start}, s) \leq h(s_{start}, s') + h(s', s)$, for all states $s \in S$ and $s' \in Pred(s)$ with $s \neq s_{start}$. The priority queue always holds exactly the inconsistent states; these are the states that need to be updated and made consistent.

The priority $k(s)$ of a state s in the queue is

$$\begin{aligned} k(s) &= [k_1(s), k_2(s)] \\ &= [min(v(s), g(s)) + h(s_{start}, s), min(v(s), g(s))]. \end{aligned}$$

The first component of this priority value, $k_1(s)$, is the f-value used in Backwards A*, while the second, $k_2(s)$, is the g-value. A lexicographic ordering is used on the priorities, so that priority $k(s)$ is less than priority $k(s')$, denoted $k(s) < k(s')$, iff $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) < k_2(s')$. D* Lite expands states from the queue in increasing priority, updating their v-values and their predecessors' g-values, until there is no state in the queue with a priority less than that of the start state. During its generation of an initial solution, D* Lite thus performs similarly to a Backwards A* search that breaks ties between two states in favor of the one with the smallest g-value.

One version of the algorithm is given in Figures 3.1 and 3.2. For simplicity, this version assumes the agent does not move. However, in the original presentation of the algorithm this capability was included and involved an additional restriction on the heuristic function and the addition of a bias function that removed the need to reorder *OPEN* every time the start state changed, an idea borrowed from [Stentz, 1995].

The algorithm begins by initializing the g-value of the goal state to be zero, then inserts this state into *OPEN* to be expanded. It then proceeds to perform a Backwards A* search: each time a state s is expanded, $v(s)$ is set to $g(s)$ and the predecessors of s have their

presented here was referred to as $rhs(s)$. We follow the more recent notation used by the developers of D* Lite [Likhachev, 2005], which is more consistent with the notation used in A* search.

UpdateSetMembership(s)

1 if ($v(s) \neq g(s)$)
2 insert/update s in $OPEN$ with key(s);
3 else
4 if ($s \in OPEN$) remove s from $OPEN$;

ComputePath()

5 while(key(s_{start}) > $\min_{s \in OPEN}$(key(s)) OR $v(s_{start}) < g(s_{start})$)
6 remove s with the smallest key(s) from $OPEN$;
7 if ($v(s) > g(s)$)
8 $v(s) = g(s)$;
9 for each predecessor s' of s
10 if s' was never visited before
11 $v(s') = g(s') = \infty$; $bp(s')$ = **null**;
12 if ($g(s') > c(s', s) + v(s)$)
13 $bp(s') = s$;
14 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembership($s'$);
15 else
16 $v(s) = \infty$; UpdateSetMembership(s);
17 for each predecessor s' of s
18 if s' was never visited before
19 $v(s') = g(s') = \infty$; $bp(s')$ = **null**;
20 if ($bp(s') = s$)
21 $bp(s')$ = $\text{argmin}_{s'' \in Succ(s')} c(s', s'') + v(s'')$;
22 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembership($s'$);

Figure 3.1: D* Lite: ComputePath function.

g-values updated to reflect this new value of $v(s)$. Here, $\text{argmin}_{s' \in Succ(s)} f()$ returns the successor of s for which function f is minimized. Once an initial path has been computed, the algorithm waits for changes to be made to the cost of edges in the graph (Figure 3.2, line 10). When an edge cost is changed, the g-value of any directly affected state is updated and, if this makes the state inconsistent, the state is inserted into $OPEN$. If the state was already on $OPEN$, its position is updated to reflect its new priority value. When all the directly-affected states have been updated, *ComputePath* is called to regenerate a new path.

During this call to *ComputePath*, it is possible that it may expand an overconsistent state s (Figure 3.1, lines 15 to 22), since edge costs may have increased. When this happens, it sets $v(s) = \infty$, which effectively erases the previous path cost of the state and allows it to compute a new one. This step also forces the state to become either consistent (if $g(s) = \infty$) or overconsistent (if $g(s) < \infty$). The predecessors of the state s are then checked to see if any of them currently use s as their best successor. To determine this, each state s stores a back-pointer, $bp(s)$, that specifies which successor it used to compute its current

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + h(s_start, s); g(s)];
 3 else
 4   return [v(s) + h(s_start, s); v(s)];

Main()
 5 g(s_start) = v(s_start) = ∞; v(s_goal) = ∞; bp(s_goal) = bp(s_start) = null;
 6 g(s_goal) = 0; OPEN = ∅;
 7 insert s_goal into OPEN with key(s_goal);
 8 forever
 9   ComputePath();
10   wait for changes in edge costs;
11   for all directed edges (u, v) with changed edge costs
12     update the edge cost c(u, v);
13     if (u ≠ s_goal)
14       if u was never visited before
15         v(u) = g(u) = ∞; bp(u) = null;
16       bp(u) = argmin_{s' ∈ Succ(u)} c(u, s') + v(s');
17       g(u) = c(u, bp(u)) + v(bp(u)); UpdateSetMembership(u);
```

Figure 3.2: D* Lite: Main function

g-value. If a predecessor s' of s uses s as its back-pointer, i.e. $bp(s') = s$, then $g(s')$ is updated to reflect the new value of $v(s)$. Any states that are made inconsistent as a result of this update are added to *OPEN* to propagate their new values to their predecessors, and so on.

D* Lite continues to expand the states in *OPEN* in order of increasing priority until it contains no state with a priority less than that of the start state. This termination condition guarantees that an optimal path will have been found from the start state to the goal state when processing is finished. Once this condition is met and the *ComputePath* function is completed, an optimal path from s_{start} to s_{goal} can be traced by starting at s_{start} and always transitioning from a state s to a successor s' that minimizes $c(s, s') + v(s')$, until s_{goal} is reached.

For mobile robot navigation, it is important that the algorithm searches in a backwards fashion from s_{goal} to s_{start} for two reasons. First, when the algorithm is being used to guide an agent across an environment, the current position s_{start} of the agent is constantly changing. This means that if the path needs to be updated to reflect new information, the new s_{start} state is not the same as the old s_{start} state. If all the path costs in the graph are stored relative to s_{start} then previous costs are no longer of any use when s_{start} changes. Since the goal s_{goal} does not change when the agent moves, it makes sense to use this as

the source of the search so that previously-computed path costs will still be useful in later replanning phases. Secondly, in mobile robot navigation it is usually the case that the new information is being received through an onboard sensor, and thus concerns the local vicinity of the agent. It is far more efficient to have edge costs changing out near states at the end of the search rather than near the source of the search, as fewer states have their paths affected by these changes. If the source of the search is the goal location s_{goal} rather than the state of the agent s_{start}, then much less replanning is required to update the path when new information is received.

The D* Lite algorithm is efficient because it uses a heuristic to restrict attention to only those states that could possibly be relevant to repairing the current solution path from a given start state to the goal state. When edge costs decrease, the heuristic ensures that only those newly-overconsistent states that could potentially decrease the cost of the start state are processed. When edge costs increase, it ensures that only those newly-underconsistent states that could potentially invalidate the current cost of the start state are processed.

Limitations

The standard assumptive-based mobile robot navigation framework as described above has a number of limitations. Firstly, as described in Section 2.1, it is common to use grid-based representations of the environment, from which a discrete graph is extracted and used for planning. However, when applied to such graphs, even 'optimal' search algorithms such as A* and D* Lite produce jagged, suboptimal paths that are difficult for our agents to traverse. This is because the successor function defined over the graph (shown in Figure 2.2) only allows transitions between adjacent grid cell centers or corners, which limits the paths produced to having heading increments of 45 degrees and results in awkward, costly paths for our agents to traverse. We would like to improve the quality of these returned paths so that they are more direct and less costly.

Secondly, the use of uniform grid-based representations can be extremely taxing in terms of both memory and computational requirements when operating over large areas. Non-uniform representations can offer much better memory performance, but their use has been limited in robotics because of the quality of the resulting paths produced. We would also like to improve the quality of the paths produced using these representations so that our agents can navigate effectively over larger outdoor environments.

Finally, using a replanning algorithm such as D* Lite to generate paths over graphs is far more efficient than planning from scratch every time the graph is updated. However, even these algorithms can process far more states than is necessary when edge costs increase. This is due to the fact that any state whose priority value is less than the start state will be expanded, even if the state does not reside on the current solution path and its path cost

has increased. We would like to improve the efficiency of D* Lite by ignoring such states, yet in a way that does not sacrifice its optimality.

In the following three sections we introduce algorithms that attempt to overcome these limitations. We begin by introducing *Delayed D**, a graph-based replanning algorithm that produces optimal paths, as with D* and D* Lite, but tries to be much more restrictive with how much computation it performs. Next, we describe *Field D**, an interpolation-based planning and replanning algorithm designed to produce less-costly paths through uniform grid representations. We then present an extension of this algorithm, *Multi-resolution Field D**, that is able to plan over non-uniform grid representations, resulting in significant savings in memory and, often, computation.

3.2 Delayed D*

As discussed in Section 3.1, D* Lite is efficient because it uses a heuristic to restrict attention to only those states that could possibly be relevant to repairing the current solution path from a given start state to the goal state. When edge costs decrease, the heuristic ensures that only those newly-overconsistent states that could potentially decrease the cost of the start state are processed. When edge costs increase, it ensures that only those newly-underconsistent states that could potentially invalidate the current cost of the start state are processed.

However, perhaps we can do better by being even more restrictive. It is possible that by using the above approach, we may expand many more states than is necessary. This is because even if an inconsistent state has a priority less than the start state, its inconsistency may not affect the optimality of the current solution path. Figure 3.3 illustrates such a scenario. It would be ideal if we could tell whether the inconsistency of a particular state is of consequence to the validity of the current solution without actually propagating this inconsistency.

When we encounter a series of cost decreases that affect states with lower priorities than the start state, there is no simple check that will provide us with this information. When we encounter a similar series of cost increases, however, there *is* such a check: we can guarantee the optimality of our current solution if none of these increases takes place along the solution path. This qualitative difference suggests different treatment for states made underconsistent (due to edge cost increases) and states made overconsistent (due to edge cost decreases).

Motivated by this finding, we have developed *Delayed D**, a replanning algorithm that processes underconsistent states much more selectively than overconsistent states [Ferguson and Stentz, 2005b]. Delayed D* is a modified version of D* Lite that delays the propagation of cost increases as long as possible. When cost changes occur, the g-values of all affected

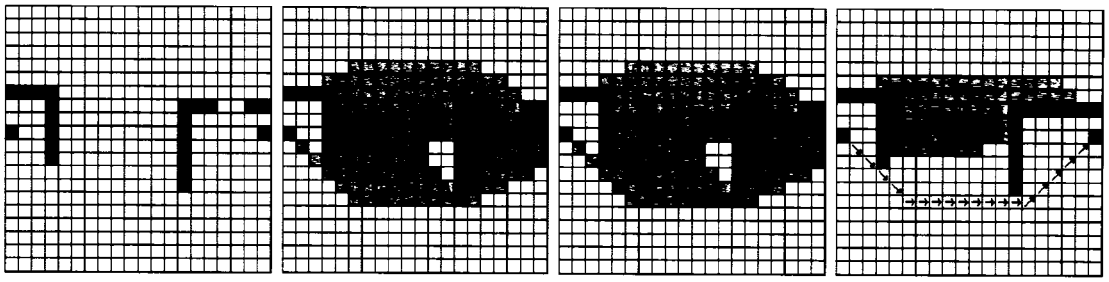


Figure 3.3: An eight-connected grid replanning scenario in which a couple of inconsistent states cause D* Lite to make significant repairs. A least-cost path is maintained between state s_{start} and state s_{goal}, the two dark blue cells on the left and right sides of each grid, respectively. The center-left grid shows the cells expanded by D* Lite (shaded light-gray) during initial planning. The center-right grid shows the cells along the initial solution path (dark blue arrows). The right grid shows the states re-expanded by D* Lite (shaded light-gray) after information reveals that the gap in the right wall is blocked. Several states become underconsistent and have their costs updated by D* Lite. However, the original path from s_{start} to s_{goal} remains valid. Since none of these underconsistent states resided upon the path from s_{start} to s_{goal}, they could have been ignored without jeopardizing optimality.

states are updated and the overconsistent states are processed immediately, as in D* Lite. The underconsistent states are ignored. Then, when new values of the overconsistent states have been adequately propagated through the state space, the returned solution path is checked for any underconsistent states. All underconsistent states on the path are added to the priority queue and their updated values are propagated through the state space. Because the current propagation phase may alter the solution path, the new solution path needs to be checked for underconsistent states. The entire process repeats until a solution path that contains only consistent states is returned.

By processing underconsistent states in a lazy fashion, Delayed D* holds two advantages over D* Lite. Firstly, it ignores some underconsistent states entirely and thus reduces the overall computation required to generate a solution. Secondly, even when it has to process underconsistent states, if it has delayed this processing long enough then the effects of various underconsistent states may be incorporated at the same time, in a single propagation. So, rather than propagating cost increases through the state space every time a new underconsistent state turns up, it waits until some underconsistent state jeopardizes the optimality of the current solution, then does a single propagation of values that may deal with several underconsistent states at once.

UpdateSetMembership(s)

1 if ($v(s) \neq g(s)$)
2 insert/update s in *OPEN* with key(s);
3 else
4 if ($s \in$ *OPEN*) remove s from *OPEN*;

UpdateSetMembershipLower(s)

5 if ($v(s) > g(s)$)
6 insert/update s in *OPEN* with key(s);
7 else if ($v(s) = g(s)$)
8 if ($s \in$ *OPEN*) remove s from *OPEN*;

ComputePathDelayed()

9 while(key(s_{start}) > $\min_{s \in OPEN}$(key(s)) OR $v(s_{start}) < g(s_{start})$)
10 remove s with the smallest key(s) from *OPEN*;
11 if ($v(s) > g(s)$)
12 $v(s) = g(s)$;
13 for each predecessor s' of s
14 if s' was never visited before
15 $v(s') = g(s') = \infty$; $bp(s')$ = **null**;
16 if ($g(s') > c(s', s) + v(s)$)
17 $bp(s') = s$;
18 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembershipLower(s');
19 else
20 $v(s) = \infty$; UpdateSetMembership(s);
21 for each predecessor s' of s
22 if s' was never visited before
23 $v(s') = g(s') = \infty$; $bp(s')$ = **null**;
24 if ($bp(s') = s$)
25 $bp(s') = \text{argmin}_{s'' \in Succ(s')} c(s', s'') + v(s'')$;
26 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembership($s'$);

Figure 3.4: Delayed D*: ComputePathDelayed function

The Delayed D* Algorithm

The Delayed D* algorithm is shown in Figures 3.4 and 3.5. We have presented the algorithm in the same framework as that used to present D* Lite in Section 3.1 to highlight its similarities. Differences between Delayed D* and D* Lite are shown highlighted in red, and include two new functions *UpdateSetMembershipLower* and *FindRaiseStatesOnPath*. In our pseudocode, we have used notation defined earlier as well as some extra: *OPEN* is the priority queue, $h(s, s')$ is the heuristic cost from state s to state s', $\text{argmin}_{s' \in succ(s)} f()$ returns the successor of s for which function f is minimized, and all variables not already mentioned are local to the respective functions (*raise*, *loop*, *ctr*, c_{old}, u, etc).

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + h(s_start, s); g(s)];
 3 else
 4   return [v(s) + h(s_start, s); v(s)];

FindRaiseStatesOnPath()
 5 s = s_start; raise = FALSE; CLOSED = ∅;
 6 while (s ≠ s_goal)
 7   add s to CLOSED;
 8   bp(s) = argmin_{s'∈Succ(s)}(c(s, s') + v(s'));
 9   g(s) = c(s, bp(s)) + v(bp(s));
10   if (v(s) ≠ g(s))
11     UpdateSetMembership(s);
12     raise = TRUE;
13   if (bp(s) ∈ CLOSED)
14     return raise;
15   else
16     s = bp(s);
17 return raise;

Main()
18 g(s_start) = v(s_start) = ∞; v(s_goal) = ∞; bp(s_goal) = bp(s_start) = null;
19 g(s_goal) = 0; OPEN = ∅;
20 insert s_goal into OPEN with key(s_goal);
21 forever
22   ComputePathDelayed();
23   raise = FindRaiseStatesOnPath();
24   while (raise = TRUE)
25     ComputePathDelayed();
26     raise = FindRaiseStatesOnPath();
27   wait for changes in edge costs;
28   for all directed edges (u, v) with changed edge costs
29     update the edge cost c(u, v);
30     if (u ≠ s_goal)
31       if u was never visited before
32         v(u) = g(u) = ∞; bp(u) = null;
33       bp(u) = argmin_{s'∈Succ(u)} c(u, s') + v(s');
34       g(u) = c(u, bp(u)) + v(bp(u)); UpdateSetMembershipLower(u);
```

Figure 3.5: Delayed D*: Main function

Delayed D* begins by initializing the g-value of the goal state and placing this state in *OPEN*, exactly as in D* Lite. Next, *ComputePathDelayed* is called, which computes a least-cost path from s_{start} to s_{goal}. This initial path is computed in exactly the same way

as it would be in D* Lite.

Once the initial least-cost path has been found, Delayed D* waits for edge costs to change. When they do, it updates the g-value of all immediately affected states and adds only the newly *overconsistent* states to the queue (Figure 3.5, line 34). It then calls *ComputePathDelayed* again to propagate the new values of these overconsistent states through the state space (Figure 3.5, line 22). When it has finished, the solution path is checked in *FindRaiseStatesOnPath* for any underconsistent states. If any exist, they are placed on the queue and their new values are propagated through the state space by another call to *ComputePathDelayed.*

When a state s is expanded in *ComputePathDelayed*, it is treated differently depending on whether it is overconsistent or underconsistent. If it is overconsistent (Figure 3.4, lines 11 to 18), it updates its v-value to equal its g-value, then uses its new v-value to lower the g-values of its predecessors. If this causes some predecessor to become overconsistent, the predecessor is added to *OPEN*. Any predecessor states that are made underconsistent are ignored. This restricted addition of states to *OPEN* takes place in the *UpdateSetMembershipLower* function.

If the state s to be expanded is underconsistent (Figure 3.4, lines 19 to 26), it sets its v-value to infinity, then updates the g-values of all predecessors that use the v-value of the current state for their g-values. Any predecessors that are made either overconsistent or underconsistent as a result of this update are added to *OPEN*.

Since all underconsistent predecessor states are ignored whenever an overconsistent state is expanded, the v-values of states along the solution path returned by *ComputePathDelayed* are lower bounds of the optimal path costs of the states. Thus, each time *ComputePathDelayed* terminates, we must check that all the states along the solution path from s_{start} to s_{goal} are consistent. If they are, then we can guarantee that the solution path is optimal. This check is performed in the *FindRaiseStatesOnPath* function, which steps through the current path and looks for inconsistent states. Any such states that it finds have their g-values updated and are added to *OPEN*.

If *FindRaiseStatesOnPath* finds some inconsistent states on the current path, then *ComputePathDelayed* is called again to process these states and propagate their new values to other affected states. During this call, underconsistent states found on the path will be able to update the g-values of their predecessors, and if this causes these predecessors to become underconsistent, they will also be added to *OPEN*. As a result, the underconsistent states found on the path are used to seed a wave of underconsistent states propagating through the state space.

As stated earlier, the intuition behind Delayed D* is that underconsistent states that do not reside on the solution path can be ignored without sacrificing optimality. The algorithm tries to avoid processing these states as long as possible, but when some of these states turn

up on the solution path, it is forced to bite the bullet and update their values. It does this by performing a single update sweep that starts with all the underconsistent states on the path and propagates their new values to all the affected states[3]

The Delayed D* algorithm is guaranteed to terminate, and it is also guaranteed to produce an optimal solution path. The proofs of these properties can be found in the appendix.

Theorem 1. *The Delayed D* algorithm always terminates and an optimal solution path can then be followed from s_{start} to s_{goal} by always moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + v(s')$.*

As with D* Lite, it is straightforward to modify Delayed D* to account for a changing s_{start} in order to model an agent moving through the environment. For details on the required modifications, see [Ferguson and Stentz, 2005b]. Note that, for both the version of Delayed D* where s_{start} remains fixed and the version where s_{start} is changing, it is important to be careful when extracting the final solution path to be traversed. Specifically, when selecting the next state to transition to from some state s, ties can be broken by any method so desired, but the method must be consistent with that used to update back-pointers in *FindRaiseStatesOnPath*. If a particular back-pointer state $bp(s)$ is chosen from a state s in *FindRaiseStatesOnPath*, then this same state should be selected when extracting the final solution path. This is important for ensuring that the agent traverses the least-cost path shown to be valid in *FindRaiseStatesOnPath*. Evaluating the successors of each state in a fixed order is one simple way to guarantee this.

Delayed D* Results

In order to analyse the performance of Delayed D*, we compared the algorithm to the optimized version of D* Lite [Koenig and Likhachev, 2002] on three common path planning tasks. The first task was to maintain a least-cost path from the left side of an environment to the right side, as the terrain associated with areas of the environment changed. We generated 1050 random environments of size 500 × 500: 50 environments with no obstacles (but with varied terrain costs), 50 with 1% obstacle cells, 50 with 2% obstacle cells, and so on, up to 50 with 20% obstacle cells. The terrain cost associated with traversing non-obstacle cells was also randomly generated. We used Euclidean distance for our heuristic h.

For each environment, we first generated an initial optimal path using D* Lite. We then randomly selected 100 cells and flipped their terrain values: traversable cells became untraversable and untraversable cells became traversable. We then used D* Lite and Delayed

[3]Of course, not all the states in the state space whose path costs are affected are processed, only those whose priority values are less than that of s_{start}, as in D* Lite.

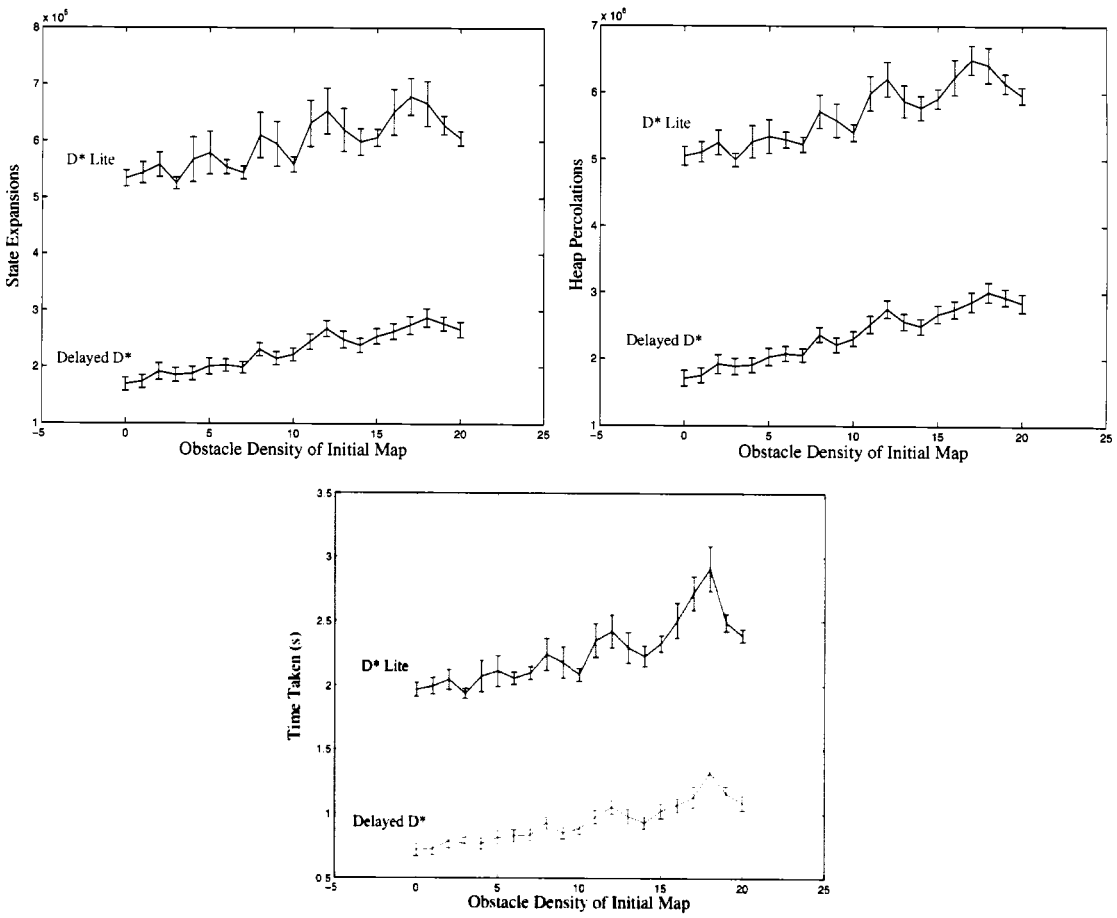


Figure 3.6: Results from our first experiment. A least-cost path was maintained between a fixed initial state and a goal state as cells in the environment had their traversability changed. Shown here are the number of states expanded (top-left), the number of heap percolations (top-right), and the total CPU time taken (bottom).

D* to replan an optimal path given these changes. We repeated the above steps 50 times (for a total of 5000 altered terrain costs) and recorded the performance of each algorithm in replanning.

Figure 3.6 shows the results of this experiment. We have included three performance measures: the number of states expanded, the number of heap percolations (i.e., the number of times a parent and a child are swapped in our heap priority queue), and the CPU time taken by a P3 1.4 GHz processor. In all our graphs, we have included error bars representing the standard error of the mean. According to all three measures, Delayed D* outperformed D* Lite by roughly a factor of 2.

Note that in this experiment the agent was not moving through the environment; we were maintaining an optimal path from a fixed start state to the goal. Such a situation

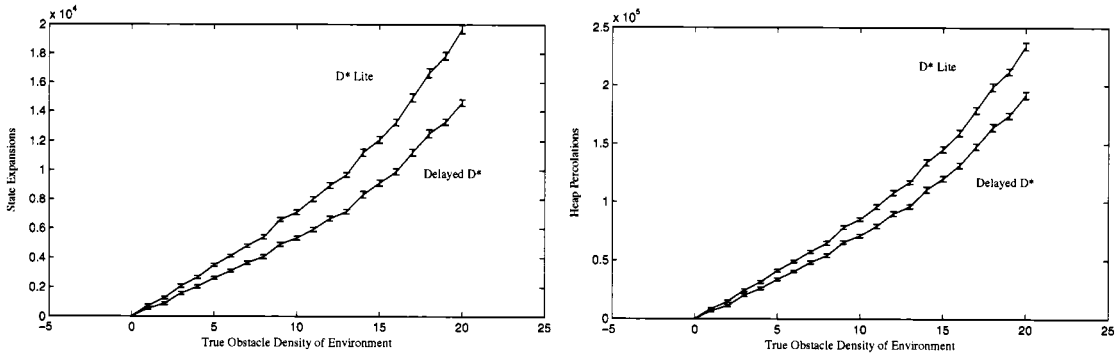


Figure 3.7: Results from our first navigation experiment. The agent began with an empty map and updated its map as it traversed the environment. Shown here are the number of states expanded and heap percolations.

arises when we have a number of agents traversing from the same initial position to the goal, or in domains such as network routing, where we are interested in maintaining an optimal path between two fixed states in our system.

The second and third tasks we looked at concerned an agent moving through a state space in which edge costs are changing or for which the agent had imperfect initial information. In these scenarios, the agent began with some prior information about the costs of edges between states and could update its information as it traversed through the state space. As we are most concerned with robot navigation, we simulated an agent equipped with a sensor that would allow it to detect the terrain value of areas of the environment within some sensor radius of the agent. For our testing, we again used the same series of 500×500 environments but made the common simplification that all traversable cells had the same terrain cost, resulting in a binary map. For each traverse, the agent started at the left side and worked its way to the right.

We first looked at an agent that began with *no* information about its environment, so that its initial map was completely empty and assumed to be everywhere traversable. As the agent moved through the environment, it updated its map to reflect the true nature of the environment, using an omnidirectional sensor with a 30-cell field of view. The results of this experiment are shown in Figure 3.7.

In these completely unknown environments, Delayed D* showed a slight performance improvement; this improvement increased with the difficulty of the environment. Because unknown environments are assumed to be completely free of obstacles, the number of states expanded when producing the initial solution path in these domains is very small. Thus, in such situations D* Lite performs in a similar manner to Delayed D*: the only states that are processed as a result of observed cost changes are those underconsistent states that reside on or are very close to the solution path.

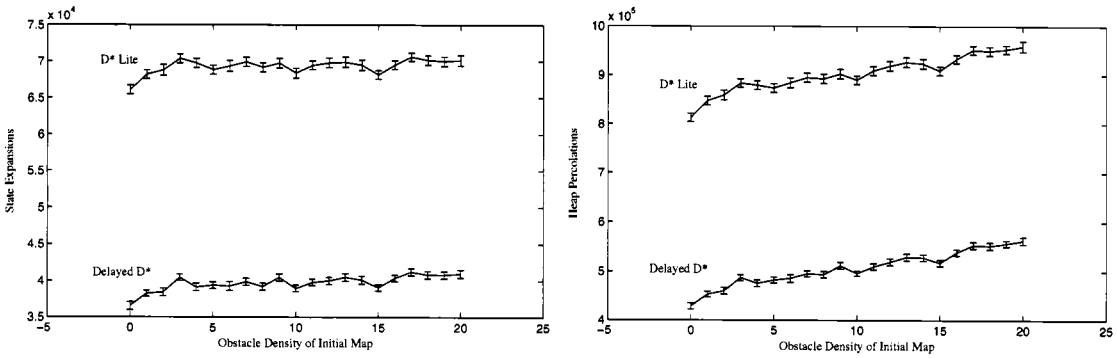


Figure 3.8: Results from our second navigation experiment. The agent began with a map that contained incorrect values for 25% of the cells. Shown here are the number of states expanded and heap percolations.

Our final simulation concerned an agent that began with a complete but inaccurate map of its environment. Here, we used the same series of 500 × 500 environments, but we randomly flipped the terrain values of 25% of the cells, so that traversable cells became untraversable, and vice versa. The agent then moved through the environment, updating its map information to reflect the observed environment. The results of this experiment are shown in Figure 3.8.

In these partially known environments, where the number of states with key values less than the start state can be large and costs both increase and decrease, Delayed D* showed a significant improvement in efficiency.

It is worth making two points about the version of Delayed D* used for our experiments. Firstly, in our simulations we dealt with random environments. However, real environments navigated by mobile robots may exhibit structure. In such cases, we may want to alter the *FindRaiseStatesOnPath* function to add to the queue not just the underconsistent states found on the current solution path, but all states adjacent to these states that are also underconsistent. If the agent were to observe a large obstacle in its path, this would prevent it from potentially processing only a small "wedge" of the obstacle each time *ComputePathDelayed* is called. Instead, if the entire obstacle was added at once, it could be processed more efficiently.

Secondly, when a cell becomes untraversable (i.e., an obstacle) during our simulations, it is *never* again placed on the queue, even if its least-cost successor is popped as an underconsistent state. Once a cell becomes an obstacle, any dependent predecessors are updated and the cell itself is afterwards ignored. This means that the propagation of underconsistent states can terminate early if obstacles are encountered. This early termination further delays the processing of underconsistent states and contributes to the overall efficiency gain of our Delayed D* implementation.

Looking critically at the Delayed D* algorithm, two possible sources of inefficiency can be found. Firstly, it is possible to construct worst-case scenarios where the processing of underconsistent states changes the solution path several times, each time producing a new path containing underconsistent states. This results in a number of propagation phases, each starting from a new set of underconsistent states, where each propagation phase may process roughly the same area of the state space. This will be less efficient than dealing with all the relevant underconsistent states at once. However, in realistic navigation tasks, worst-case scenarios occur very infrequently. As our results have suggested, Delayed D* is far more efficient on average than D* Lite. In fact, over all our test cases (1050 environments, 3150 total test runs), there was not a single run during which D* Lite expanded fewer states than Delayed D*.

The second possible source of inefficiency concerns the *FindRaiseStatesOnPath* function. Since Delayed D* ignores underconsistent states when they first appear, the consistency of the current solution path needs to be checked after each propagation phase. This adds a source of computation to the planning task not associated with competing algorithms such as D* Lite. However, the extra processing required to perform this check is only influential in the most trivial of state spaces. Because the solution path always represents a one dimensional slice through the space, in complex or higher-dimensional state spaces (where the underconsistent states processed unnecessarily by D* Lite become a real burden) this check requires a relatively insignificant amount of computation.

For typical mobile robot navigation scenarios, we have found Delayed D* to be a valuable addition to the D* family of replanning algorithms. Its ability to delay the processing of underconsistent states as long as possible can save a significant amount of computation, and this idea can be applied in several other path planning domains.

3.3 Field D*

In Chapter 2, we mentioned that in mobile robotics is common to represent both indoor and outdoor environments as uniform resolution 2D traversability grids, in which cells are given a traversal cost reflecting the difficulty of navigating the corresponding area of the environment. As long as these costs are expanded to reflect the physical dimensions of the vehicle, then planning a path for a robot translates to generating a trajectory through this grid for a single point.

We also described how this grid can be transformed into a graph, and how we can efficiently plan and replan paths over the subsequent graph. Unfortunately, paths produced using this graph are restricted to headings of $\frac{\pi}{4}$ increments. This means that the final solution path may be suboptimal in path cost, involve unnecessary turning, or both.

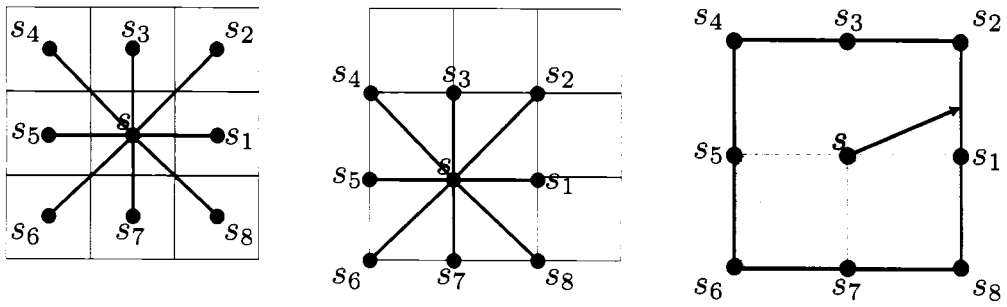


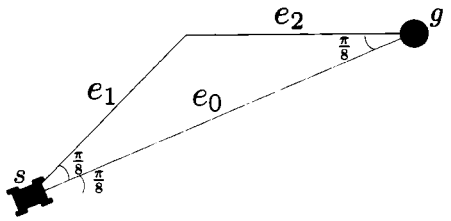
Figure 3.9: (a) A standard 2D grid used for global path planning in which states reside at the centers of the grid cells. The edges emanating from the center state represent all the possible actions that can be taken from this state. (b) A modified representation used by Field D*, in which states reside at the corners of grid cells. (c) The optimal path from state s must intersect one of the edges $\{\overrightarrow{s_1s_2}, \overrightarrow{s_2s_3}, \overrightarrow{s_3s_4}, \overrightarrow{s_4s_5}, \overrightarrow{s_5s_6}, \overrightarrow{s_6s_7}, \overrightarrow{s_7s_8}, \overrightarrow{s_8s_1}\}$.

In this section, we present an algorithm for planning and replanning more direct, less-costly paths through grids. We begin by describing the limitations of standard grid-based paths, and discuss a number of improvements that have been proposed by various researchers. We then introduce *Field D**, a replanning algorithm that uses interpolation to plan better paths through grids. We present a number of example paths, experimental results, and several implementations on robotic systems.

To begin with, consider a robot facing its goal in a completely obstacle-free environment (see Figure 3.10). Obviously, the optimal path is a straight line between the robot and the goal. However, if the robot's initial heading is not a multiple of $\frac{\pi}{4}$, traditional grid-based planners would return a path that has the robot first turn to attain the nearest grid heading, move some distance along this heading, and then turn $\frac{\pi}{4}$ in the opposite direction of its initial turn and continue to the goal. Not only does this path have clearly suboptimal length, it contains possibly expensive or difficult turns that are purely artifacts of the limited representation. Such global paths, when coupled with the results of a local planner, cause the robot to behave suboptimally. Further, this limitation of traditional grid-based planners is not alleviated by increasing the resolution of the grid.

Sometimes it is possible to reduce the severity of this problem by post-processing the path. Usually, given a robot location s, one finds the furthest point p along the solution path for which a straight line path from s to p is collision-free, then replaces the original path to p with this straight line path. However, this does not always work, as illustrated by Figure 3.11. Indeed, for non-uniform cost environments such post-processing can often *increase* the cost of the path.

A more comprehensive post-processing approach is to take the result of the global planner and use it to seed a higher dimensional planner that incorporates the kinematic or



$$|e_1| = |e_2|$$
$$\rightarrow |e_0|^2 = 2|e_1|^2 - 2|e_1|^2 \cos\tfrac{3\pi}{4}$$
$$\rightarrow |e_1| + |e_2| \approx 1.08|e_0|$$

Figure 3.10: A uniform resolution 2D grid-based path (e_1 plus e_2) between two grid states can be up to 8% longer than an optimal straight-line path (e_0). Here, the desired straight-line heading is $\frac{\pi}{8}$ and lies perfectly between the two nearest grid-based headings of 0 and $\frac{\pi}{4}$. This result is independent of the resolution of the grid.

dynamic constraints of the robot. Stachniss and Burgard [Stachniss and Burgard, 2002] present an approach that takes the solution generated by the global planner and uses it to extract a local waypoint to use as the goal for a 5D trajectory planner. The search space of the 5D planner is limited to a small area surrounding the global solution path. Likhachev, Gordon, and Thrun [Likhachev et al., 2003, Likhachev et al., 2005a] present an approach that uses the cost-to-goal value function of the global planner to focus an anytime global 4D trajectory planner. Their approach improves the quality of the global trajectory while deliberation time allows. However, these higher dimensional approaches can be much more computationally expensive than standard grid-based planners and are still influenced by the results of the initial grid-based solution.

Planning a path through a varying-cost grid can be seen as a specific instance of the Weighted Region Problem [Mitchell and Papadimitriou, 1991], where the regions are uniform square tiles. A number of algorithms exist to solve this problem in the computational geometry literature (see [Mitchell, 2000] for a good survey). In particular, Mitchell and Papadimitriou [Mitchell and Papadimitriou, 1991] and Rowe and Richbourg [Rowe and Richbourg, 1990] present approaches based on Snell's law of refraction that compute optimal paths by simulating a series of light rays that propagate out from the start position and refract according to the different traversal costs of the regions encountered. These approaches are efficient for planning through environments containing a small number of homogenous-cost regions, but are computationally expensive when the number of such regions is very large, as in the case of a uniform grid with varying cell costs.

Recently, robotics researchers have looked at more sophisticated methods of obtaining better paths through grids without sacrificing too much of the efficiency of the classic grid-based approach described above. Konolige [Konolige, 2000] presents an interpolated planner that first uses classic grid-based planning to construct a cost-to-goal value function over the grid and then interpolates this result to produce a shorter path from the initial position to the goal. This method results in shorter, less-costly paths for agents to traverse

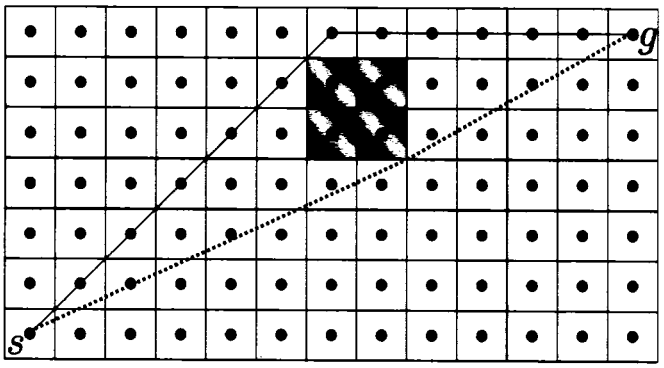


Figure 3.11: 2D grid-based paths cannot always be shortened in a post-processing phase. Here, the grid-based path from s to g (top, in black) cannot be shortened because there are four obstacle cells (shaded). The optimal path is shown in blue/dashed.

but does not incorporate the reduced path cost into the planning process. Consequently, the resulting path is not necessarily as good as the path the algorithm would produce if interpolated costs were calculated during planning. Further, if we are computing paths from several locations (which is common when combining the global planner with a local planner) then this post-processing interpolation step can be expensive. Also, this approach provides no replanning functionality to update the solution when new information concerning the environment is received.

Philippsen and Siegwart [Philippsen and Siegwart, 2005] present an algorithm based on Fast Marching Methods [Sethian, 1996] that computes a value function over the grid by growing a surface out from the goal to every region in the environment. The surface expands according to surface flow equations, and the value of each grid point is computed by combining the values of two neighboring grid points. This approach incorporates the interpolation step into the planning process, producing low-cost, interpolated paths. The resulting algorithm is similar in performance to an approach originally developed by Tsitsiklis [Tsitsiklis, 1995], with the added benefit of being able to efficiently repair previous solutions when new information is received. Philippsen has shown this technique generates nice paths in indoor environments [Philippsen, 2004, Philippsen and Siegwart, 2005]. However, the search is not focused towards the robot location (such as in A*) and assumes that the transition cost from a particular grid state to each of its neighbors is constant. Consequently, it is not as applicable to navigation in outdoor environments, which are often best represented by large grids with widely-varying cell traversal costs.

The idea of using interpolation to produce better value functions for discrete samples over a continuous state space is not new. This approach has been used in dynamic programming for some time to compute the value of successors that are not in the set of samples [Larson, 1967, Larson and Casti, 1982, LaValle, 2006]. However, as LaValle points out

[LaValle, 2006], this becomes difficult when the action space is also continuous, as solving for the value of a state now requires minimizing over an infinite set of successor states.

We have developed an extension of the widely-used D* family of algorithms that uses linear interpolation to produce near-optimal paths that eliminate unnecessary turning. It relies upon an efficient, closed-form solution to the above minimization problem for 2D grids, which we introduce below. This method produces much straighter, less-costly paths than classical grid-based planners without sacrificing real-time performance. As with D* and D* Lite, our approach focuses its search towards the most relevant areas of the state space during both initial planning and replanning. Further, it takes into account local variations in cell traversal costs and produces paths that are optimal given a linear interpolation assumption. As the resolution of the grid increases, the solutions returned by the algorithm improve, approaching true optimal paths.

Improving Cost Estimation through Interpolation

The key to our algorithm is a novel method for computing the path cost of each grid state s given the path costs of its neighboring states. By the path cost of a state s we mean the cost of the cheapest path from s to the goal s_{goal}. In classical graph-based planning this value is computed as

$$g(s) = \min_{s' \in Succ(s)} (c(s, s') + v(s')), \tag{3.1}$$

where $Succ(s)$ is the set of all successor states of s (see Figure 3.9 for the set of successors of a state s in a grid), $c(s, s')$ is the cost of traversing the edge between s and s', and $v(s')$ is the path cost of state s', as introduced in Section 3.1.

When applied to graphs extracted from grids, this calculation assumes that the only transitions possible from state s are straight-line trajectories to one of its neighboring states. This assumption results in the limitations of grid-based plans discussed earlier. However, consider relaxing this assumption and allowing a straight-line trajectory from state s to any point on the boundary of its grid cell. If we knew the value of every point s_b along this boundary, then we could compute the optimal value of state s simply by minimizing $c(s, s_b) + v(s_b)$, where $c(s, s_b)$ is computed as the distance between s and s_b multiplied by the traversal cost of the cell in which s resides. Unfortunately, there are an infinite number of such points s_b and so computing $v(s_b)$ for each of them is not possible.

It is possible, however, to provide an approximation to $v(s_b)$ for each boundary point s_b by using linear interpolation. To do this, we first modify the graph extraction process discussed earlier. Instead of assigning states to the centers of grid cells, we assign states to the *corners* of each grid cell, with edges connecting states that reside at corners of the same grid cell (see Figure 3.9(b)).

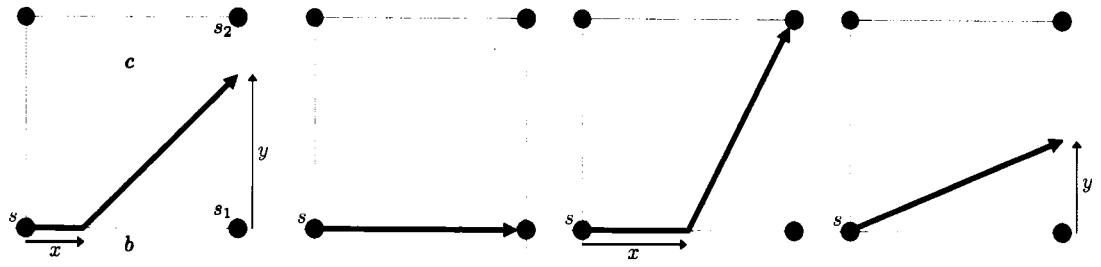


Figure 3.12: Computing the path cost of state s using the path cost of two of its neighbors, s_1 and s_2, and the traversal costs c of the center cell and b of the bottom cell. Illustrations (ii) through (iv) show the possible optimal paths from s to edge $\overrightarrow{s_1 s_2}$.

Given this modification, the traversal costs of any two equal-length segments of an edge will be the same. This differs from the original graph extraction process in which the first half of an edge was in one cell and the second half was in another cell, with the two cells possibly having different traversal costs. In the modified approach the cost of an edge that resides on the boundary of two grid cells is defined as the minimum of the traversal costs of each of the two cells.

We then treat the states in our graph as sample points of a continuous cost field. The optimal path from a state s must pass through an edge connecting two consecutive neighbors of s, for example $\overrightarrow{s_1 s_2}$ (see Figure 3.9(c)). The path cost of s is thus set to the minimum cost of a path through any of these edges, which are considered one at a time. To compute the path cost of state s using edge $\overrightarrow{s_1 s_2}$, we use the path costs of states s_1 and s_2 and the traversal costs c of the center cell and b of the bottom cell (see Figure 3.12).

To compute this cost efficiently, we assume the path cost of any point s_y residing on the edge between s_1 and s_2 is a linear combination of $v(s_1)$ and $v(s_2)$:

$$g(s_y) = yv(s_2) + (1 - y)v(s_1), \tag{3.2}$$

where y is the distance from s_1 to s_y (assuming unit cells). This assumption is not perfect: the path cost of s_y may not be a *linear* combination of $v(s_1)$ and $v(s_2)$, nor even a function of these path costs. However, this linear approximation works well in practice, and allows us to construct a closed form solution for the path cost of state s.

Given this approximation, the path cost of s given s_1, s_2, and cell costs c and b can be computed as:

$$\min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + yv(s_2) + (1 - y)v(s_1)], \tag{3.3}$$

where $x \in [0, 1]$ is the distance traveled along the bottom edge from s before cutting across the center cell to reach the right edge a distance of $y \in [0, 1]$ from s_1 (see Figure

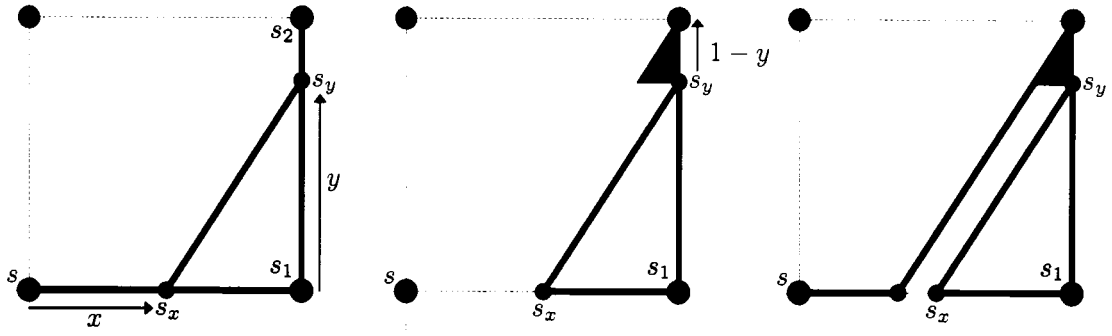


Figure 3.13: (a) Imagine the blue path is the optimal path, traveling along the bottom edge to s_x then across the center cell to s_y, then up the right edge to s_2. Notice the triangle formed between vertices s_x, s_1, and s_y. (b) We can create a scaled version of this triangle with a vertical edge length of $1 - y$. (c) Combining the hypotenuses of the two triangles shown in (b) produces a lower-cost path than the one shown in (a), forcing a contradiction.

3.12(i)). Note that if both x and y are zero in the above equation the path taken is along the bottom edge but its cost is computed from the traversal cost of the center cell.

Let (x^*, y^*) be a pair of values for x and y that solve the above minimization. Because of the linear interpolation, at least one of these values will be either zero or one. Intuitively, if it is less expensive to partially cut through the center cell than to traverse around the boundary, then it is least expensive to completely cut through the cell. Thus, if there is any component to the cheapest solution path from s that cuts through the center cell, it will be as large as possible, forcing $x^* = 0$ or $y^* = 1$. If there is no component of the path that cuts through the center cell, then $y^* = 0$. The proof of this is included in the appendix.

Theorem 2. *Let* $\{x^*, y^*\} = argmin_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + yv(s_2) + (1-y)v(s_1)]$, $x \in [0,1]$, $y \in [0,1]$, *where* s_1, s_2, b, *and* c *are as defined in the text, and* $v(s_1)$ *and* $v(s_2)$ *are optimal path costs for* s_1 *and* s_2 *given our linear interpolation assumption. Then* $x^* \in \{0,1\}$ *or* $y^* \in \{0,1\}$.

From this theorem, we know that no optimal path involves traveling along sections of both the bottom and right edges *and* a component cutting across part of the center cell[4]. Instead, the path will either travel along the entire bottom edge to s_1 (Figure 3.12(ii)), or will travel a distance x along the bottom edge then take a straight-line path directly to s_2 (Figure 3.12(iii)), or will take a straight-line path from s to some point s_y on the right edge

[4]Note that it may be possible that the optimal path to s_2 involves traveling along the vertical edge from s for some distance, then cutting across to s_2. However, this possibility is examined when computing the path cost from s to neighbors s_2 and s_3 (see Figure 3.9). We can thus restrict our attention when computing the path cost of s using neighbors s_1 and s_2 to paths that fully reside within the triangle defined by vertices s, s_1, and s_2.

```
ComputeCost(s, s_a, s_b)
 1  if (s_a is a diagonal neighbor of s)
 2    s_1 = s_b; s_2 = s_a;
 3  else
 4    s_1 = s_a; s_2 = s_b;
 5  c is traversal cost of cell with corners s, s_1, s_2;
 6  b is traversal cost of cell with corners s, s_1 but not s_2;
 7  if (min(c, b) = ∞)
 8    g = ∞;
 9  else if (v(s_1) ≤ v(s_2))
10    g = min(c, b) + v(s_1);
11  else
12    f = v(s_1) − v(s_2);
13    if (f ≤ b)
14      if (c ≤ f)
15        g = c√2 + v(s_2);
16      else
17        y = min(f/√(c^2 − f^2), 1);
18        g = c√(1 + y^2) + f(1 − y) + v(s_2);
19    else
20      if (c ≤ b)
21        g = c√2 + v(s_2);
22      else
23        x = 1 − min(b/√(c^2 − b^2), 1);
24        g = c√(1 + (1 − x)^2) + bx + v(s_2);
25  return g;
```

Figure 3.14: The Interpolation-based Path Cost Calculation

(Figure 3.12(iv)). Which of these paths is cheapest depends on the relative sizes of c, b, and the difference f in path cost between s_1 and s_2: $f = v(s_1) - v(s_2)$. Specifically, if $f < 0$ then the optimal path from s travels straight to s_1 and will have a cost of $(\min(c, b) + v(s_1))$ (Figure 3.12(ii)). If $f = b$ then the cost of a path using some portion of the bottom edge (Figure 3.12(iii)) will be equivalent to the cost of a path using none of the bottom edge (Figure 3.12(iv)). We can solve for the value of y that minimizes the cost of the latter path as follows.

First, let $k = f = b$. The path cost of s is:

$$c\sqrt{1 + y^2} + k(1 - y) + v(s_2). \tag{3.4}$$

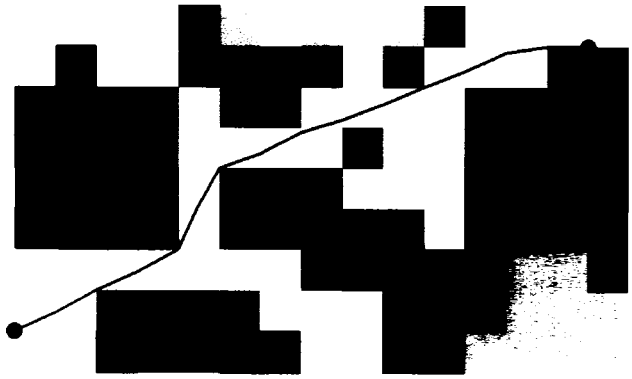


Figure 3.15: A close-up of a path planned using Field D* showing individual grid cells. Darker cells have larger traversal costs. Notice that the path is not limited to entering and exiting cells at corner points.

Taking the derivative of this cost with respect to y and setting it equal to zero yields:

$$y^* = \sqrt{\frac{k^2}{c^2 - k^2}}. \tag{3.5}$$

Whether the bottom edge or the right edge is used, we end up with the same calculations and path cost computations. So all that matters is which edge is cheaper. If $f < b$ then we use the right edge and compute the path cost as above (with $k = f$), and if $b < f$ we use the bottom edge and substitute $k = b$ and $y^* = 1 - x^*$ into the above equation. The resulting algorithm for computing the minimum-cost path from s through an edge between *any* two consecutive neighbors s_a and s_b is provided in Figure 3.14. Given the minimum-cost paths from s through each of its 8 neighboring edges, we can compute the path cost for s to be the cost of the cheapest of these paths. The associated path is optimal given our linear interpolation assumption.

The Field D* Algorithm

Once equipped with this interpolation-based path cost calculation for a given state in our graph, we can plug it into any of a number of current planning and replanning algorithms to produce low-cost paths. Figures 3.16 and 3.17 present one formulation of *Field D**, an incremental replanning algorithm that incorporates these interpolated path costs [Ferguson and Stentz, 2005a, Ferguson and Stentz, 2006d]. This version of Field D* is based on D* Lite, with differences shaded in red (Figure 3.16 lines 12, 14 to 17, and 24 to 26; Figure 3.17 lines 9 to 16).

In these figures, $connbrs(s)$ contains the set of consecutive neighbor pairs of state s: $connbrs(s) = \{(s_1,s_2), (s_2,s_3), (s_3,s_4), (s_4,s_5), (s_5,s_6), (s_6,s_7), (s_7,s_8), (s_8,s_1)\}$, where s_i is

UpdateSetMembership(s)

```
1  if (v(s) ≠ g(s))
2    insert/update s in OPEN with key(s);
3  else
4    if (s ∈ OPEN) remove s from OPEN;
```

ComputeInterpolatedPath()

```
5  while(key(s_start) > min_{s∈OPEN}(key(s)) or v(s_start) < g(s_start))
6    remove s with the smallest key(s) from OPEN;
7    if (v(s) > g(s))
8      v(s) = g(s);
9      for each predecessor s' of s
10       if s' was never visited before
11         v(s') = g(s') = ∞; bp(s') = null;
12       if (g(s') > ComputeCost(s', s, ccknbr(s', s)))
13         bp(s') = s;
14         g(s') = ComputeCost(s', bp(s'), ccknbr(s', bp(s')));
15       if (g(s') > ComputeCost(s', cknbr(s'), s))
16         bp(s') = cknbr(s', s);
17         g(s') = ComputeCost(s', bp(s'), ccknbr(s', bp(s')));
18       UpdateSetMembership(s');
19   else
20     v(s) = ∞; UpdateSetMembership(s);
21     for each predecessor s' of s
22       if s' was never visited before
23         v(s') = g(s') = ∞; bp(s') = null;
24       if (bp(s') = s) or (bp(s') = cknbr(s', s))
25         bp(s') = argmin_{s''∈Succ(s')} ComputeCost(s', s'', ccknbr(s', s''));
26         g(s') = ComputeCost(s', bp(s'), ccknbr(s', bp(s')));
27         UpdateSetMembership(s');
```

Figure 3.16: Field D*: ComputeInterpolatedPath function.

positioned as shown in Figure 3.9(c).

As with D* Lite, each state s keeps track of a back-pointer $bp(s)$ specifying from which of its successor states it currently derives its path cost. Since in Field D* the successor of each state is a point on an edge connecting two of its neighboring states, this back-pointer needs to specify the two states that form the endpoints of this edge. We use $bp(s)$ to refer to the most *clockwise* of the two endpoint states relative to state s. For example, if the current best path from s intersects edge $\overrightarrow{s_1 s_2}$ (see Figure 3.9), then $bp(s) = s_1$. We also make use of two new operators, $cknbr(s, s')$ and $ccknbr(s, s')$, that, given a state s and some successor state s' of s, return the next successor state of s in the clockwise and counter-clockwise directions, respectively. Thus, using the state labels from Figure 3.9, $cknbr(s, s_1) = s_8$,

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + h(s_start, s); g(s)];
 3 else
 4   return [v(s) + h(s_start, s); v(s)];
Main()
 5 g(s_start) = v(s_start) = ∞; v(s_goal) = ∞; bp(s_goal) = bp(s_start) = null;
 6 g(s_goal) = 0; OPEN = ∅;
 7 insert s_goal into OPEN with key(s_goal);
 8 forever
 9   ComputeInterpolatedPath();
10   wait for changes in cell traversal costs;
11   for all cells x with new traversal costs
12     for each state s on a corner of x
13       if s was never visited before
14         v(s) = g(s) = ∞; bp(s) = null;
15       bp(s) = argmin_{s'∈Succ(s)} ComputeCost(s, s', ccknbr(s, s'));
16       g(s) = ComputeCost(s, bp(s), ccknbr(s, bp(s)));
17       UpdateSetMembership(s);
```

Figure 3.17: Field D*: Main function

$cknbr(s, s_8) = s_7$, etc, and $ccknbr(s, s_1) = s_2$, $ccknbr(s, s_2) = s_3$, etc. Thus, if $bp(s) = s_1$ then $ccknbr(s, bp(s)) = s_2$.

Apart from this construction, notation follows the D* Lite algorithm: $v(s)$ stores the path cost of state s the last time it was expanded (its v-value), $g(s)$ is the current path cost for s (its g-value), *OPEN* is a priority queue containing inconsistent states (i.e., states s for which $v(s) \neq g(s)$) in increasing order of *key* values (Figure 3.17 lines 1 through 4), s_{start} is the initial agent state, and s_{goal} is the goal state. $h(s_{start}, s)$ is a heuristic estimate of the cost of a path from s_{start} to s. As with D* Lite, a lexicographic ordering is used on the key values: key(s) < key(s') iff the first element of key(s) is less than the first element of key(s') or the first element of key(s) equals the first element of key(s') and the second element of key(s) is less than the second element of key(s').

Note that some sections of the algorithm as presented are clearly not as efficient as they could be (e.g. the repeated path cost calculation in Figure 3.16 lines 12, 14, 15, 17, 25, and 26) and have only been shown in the current form for clarity. Further, a number of optimizations exist for Field D*, details of which, along with various extensions, can be found in [Ferguson and Stentz, 2006d].

Because the path cost for a state can be a combination of the costs of two other states, not just one, generating a heuristic function to use for Field D* is not as easy as for D* Lite or

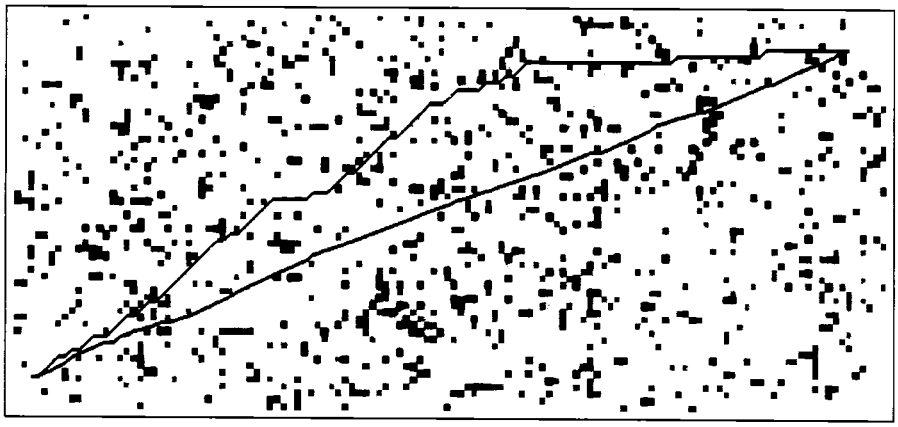


Figure 3.18: Paths produced by D* Lite (top) and Field D* (bottom) in a 150×60 uniform resolution grid. Again, darker cells have larger traversal costs.

A*. If no heuristic is used, i.e. $h(s_{start}, s) = 0$ for all s, then each state will only be expanded once and the resulting path will be optimal given our linear interpolation assumption. However, when using informed heuristics, it is more difficult to provide guarantees. This is because the path cost of a state s may be as little as $(\sqrt{2} - 1)$ more than the path cost of the most-costly of its two back-pointer states (call this state s_p)[5]. This doesn't leave much room for heuristic contributions to the *key* values that will preserve an efficient expansion ordering. For example, if we use a Euclidean heuristic function, then the difference between the heuristic value of s and s_p may be as high as 1 (i.e. $h(s_{start}, s_p) = h(s_{start}, s)+1$) if s_{start} is spatially located on the same side of s_p as s. In such a case, s would be expanded before s_p if both were on the queue with optimal costs. This results in multiple expansions of states and can be quite inefficient. In our implementations, we typically take the Euclidean heuristic and we divide it by two, then use this for our heuristic contribution to the *key* values. Although this is not guaranteed to expand states in the best possible order in the worst case, it typically performs well and strikes a good balance between the number of unique states expanded and the number of times each state is expanded.

Once the cost of a path from the initial state to the goal has been calculated, the path can be extracted by starting at the initial position and iteratively computing the cell boundary point to move to next. Because of our interpolation-based cost calculation, it is possible to compute the path cost of *any* point inside a grid cell, not just the corners, which is useful for both extracting the entire path and calculating accurate path costs from non-corner points. In the following section we provide more details on this extraction process.

Figures 3.15, 3.18, and 3.19 illustrate paths produced by Field D* through three non-

[5]This occurs when s_p is a non-diagonal neighbor of s and the path cost of s_p is greater by 1 than the cost of the other (diagonal) back-pointer of s.

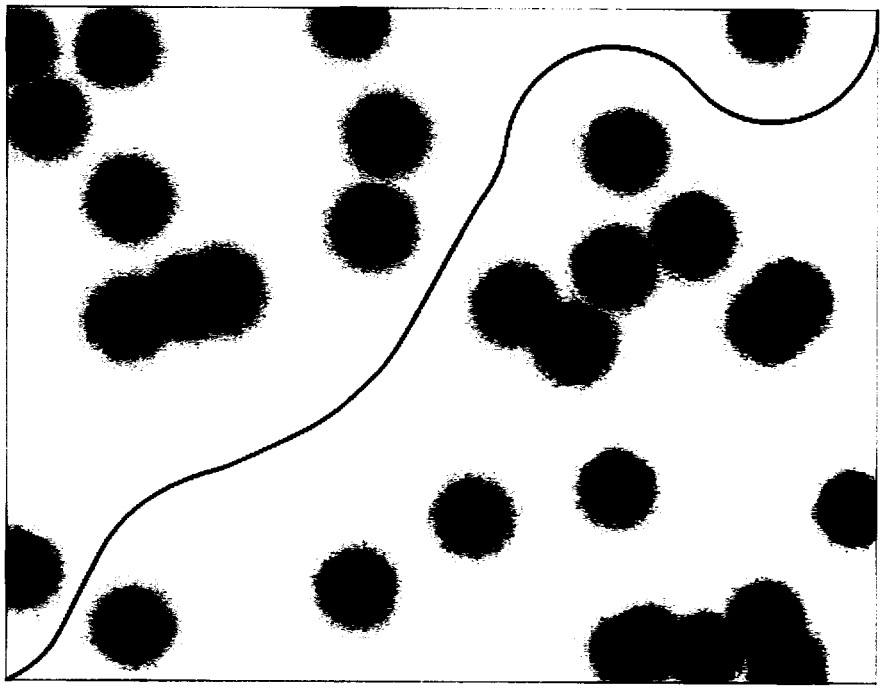


Figure 3.19: Field D* planning through a potential field of obstacles.

uniform cost environments. In each of these figures, darker areas represent regions that are more costly to traverse. Notice that, unlike paths produced using classical grid-based planners, the paths produced using Field D* are not restricted to a small set of headings. As a result, Field D* provides lower-cost paths through both uniform and non-uniform cost environments.

Path Extraction

The linear interpolation assumption made by Field D* generally produces accurate path cost approximations. However, this assumption is clearly not perfect, and there are situations in which it is seriously violated. In order to avoid returning invalid or grossly suboptimal paths in these cases, the path extraction process must be performed carefully.

In particular, we've found that we can reduce errors due to our interpolation-based approximation by using a one-step lookahead when computing the next waypoint in the path. Basically, before transitioning to an edge point p for which we have computed a simple interpolated path cost, we calculate a more accurate approximation of the path cost of p. We do this by looking to its neighboring edges and computing a locally-optimal path from p given the path costs of the endpoint states of these edges and interpolated path

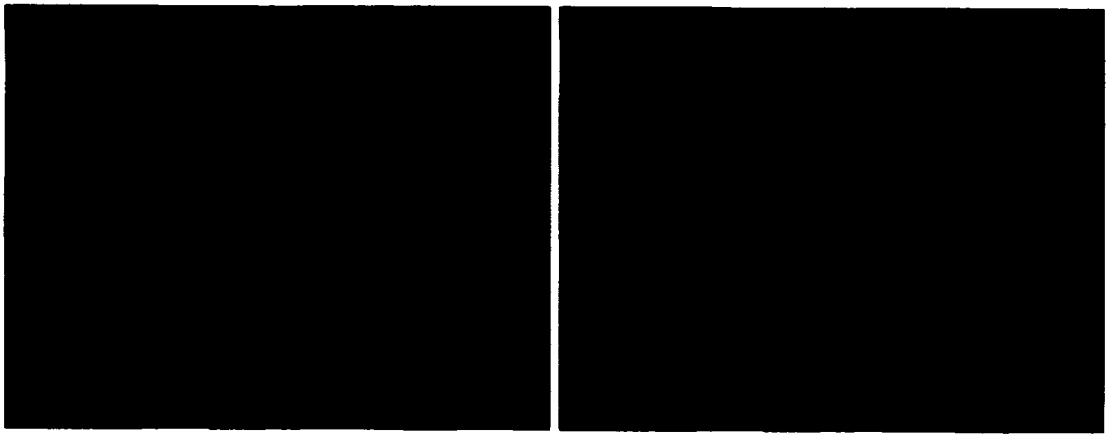


Figure 3.20: Paths produced by D* Lite (left) and Field D* (right) in a 900 × 700 binary cost grid. Here, obstacles are shown in blue/dark gray and traversable area is shown in black.

costs for points along the edges (as in Figure 3.14). Then, given this new path cost for p, we check if it is still the best point to use as the next waypoint in the path. This simple step reduces the effects of interpolation error on our path, particularly in pathological cases such as the one discussed in Section 3.3.

One other small modification we have found useful is to make sure that if one point in the path resides within some grid cell g, then the next point in the path can only be a corner of g if the computed path from the state at that corner does not transition back through g. Again, due to our use of interpolation this is not always naturally the case.

In practise it is most effective to use Field D* to compute the cost-to-goal value function over the grid, and use some local planner to compute the actual vehicle trajectory, as described in Section 2.1.

Field D* Results

The true test of an algorithm is its practical effectiveness. We have found Field D* to be extremely useful for a wide range of robotic systems navigating through terrain of varying degrees of difficulty (see Figure 3.23). Figure 3.21 shows a simulated example of Field D* being used to navigate a robot through an initially unknown environment. Figure 3.22 shows results from an autonomous passenger vehicle using Field D* to navigate an unstructured enviroment.

To provide a quantitative comparison of the performance of Field D* relative to D* Lite, we ran a number of replanning simulations in which we measured both the relative solution path costs and runtimes of the optimized versions of the two approaches. We generated 100 different 1000 × 1000 non-uniform cost grid environments in which each grid cell was

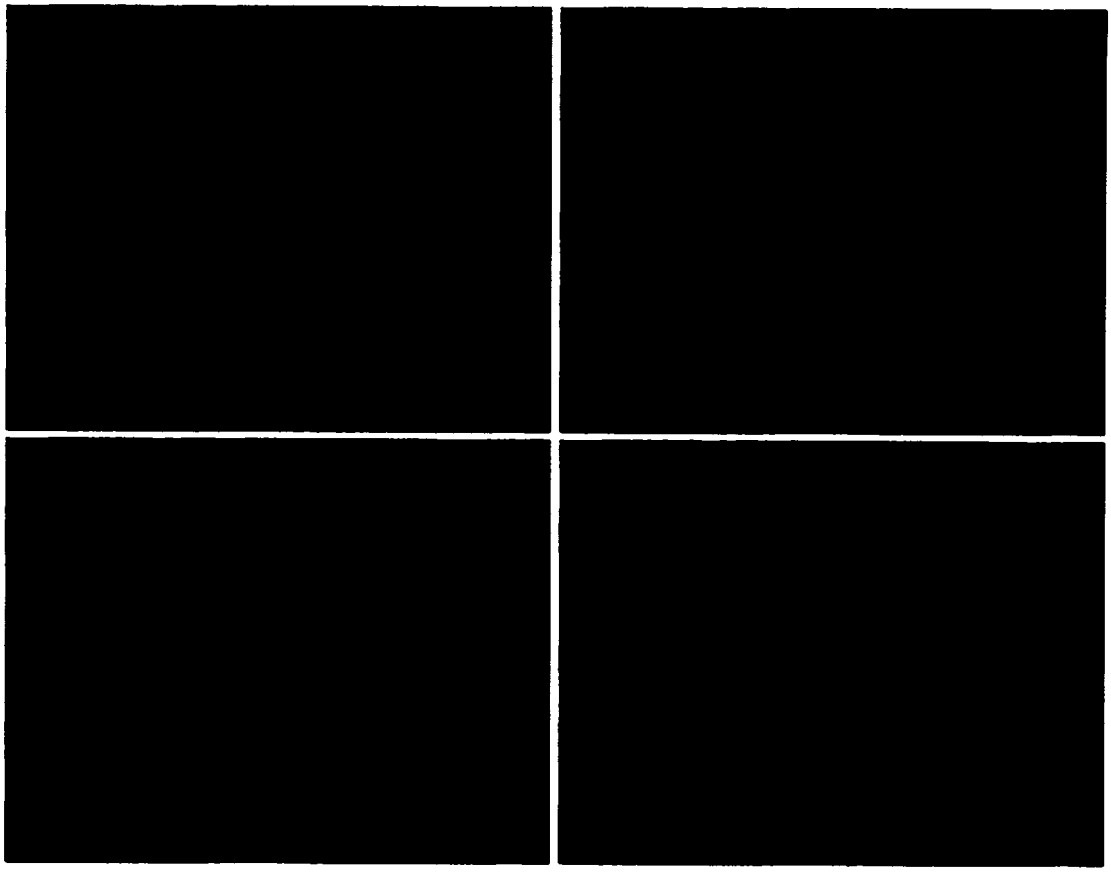


Figure 3.21: A robot navigation example using Field D*. The robot starts at the top of the environment (about two-thirds of the way to the right edge) and plans a path (in white) to the bottom left corner, assuming the environment is empty. As it traverses its path (in green/light gray), it receives updated environmental information through an onboard sensor (observed obstacles shown in blue/dark gray, actual obstacles shown in red/gray, traversable area shown in black). At each step it repairs its previous solution path based on this new information. Notice that the path segments are straight lines with widely-varying headings.

assigned an integer traversal cost between 1 (free space) and 16 (obstacle). With probability 0.5 this cost was set to 1, otherwise it was randomly selected. For each environment, the initial task was to plan a path from the lower left corner to a randomly selected goal on the right edge. After this initial path was planned, we randomly altered the traversal costs of cells close to the agent (10% of the cells in the environment were changed) and had each approach repair its solution path. This represents a significant change in the information held by the agent and results in a large amount of replanning.

The results from these experiments are shown in Table 3.1. During initial planning,

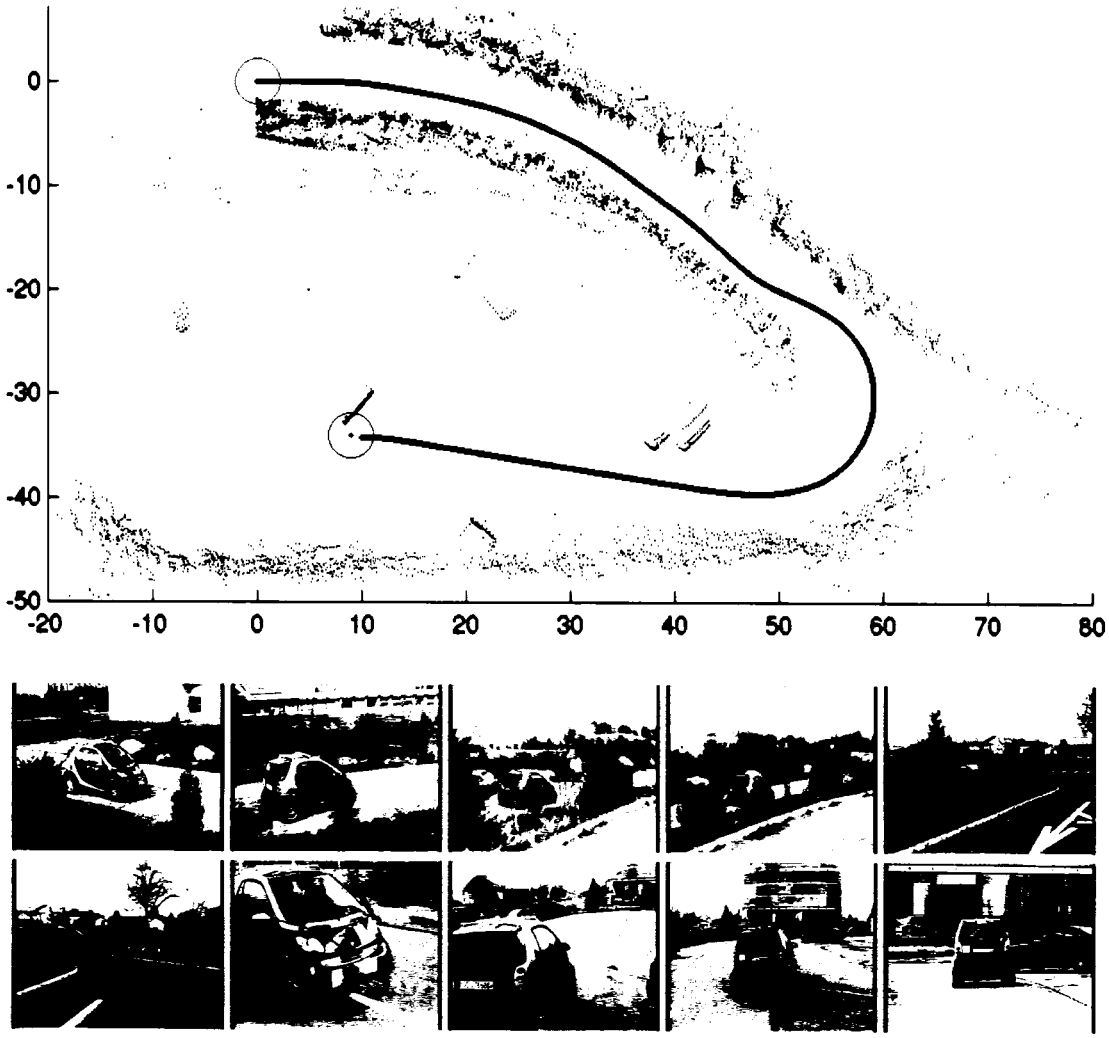


Figure 3.22: Results from an automated passenger vehicle (a SMART fortwo) performing global planning and mapping in an unstructured environment. Shown at the top is the map created from the laser during an autonomous traverse from an initial position on a rural road to a goal position inside a large parking lot. Also shown is the path (in blue) traversed by the vehicle. The vehicle began from the position marked in green at the top of the map, and navigated to the goal position marked in red at the bottom. The units on each axes are in meters. The bottom rows show snapshots from a video taken of this traverse, and are ordered from left to right, top to bottom.

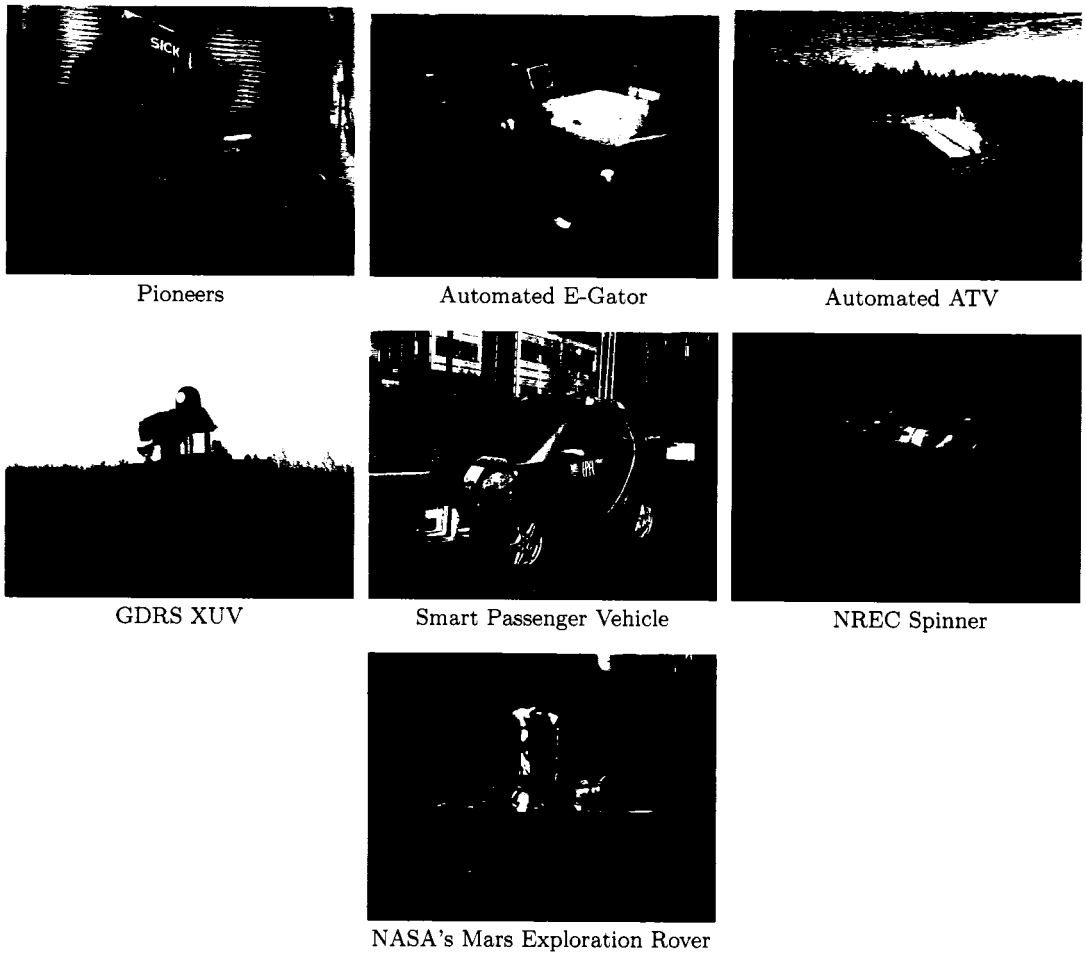


Figure 3.23: Some robots that currently use Field D* for global path planning. These range from indoor planar robots (the Pioneers) to outdoor robots able to operate in harsh terrain (the XUV and Spinner). It has also been incorporated into the onboard navigation autonomy of the Mars Exploration Rovers and was uplinked to Mars in June 2006.

Field D* generated solutions that were on average 96% as costly as those generated by D* Lite, and took 1.7 times as long to generate these solutions. During replanning, the results were similar: Field D* provided solutions on average 96% as costly and took 1.8 times as long. The average replanning runtime for Field D* on a 1.5 GHz Powerbook G4 was 0.07s. In practice, the algorithm is able to provide real-time performance for fielded systems.

Although the results presented above show that Field D* generally produces less costly paths than regular grid-based planning, this is not guaranteed. It is possible to construct pathological scenarios where the linear interpolation assumption is grossly incorrect (for instance, if there is an obstacle in the cell to the right of the center cell in Figure 3.12(i)

	D* Lite	Field D*
Initial Planning Time (s)	0.83	1.46
Initial Path Cost (relative)	1.00	0.96
Traversal Cost Update Time (s)	0.06	0.01
Replanning Time (s)	0.04	0.07
Replanned Path Cost (relative)	1.00	0.96

Table 3.1: The time and quality of solution associated with initial planning and replanning for Field D* and D* Lite. Also shown is the amount of time required to update the traversal cost of areas of the environment that have changed between replanning episodes. All values are averaged over 100 random environments with changes to the traversal cost of 10% of the environment. Reported run times are in seconds for a 1.5 GHz Powerbook G4 Processor.

and the optimal path for state s_2 travels above the obstacle and the optimal path for state s_1 travels below the obstacle). In such cases, the interpolated path cost of a point on an edge between two states may be either too low or too high. This in turn can affect the quality of the extracted solution path. However, such occurrences are very rare, and in none of our random test cases (nor any cases we have ever encountered in practice) was the path returned by Field D* more expensive than the grid-based path returned by D* Lite. In general, even in carefully-constructed pathological scenarios the path generated by Field D* is very close in cost to the optimal solution path.

Moreover, it is the ability of Field D* to plan paths with a continuous range of headings, rather than simply its lower-cost solutions, that is its true advantage over regular grid-based planners. In both uniform and non-uniform cost environments, Field D* provides direct, sensible paths for our agents to traverse.

3.4 Multi-resolution Field D*

Thus far, we have focused on algorithms appropriate for planning through uniform resolution grids. Although such grids are a common representation in mobile robotics, they can be very memory-intensive. This is because the entire environment must be represented at the highest resolution for which information is available. For instance, consider a robot navigating a large outdoor environment with a prior overhead map. The initial information contained in this map may be coarse. However, the robot may be equipped with onboard sensors that provide very accurate information about the area within some field of view of the robot. Using a uniform resolution grid-based approach, if *any* of the high-resolution information obtained from the robot's onboard sensors is to be used for planning, then the *entire environment* needs to be represented at a high-resolution, including the areas for

which only the low-resolution prior map information is available. Storing and planning over this representation can require vast amounts of memory.

In this section, we present an extension to the Field D* algorithm that is able to plan low-cost paths through non-uniform resolution grids. This algorithm allows us to represent the environment much more compactly than standard uniform grid-based approaches, saving us both computation and memory. The resulting approach effectively extends the range over which our outdoor vehicles can operate by one to two orders of magnitude.

Multi-resolution Grid Representations

A number of techniques have been devised to tackle the memory requirements of classical grid-based planning. One popular approach is to use quadtrees rather than uniform resolution grids [Samet, 1982, Kambhampati and Davis, 1986]. Quadtrees offer a compact representation by allowing large constant-cost regions of the environment to be modeled as single cells. They thus represent the environment using grids containing cells of varying sizes, known as non-uniform resolution grids or *multi-resolution* grids.

However, paths produced using quadtrees and traditional quadtree planning algorithms are again constrained to transitioning between the centers of adjacent cells and can be grossly suboptimal. More recently, framed quadtrees have been used to alleviate this problem somewhat [Chen et al., 1995, Yahja et al., 2000]. Framed quadtrees add cells of the highest resolution around the boundary of each quadtree region and allow transitions between these boundary cells. As a result, the paths produced can be much less costly, but the computation and memory required can be large due to the overhead of the representation (and in pathological cases can be significantly more than is required by a full high-resolution representation). Also, segments of the path between adjacent high-resolution cells suffer from the same limitations as classical uniform resolution grid approaches, since interpolation is not used.

As with uniform resolution grids, path planning through a multi-resolution grid is another special case of the Weighted Region Problem, and the general-purpose algorithms discussed in Section 2.1 are applicable. However, as the number of cells increases (e.g. as the agent observes information at a high-resolution through its sensors and updates its representation), these algorithms become very computationally expensive. Ideally, we would like a planning algorithm that is as efficient as traditional multi-resolution grid-based algorithms but produces better paths.

In multi-resolution grids, interpolation has the potential to be of huge benefit, since it can eliminate the requirement that paths transition between the center points of adjacent grid cells. Further, it can be used without adding any extra cells or modifications to the grid.

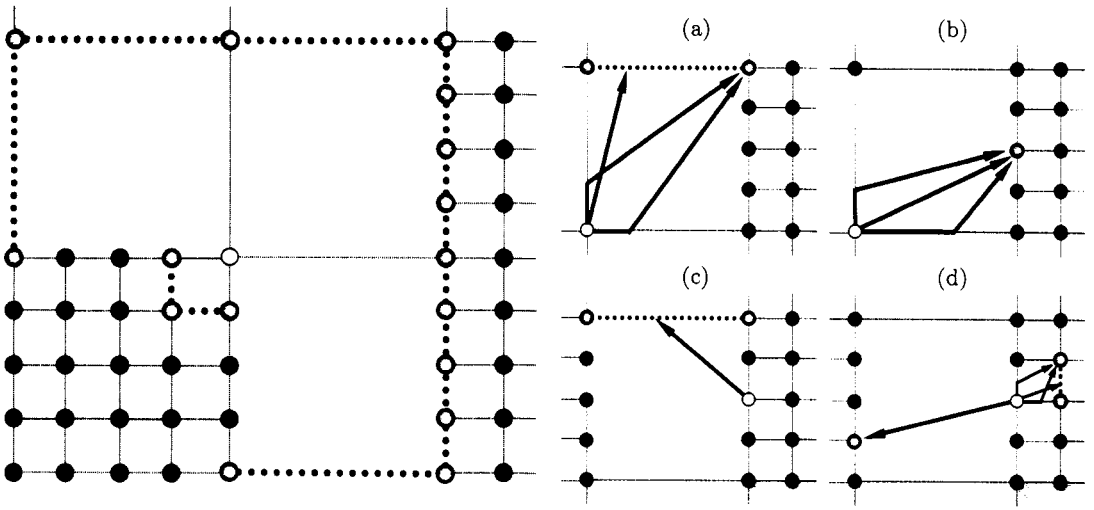


Figure 3.24: (left) The neighboring edges (dashed in red/gray, along with their endpoints in gray) from a given state (white) in a grid containing cells with two different resolutions: low-resolution and high-resolution. (a, b) Some of the possible paths to a neighboring edge/state from a low-resolution state. On the left are the possible optimal path types (in blue/dark gray) through the top low-resolution edge (dashed in red/gray) and its endpoint states (in gray). Linear interpolation is used to compute the path cost of any point along the top edge. On the right are the possible optimal path types (in blue/dark gray) to one neighboring high-resolution corner state (in gray). (c, d) Some of the possible paths (in blue/dark gray) from a high-resolution corner state (white) to a neighboring low-resolution edge (c) and to a high-resolution state (d, left) and edge (d, right).

Combining Interpolation with Multi-resolution Grids

We can use the same basic interpolation approach used by Field D* in uniform resolution grids to provide accurate path costs for states in multi-resolution grids. To begin with, we assign states to the corners of every grid cell, as in the uniform resolution case. We define the neighboring edges of a state s to be all edges that can be reached from s via a straight-line path for which s is *not* an endpoint (see Figure 3.24 (left)). We allow each state to transition to any point on any of its neighboring edges. The rationale here is that the optimal path from s must pass through one of these neighboring edges, so if we knew the optimal path cost of every point on any of these edges we could compute the optimal path cost for s.

The main difference between the uniform resolution and non-uniform resolution grid scenarios is that, in the uniform resolution case, each state s has exactly 8 neighboring edges of uniform length, while in the non-uniform case, a state may have many more neighboring edges with widely-varying lengths. However, linear interpolation can still be used to approximate the path costs of points along these edges, exactly as in the uniform resolution case.

As a concrete example of how we compute the path cost of a state in a multi-resolution grid, we now focus our attention on a grid containing cells of two different resolutions: high-resolution and low-resolution. This two-resolution case addresses the most common navigation scenario we are confronted with: a low-resolution prior map is available and the robot is equipped with high-resolution onboard sensors. Although we restrict our attention to this scenario, the approach is general and can be used with arbitrarily many different resolutions.

In a grid containing two different resolutions, each state can reside on the corner of a low-resolution cell, the corner of a high-resolution cell, and/or the edge of a low-resolution cell. Examples of each of these possibilities can be seen in Figure 3.24(b): the white state is the corner of a low-resolution cell and the gray state is the corner of a high-resolution cell *and* on the edge of a low-resolution cell. Let's look at each of these possibilities in turn.

Firstly, imagine we have a state that resides on the corner of a low-resolution cell. We can calculate the lowest-cost path from the state through this cell by looking at all the points on the boundary of this cell and computing the minimum cost path from the state using any of these points. We can approximate the cost of this path by using linear interpolation to provide the path cost of arbitrary boundary points, exactly as in uniform resolution Field D*. However, some of the boundary may be comprised of high-resolution states. In such a case, we can either use interpolation between adjacent high-resolution states and allow the path to transition to any point on an adjacent high-resolution edge, or we can restrict the path to transitioning to one of the high-resolution states. The former method provides more accurate approximations, but it is slightly more complicated and less efficient. Depending on the relative sizes of the high-resolution cells and the low-resolution cells, either of these approaches may be appropriate. For instance, if the high-resolution cells are much smaller than the low-resolution cells, then interpolating across the adjacent high-resolution edges when computing the path from a low-resolution state is not that critical, as there will be a wide range of heading angles available just from direct paths to the adjacent high-resolution states. However, if the high-resolution cells are not significantly smaller than the low-resolution cells then this interpolation becomes important, as it allows much more freedom in the range of headings available to low-resolution states adjacent to high-resolution states. In Figure 3.24 we illustrate the latter, simpler approach, where interpolation is used to compute the path cost of points on neighboring, strictly low-resolution edges (e.g. the top edge in (a)), and paths are computed to each neighboring high-resolution state (e.g. the gray state on the right edge in (b)).

For states that reside on the corner of a high-resolution cell, we can again use interpolation as presented in Section 3.3 to approximate the cost of the cheapest path through the high-resolution cell (see the paths to the right edge in (d)). Finally, for states that reside on the edge of a low-resolution cell, we can use a similar approach as in our low-resolution

```
ComputePathCost(s)
 1  g = ∞;
 2  for each cell x upon which s resides
 3    if x is a high-resolution cell
 4      for each neighboring edge e of s that is on the boundary of x
 5        g = min(g, min_{p∈P_e}(c(s,p) + v^i(p)));
 6    else
 7      for each neighboring edge e of s that is on the boundary of x
 8        if e is a low-resolution edge
 9          g = min(g, min_{p∈P_e}(c(s,p) + v^i(p)));
10        else
11          g = min(g, min_{p∈EP_e}(c(s,p) + v(p)));
12  return g;
```

Figure 3.25: **Computing the Path Cost of a state s in a Grid with Two Resolutions**

corner case. Again, we look at the boundary of the low-resolution cell and use interpolation to compute the cost of points on strictly low-resolution edges (e.g. the top edge in (d)), and for each high-resolution edge we can choose between using interpolation to compute the cost of points along the edge, or restricting the path to travel through one of the endpoints of the edge. The latter approach is illustrated for computing a path through the left edge in (d).

Thus, for each state, we look at all the cells that it resides upon as either a corner or along an edge and compute the minimum path cost through each of these cells using the above approximation technique. We then take the minimum of all of these costs and use this as the path cost of the state.

Pseudocode of this technique is presented in Figure 3.25. In this figure, P_e is the (infinite) set of all points on edge e, EP_e is a set containing the two endpoints of edge e, $v^i(p)$ is an approximation of the path cost of point p (calculated through using linear interpolation between the endpoints of the edge p resides on), $c(s,p)$ is the cost of a minimum-cost path from s to p, and $v(p)$ is the current path cost of corner point p. We say an edge e is a 'low-resolution edge' (line 8) if both the cells on either side of e are low-resolution. An efficient solution to the minimizations in lines 5 and 9 was presented in Section 3.3.

The Multi-resolution Field D* Algorithm

The path cost calculation discussed above enables us to plan direct, low-cost paths through non-uniform resolution grids. We can couple this with any standard path planning algorithm, such as Dijkstra's, A*, or D*. Because our motivation for this work is robotic path planning in unknown or partially-known environments, we have used it to extend the Field

```
UpdateSetMembership(s)
 1 if (v(s) ≠ g(s))
 2   insert/update s in OPEN with key(s);
 3 else
 4   if (s ∈ OPEN) remove s from OPEN;

ComputeInterpolatedPath()
 5 while(key(s_start) > min_{s∈OPEN}(key(s)) or v(s_start) < g(s_start))
 6   remove s with the smallest key(s) from OPEN;
 7   if (v(s) > g(s))
 8     v(s) = g(s);
 9     for each predecessor s' of s
10       if s' was never visited before
11         v(s') = g(s') = ∞;
12       if (s' ≠ s_goal)
13         g(s') = ComputePathCost(s');
14         UpdateSetMembership(s');
15   else
16     v(s) = ∞; UpdateSetMembership(s);
17     for each predecessor s' of s
18       if s' was never visited before
19         v(s') = g(s') = ∞;
20       if (s' ≠ s_goal)
21         g(s') = ComputePathCost(s');
22         UpdateSetMembership(s');
```

Figure 3.26: Multi-resolution Field D*: ComputeInterpolatedPath function.

D* algorithm to non-uniform resolution grids. To distinguish it from the uniform resolution version, we call the resulting algorithm *Multi-resolution Field D** [Ferguson and Stentz, 2006c]. By coupling the low-cost paths generated by interpolation-based planning with the memory efficiency of non-uniform resolution grid representations, Multi-resolution Field D* is able to provide extremely effective paths for a fraction of the memory and computational requirements of current approaches.

A basic version of the algorithm is presented in Figures 3.26 and 3.27. Here, the ComputePathCost function (Figure 3.26, lines 13 and 21) takes a state s and computes the minimum path cost for s using the path costs of all of its neighboring states and interpolation across its neighboring edges, as discussed in Section 3.4 and presented in Figure 3.25. Other notation and the structure of the algorithm is consistent with the Field D* algorithm presented in Section 3.3.

As with other members of the D* family of algorithms, significant optimizations can be made to this initial algorithm. In particular, several of the optimizations available to Field

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + h(s_start, s); g(s)];
 3 else
 4   return [v(s) + h(s_start, s); v(s)];
Main()
 5 g(s_start) = v(s_start) = ∞; v(s_goal) = ∞; bp(s_goal) = bp(s_start) = null;
 6 g(s_goal) = 0; OPEN = ∅;
 7 insert s_goal into OPEN with key(s_goal);
 8 forever
 9   ComputeInterpolatedPath();
10   wait for changes to grid or traversal costs;
11   for all new cells or cells with new traversal costs x
12     for each state s on an edge or corner of x
13       if s was never visited before
14         v(s) = g(s) = ∞;
15       g(s) = ComputePathCost(s);
16       UpdateSetMembership(s);
```

Figure 3.27: Multi-resolution Field D*: Main function

D* can also be applied to the multi-resolution version [Ferguson and Stentz, 2006d].

Multi-resolution Field D* Results

Multi-resolution Field D* was originally developed to extend the range over which unmanned ground vehicles (such as our outdoor vehicles in Figure 3.23) could operate by orders of magnitude. We have found the algorithm to be extremely effective at reducing both the memory and computational requirements of planning over large distances and at producing direct, low-cost paths. Figures 3.28 and 3.30 show example paths planned using the algorithm.

To quantify its performance, we ran experiments comparing Multi-resolution Field D* to uniform resolution Field D*. We used uniform resolution Field D* for comparison because it produces less costly paths than regular uniform grid-based planners and far better paths than regular non-uniform grid-based planners.

Our first set of experiments investigated both the quality of the solutions and the computation time required to produce these solutions as a function of the memory requirements of Multi-resolution Field D*. We began with a randomly generated 100×100 low-resolution environment with an agent at one side and a goal at the other. We then took some percent p of the low-resolution cells (centered around the agent) and split each into a 10×10 block of high-resolution cells. We varied the value of p from 0 percent up to 100 percent. We

Figure 3.28: Multi-resolution Field D* produces direct, low-cost paths (in blue/black) through both high-resolution and low-resolution areas. Each filled circle represents the lower-left corner state of a low-resolution cell (if the circle is large) or a high-resolution cell (if the circle is small). The detailed textures that appear in the right illustration are for visualization purposes only; the traversal cost is constant within each grid cell.

then planned an initial path to the goal. Next, we randomly changed 5% of the cells around the agent (at a low-resolution) and replanned a path to the goal. We focused the change around the robot to simulate new information being gathered in its vicinity. The results from these experiments are presented in Figure 3.29. The x-axis of each graph represents how much of the environment was represented at a high-resolution. The left graphs show how the time required for planning changes as the percent of high-resolution cells increases, while the middle and right graphs show how the path cost changes. The y-values in the middle and right graphs are path costs, relative to the path cost computed when 100% of the environment is represented at a high-resolution. The right graph shows the standard error associated with the relative path costs for smaller percentages of high-resolution cells. As can be seen from these results, modeling the environment as mostly low-resolution produces paths that are only trivially more expensive than those produced using a full high-resolution representation, for a small fraction of the memory and computational requirements.

This first experiment shows the advantage of Multi-resolution Field D* as we reduce the percentage of high-resolution cells in our representation. However, because there is some overhead in the multi-resolution implementation, we also ran uniform resolution Field D* over a uniform, high-resolution grid to compare the runtime of this algorithm with our Multi-resolution version. The results of uniform resolution Field D* have been overlaid on our runtime graphs. Although uniform resolution Field D* is more efficient than Multi-resolution Field D* when 100% of the grid is composed of high-resolution cells, it is far less efficient than Multi-resolution Field D* when less of the grid is made up of high-resolution cells.

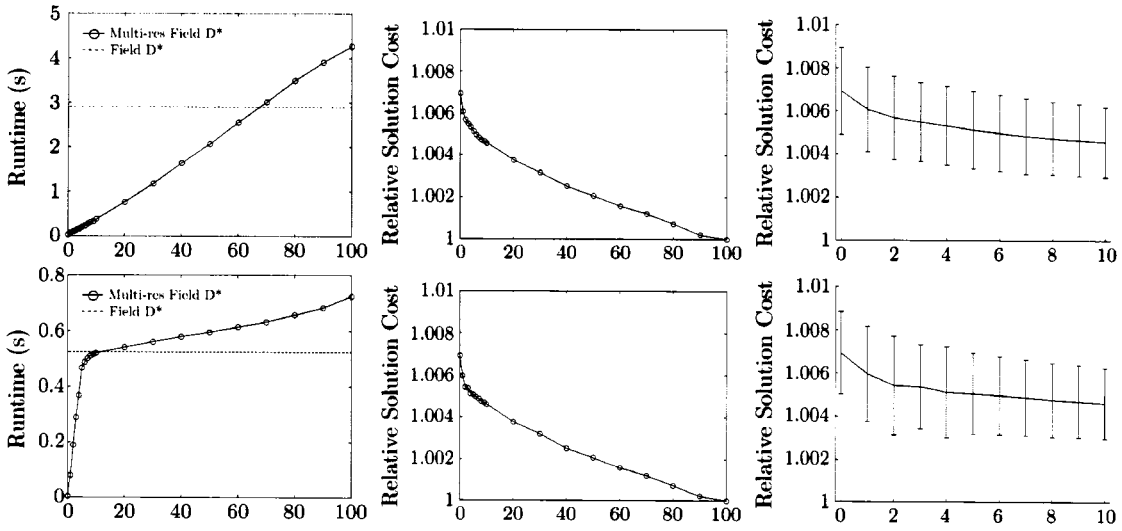


Figure 3.29: Computation time and solution cost as a function of how much of the environment is represented at a high resolution. The x-axis of each graph depicts the percentage of the map modeled using high-resolution cells, ranging from 0 (all modeled at a low resolution) to 100 (all modeled at a high resolution). (*top*) Initial planning. A path was planned from one side to the other of 100 randomly generated environments and the results were averaged. (*bottom*) Replanning. 5% of each environment was randomly altered and the initial paths were repaired.

Our second experiment simulated the most common case for outdoor mobile robot navigation, namely where we have an agent with some low-resolution prior map and high-resolution onboard sensors. For this experiment, we took real data collected from Fort Indiantown Gap, Pennsylvania, and simulated a robotic traverse from one side of this 350 × 320 meter environment to the other. We blurred the data to create a low-resolution prior map (at 10 × 10 meter accuracy) that the robot updated with a simulated medium-resolution sensor (at 1 × 1 meter accuracy with a 10 meter range) as it traversed the environment.

The results from this experiment are shown in Table 3.2. Again, Multi-resolution Field D* requires only a fraction of the memory of uniform resolution Field D*, and its runtime is very competitive. In fact, it is only in the replanning portion of this final experiment that Multi-resolution Field D* requires more computation time than uniform resolution Field D*, and this is only because the overhead of converting part of its map representation from low-resolution to high-resolution overshadows the trivial amount of processing required for replanning.

By combining the ideas of interpolation and non-uniform resolution grid representations, Multi-resolution Field D* is able to provide very cost-effective paths for a fraction of the

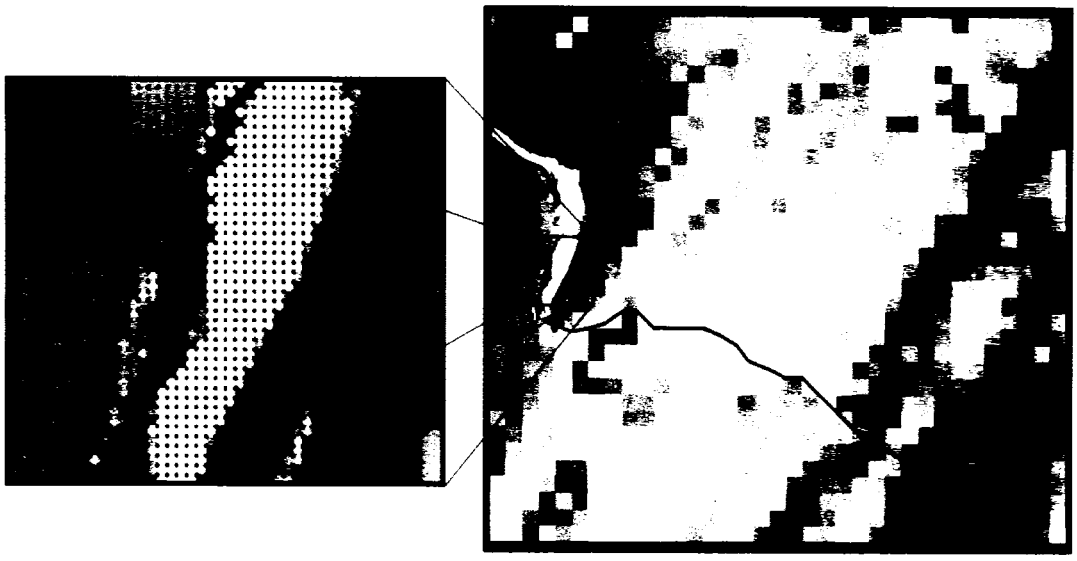


Figure 3.30: Multi-resolution Field D* used to guide an agent through a partially-known environment. On the left is a section of the path already traversed showing the high-resolution cells. This data was taken from Fort Indiantown Gap, Pennsylvania.

	Field D*	Multi-res Field D*
Total Planning and Replanning Time (s)	0.493	0.271
Initial Planning Time (s)	0.336	0.005
Average Replanning Time (s)	0.0007	0.0012
Percent high-resolution cells	100	13

Table 3.2: Results for uniform resolution Field D* versus Multi-resolution Field D* on a simulated robot traverse through real data acquired at Fort Indiantown Gap. The robot began with a low-resolution map of the area and updated this map with a high-resolution onboard sensor as it traversed the environment. The map was 350 × 320 meters in size.

memory and, often, for a fraction of the *time* required by uniform resolution grid approaches. We have found it to be very useful for extending the range over which our outdoor vehicles operate.

3.5 Planning with Uncertainty

The assumptive-based planning strategy described in the previous sections is computationally efficient and works well in many situations, particularly when errors in the initial estimate of the environment do not have significant consequences. However, there are scenarios where this strategy can lead to highly suboptimal results. For example, consider a

robot navigating outdoors equipped with a low resolution overhead map of the surrounding area generated by a helicopter. Due to the low resolution of this map, there may be some uncertainty as to the terrain of certain areas. If some of these areas reside in narrow passageways, whether they are traversable or not may have a large impact on the length of the robot's path. Thus, incorporating the uncertainty associated with the terrain of these areas into the planning process allows the robot to make more informed decisions when selecting its path.

In examples like these, the navigating agent knows that it has incomplete or uncertain information. Moreover, it is often also aware of the *nature* of this uncertainty. In our example, the robot may know the density of the data used to produce the original map, as well as the uncertainty associated with the helicopter position for each data point. It would also have an idea of the error associated with any footprint convolution process used to take the overhead map and extract a terrain map for planning.

The combination of these sources of information can be used to derive error models for the final planning map. Given these error models, the robot can determine a reasonable approximation of the probability that a given cell in its planning grid holds a particular terrain value. In other words, by paying close attention to the uncertainty associated with its information, an agent can derive distributions over the possible terrain values of a given area of the environment, which it can then use to produce more robust paths to traverse.

Current Approaches

Dealing with distributions over terrains rather than fixed values requires some modification to classical planning techniques. As discussed earlier, one popular approach for dealing with uncertainty is to perform assumptive planning and just use an approximate, single terrain value for each cell. However, this can lead to suboptimal results. Typically, assumptive planners solve for a path from the start state s_{start} to a goal state s_{goal} while minimizing the overall cost of the path. As mentioned earlier, the minimum cost of a path from state s to the goal s_{goal} is defined as

$$\min_{s' \in Succ(s)} (c(s, s') + v(s')).$$

The value $c(s, s')$ is computed from the approximate cost of traversing from state s to neighboring state s'. But if there is some non-zero probability that the edge between s and s' is untraversable, the equation above is no longer valid, as it does not account for this possibility and its associated cost. In order to get the appropriate cost, an alternative path needs to be planned *around* the untraversable edge to the goal. The cost of this path is nontrivial to compute, as it requires making similar considerations for all other edges encountered, which causes the overall computation to be exponential in the number

of edges in the graph. As a result, accurate cost approximations are difficult to generate and computed solution paths based on rough estimates can be highly suboptimal.

Instead, it is possible to deal with the terrain uncertainty more comprehensively by incorporating this uncertainty into the planning task. Partially Observable Markov Decision Processes (POMDPs) are a representation that explicitly encode the uncertainty associated with state observability and action outcomes [Astrom, 1965, Sondik, 1971, Monahan, 1982, Kaelbling et al., 1998]. They have been used to represent a number of application problems in robotics, ranging from dialog management [Pineau et al., 2003b] to robotic hide and seek [Roy et al., 2005]. Formally, a POMDP is a tuple $< S, A, O, R, T, P_0, P >$, where S is a finite set of states, A is a finite set of actions, O is a finite set of observations, $R : S \times A \rightarrow \mathcal{R}$ is the reward function, $T : S \times A \times S \rightarrow \mathcal{R}$ is the state transition function (where $T(s, a, s')$ denotes the probability of transitioning to state s' when performing action a in state s), $P_0 : S \rightarrow \mathcal{P}$ is the probability distribution over initial states, and $P : S \times O \rightarrow \mathcal{R}$ denotes the probability of observing a particular observation in a particular state. Because of the uncertainty associated with observations and actions, in POMDPs the actual state of the world is rarely perfectly-known. Thus, planning takes place in *information space*, which consists of all beliefs an agent may have regarding the actual state of the world. These beliefs take the form of probability distributions over the states (as with P_0).

Our prior map uncertainty can be encoded as a special case POMDP, known as a Deterministic Decision Problem with Hidden State [Ferguson et al., 2004]. In this case, we can encode the map uncertainty into the initial probability distribution P_0, and then plan in information space to generate an optimal solution that trades off information gathering (to refine the agent's knowledge) with purely goal-directed actions.

Unfortunately, POMDP planning is very computationally expensive [Papadimitriou and Tsitsiklis, 1987]. Various good approximation techniques have recently been developed [Pineau et al., 2003a, Smith and Simmons, 2004, Roy et al., 2005, Spaan and Vlassis, 2005], but even these struggle to solve POMDPs for which the size of the state space is much larger than 10000. Even under the simplest instance of our current problem, where the environment is represented as a traversability grid and each uncertain grid cell can hold only one of two possible terrain values (traversable or untraversable), we have a state space of $m \cdot n \cdot 2^p$, where the map is $m \times n$ cells in size and there are p uncertain cells in the map. For realistic values of m, n, and p, the corresponding state space is much too large for current POMDP planning algorithms.

Limitations

As just mentioned, incorporating the terrain uncertainty into a large POMDP and then solving this POMDP is usually infeasable for realistic robot navigation problems. However,

ignoring the uncertainty associated with the terrain and using the assumptive planning approach described earlier in this chapter can be grossly suboptimal in certain cases.

To address the limitations of each of these techniques, we would ideally like an approach that *(a)* determines which areas of uncertainty are most crucial to the planning task, and *(b)* efficiently incorporates the uncertainty associated with just these areas into the planning task.

To produce robust paths in a tractable manner given uncertain a priori information, we have developed an approach that focuses computation on areas of the environment whose traversability, and hence uncertainty, is crucial to the planning task. Thus, rather than dealing comprehensively with the uncertainty associated with every area of the environment, as POMDP planning does, we restrict our attention to those areas that are most important. We then incorporate the uncertainty associated with these areas, coined *pinch points* [Ferguson et al., 2004], into the planning process to produce more effective paths for our agents to traverse. The intuition behind this approach is to reduce the size of the state and information spaces so that solutions can be generated, and to perform this reduction in a way that ensures the generated solutions are of high quality. The following two sections detail the approach and provide examples and results.

3.6 Pinch Point Extraction

The idea behind pinch point extraction is to focus on the most important areas of uncertainty in the environment. These are the ones that would cause a costly detour in the robot's path to the goal if they turned out to be untraversable. We would like to be able to detect these areas and incorporate their terrain uncertainty in our planner so that we could decide from the outset whether it is better to avoid these cells entirely or risk going through them.

To find these areas in a grid-based representation of our environment, we generate a set of grid cells that could reasonably be encountered by an agent navigating to the goal and look for key members of this set [Ferguson and Stentz, 2004b]. Figure 3.31 outlines the process.

We first assume each cell with a reasonable probability of being traversable is in fact traversable and generate its resulting expected terrain cost[6]. We then plan a path over the resulting traversability grid by treating the center of each grid cell as a state and allowing transitions between states in adjacent grid cells, exactly as described in Section 2.1. Given the grid cells encountered by this path, labelled as consecutive cells $r_1 \ldots r_n$, we can find

[6]This is computed by normalising its terrain distribution to only contain the traversable range then taking an expectation.

1. Initially, set the cost of each cell in the environment to its expected traversable cost. Use D* to plan an optimal path (relative to these costs) from the robot position to the goal. Mark the pinch points along this path (see 2(b)) and use them to construct an ordered list P.
2. While there are still pinch points in P:
   (a) *Replan path from pinch point to goal*: Remove the top pinch point from P and set the terrain of each cell belonging to a pinch point to *untraversable*. Invoke D* to replan a path to the goal from the cell previous to the pinch point (on the path on which the pinch point was found).
   (b) *Mark new pinch points along path*: Step along this new path and look for sections that have a high probability of being untraversable *and* are costly to navigate around. Mark these cells as pinch points and add them to the end of P.
3. Return all pinch points encountered.

Figure 3.31: The Pinch Point Extraction Algorithm

all sections of the path that are potential blockages *and* would cause significant detours if found to be blocked.

One approximation method for finding such blockages is as follows. Label a section of the path centered on cell r_i a potential blockage if there is a nontrivial probability that an agent at cell r_{i-1} will not be able to get to cell r_{i+1} using one of the three shortest non-intersecting routes between the two cells. Here, the idea is that if r_i turns out to be untraversable, there is still a good chance r_{i+1} is reachable from r_{i-1} using one or more of the neighbors of r_i, but if r_i is untraversable and the next obvious two paths from r_{i-1} to r_{i+1} are also untraversable, then it is not so likely. To generate the probability that the path is blocked at cell r_i, we look at the three shortest paths from cell r_{i-1} to cell r_{i+1} and use the terrain distributions of the cells along these paths to determine the probability that all three paths are untraversable. If this probability is greater than some threshold, we group cell r_i and the cells along the shortest paths together as a potential pinch point with position r_i.

Figure 3.32 illustrates the shortest paths between r_{i-1} and r_{i+1} for four different relative positions of the consecutive path cells r_{i-1}, r_i, and r_{i+1}. The paths for all other possible relative positionings can be obtained from these four. In this figure, r_i provides one path between the two cells (shown in blue), while the other paths travel through the cells marked a_1, a_2 and b_1, b_2, respectively.

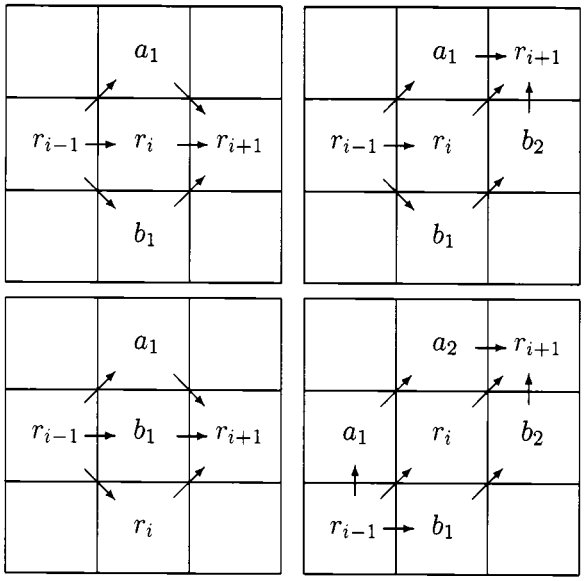


Figure 3.32: Finding potential path blockages in an eight-connected grid. Each diagram highlights three local paths from cell r_{i-1} to r_{i+1} (both shown in red) for different relative positions of cells r_{i-1}, r_i, and r_{i+1}. If there is a nontrivial probability that all of these three paths are untraversable, the cells along the paths are grouped together as a potential pinch point.

If a group of cells is marked as a potential pinch point, we know that there is a nontrivial probability that an agent may not be able to get through this area. In order to determine the consequences of this possible outcome, we then check how costly a route around the potential pinch point would be. To do this, we generate a least-cost path from the previous cell on the path, r_{i-1}, to the next cell on the path, r_{i+1}, without using any of the cells constituting the potential pinch point. If the cost of this path is significant we add the potential pinch point to our list of true pinch points.

If we come across a number of pinch points in a row, for instance in a narrow valley, we combine them into a single pinch point. The probability of the combined pinch point being blocked is taken to be $1-p$, where p is the probability that all of the pinch points are traversable. The position of the pinch point used later for planning is taken as the mean of its constituents.

Once we have found all pinch points along the original path, we then set the terrains of all cells comprising these pinch points to untraversable and replan paths from the close side of each of these pinch points to the goal, i.e., from r_{i-1} for each pinch point r_i. This enables us to locate new pinch points that might be encountered by the agent if it found the current pinch point to be untraversable. This replanning can be efficiently performed using any of the algorithms described in Section 3.1, such as D*. We iterate the procedure

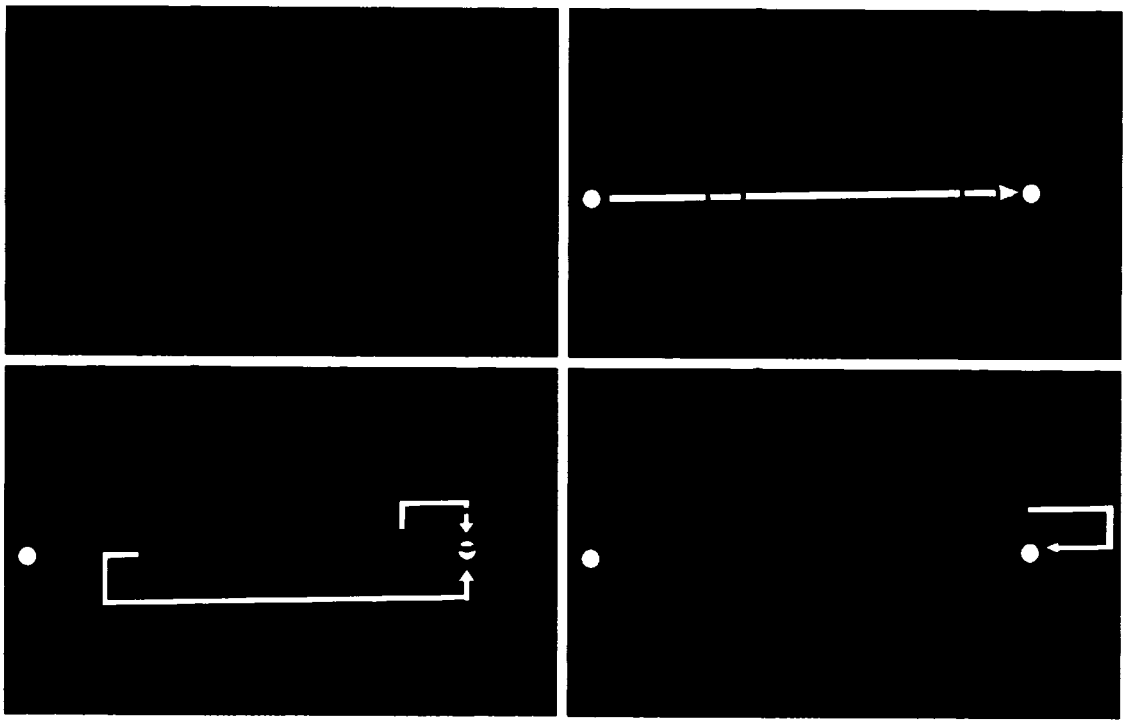


Figure 3.33: (*top-left*) A helicopter-generated map of an outdoor area. Dark areas represent trees or grass, light areas represent roads or buildings. Registered with this visual imagery is 3D terrain information. Black areas represent sections of the environment for which no information was gathered. (*top-right*) A path (in white) is planned from the white circle on the left side to the white circle on the right side of this map, assuming areas with a reasonable probability of being traversable are traversable. The two hollow black circles represent the pinch points found along this path. (*lower-left*) Alternative paths from the start-side of each pinch point to the goal. These paths are computed assuming the two pinch points from (*top-right*) are untraversable (denoted as black filled-in circles). A third pinch point is found along the alternative path from the second pinch point. (*lower-right*) The alternative path from the third pinch point. Since this path does not contain any further pinch points, the three black filled-in circles represent all the pinch points used for planning. This outdoor data was obtained by Omead Amidi and Ryan Miller.

to generate a set of pinch points that could reasonably be encountered by an agent moving towards the goal (assuming the agent always acts optimally given its map information). Figure 3.33 illustrates the approach in action.

Pinch Point Extraction Results

To test the computation required to extract pinch points in environments of varying complexity, we generated 1000 fractal terrain grid maps, each of size 200×200. These envi-

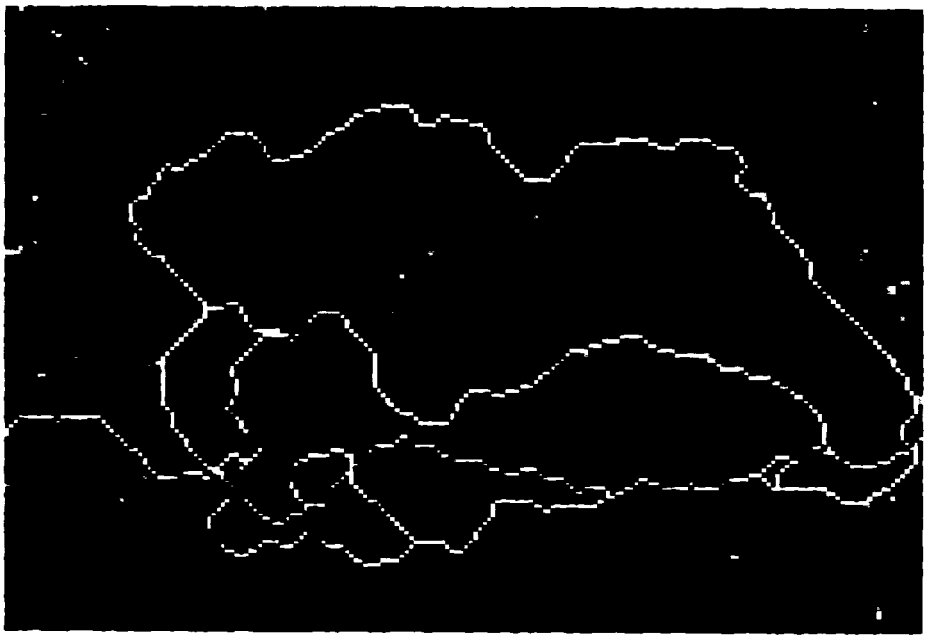


Figure 3.34: An example fractal terrain grid map used for testing, where the darker the cell, the higher the probability it is untraversable. Shown in yellow is the initial path planned from a point on the left side of the environment to a point on the right. The pinch points extracted are shown in red, with the set of alternative paths shown in white.

ronments ranged from very open, easy to navigate areas to very complex, cluttered areas[7]. With each map, we limited the total number of pinch points[8] extracted to 10. As Figure 3.34 shows, this still enables us to consider a large number of diverse paths when the environment is highly cluttered.

Over our 1000 terrain test cases, the average number of extracted pinch points was 2.47 and the average CPU time taken for this extraction was 0.07 seconds when run on a P3 1.4 GHz processor. The maximum amount of time required to extract the pinch points from any map was 0.34 seconds (with the minimum being 0.003).

3.7 Pinch Point Planning: PAO*

Once equipped with the set of pinch points, we can solve the reduced POMDP to compute the optimal grid-based path with respect to the uncertainty of these key areas. To do this,

[7]For details of the fractal generation process, see [Stentz, 1995]. We used a gain of 20 and varied the number of levels from 5 to 9.

[8]By restricting the total number of pinch points extracted, we keep the subsequent planning problem tractable, as we will discuss in the next section.

we first represent the environment as an adjacency graph containing the robot position, the goal position, and the pinch points in the environment.

Each pinch point may provide a bridge between several different regions of the environment. For instance, a pinch point located at a Y-junction connects three different regions to each other. The collection of cells adjacent to the pinch point in each region constitute a *face* of the pinch point. The adjacency graph links up these faces by inserting edges between every pair of faces that are directly reachable from one another (i.e. without having to travel through any other face). The cost of an edge between two faces represents the lowest cost associated with moving along a pinch-point free path between the faces and is used to propagate values from one face to another.

A shortest path is then planned from the robot position to the goal position using this adjacency graph. To do this, the problem is phrased as a search over an AND-OR graph [Rich and Knight, 1992]. An AND-OR graph contains two types of states: AND states obtain their values from combining the values of all their child states, while OR states compute their values from choosing a single child state value. Our planning domain can be represented as an AND-OR graph as follows. Each state in the graph corresponds to a particular location in a particular *information state*. In our setting, an information state is the state of knowledge the agent may have concerning the terrain values of each of the pinch points. Each cell may be known to be traversable (t), known to be untraversable (o), or not yet seen by the agent (u), in which case it has its initial probability distribution over being traversable/untraversable. This results in 3 different states of information the agent may have concerning the terrain of each cell.

The root of the AND-OR graph (an OR state) is the start cell s_{start} in the information state characterised by every pinch point being as yet unseen (i.e., of value u). The next level of the graph corresponds to all elements of the adjacency graph, both faces and goal, that have edges to s_{start}. The faces are AND states: each has two children representing the two possible information states realizable from visiting the face and observing the value of its associated pinch point. These two children each have the same environmental location as their parent but reside in different information states (one has the pinch point of value t, the other o). These children are OR states because their associated pinch point has a known value. See Figure 3.35 for an example AND-OR solution graph for this problem.

Intuitively, from a state s the agent can choose to move to any adjacent face or directly to the goal (if clear). Thus, its cost is a function of the *minimum* cost of the adjacent faces. Once it has moved to one of these faces, it observes the true value of the associated pinch point[9]. It does not choose this value: it is taken from the range of possibilities (in our case

[9]We assume the agent is equipped with a proximity sensor that will tell it, when it arrives at a pinch point, whether the pinch point is traversable or not.

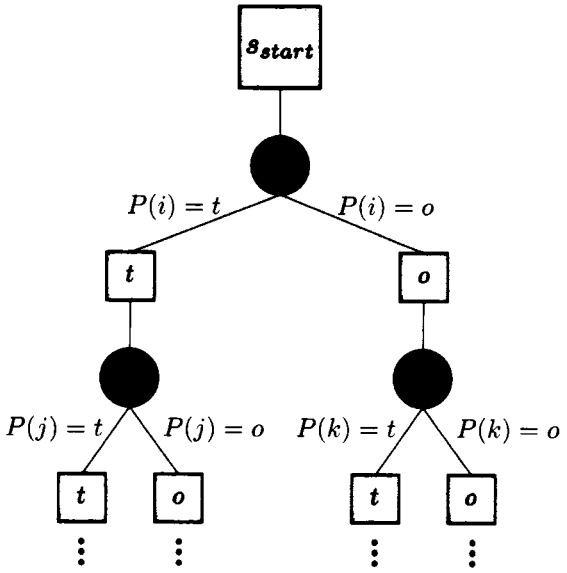


Figure 3.35: A partial solution AND-OR graph for the pinch point planning problem. Each circle corresponds to an AND state and each square to an OR state. s_{start} refers to the initial agent position, f_i refers to face i, and $P(i) = o$ represents the probability that the pinch point associated with face i is untraversable.

just {*traversable, obstacle*}) according to the pinch point's probability measure. Thus, the cost of the parent state is a combination of the cost of *both* its children.

Once equipped with this AND-OR graph representation, our planning problem is equivalent to the graph-theoretic Canadian Traveller's Problem (CTP), which consists of planning a route through a graph where some edges may turn out to be untraversable [Bar-Noy and Schieber, 1991]. Because there is some uncertainty associated with the traversability of these edges, solving the planning problem consists of generating a solution *tree*, rather than a solution path. This tree specifies the best action to take from any location and information state that could be encountered by the agent when executing the solution. For instance, if a particular solution involves the agent traveling to a particular pinch point, then it must provide contingency plans for both the case that the pinch point turns out to be traversable and the case that it turns out to be untraversable. We describe three basic approaches for computing optimal solutions to this problem.

The Complete Solution

The first approach is to compute the estimated cost-to-goal value of every state in the graph in every information state. Ultimately, we are trying to compute an optimal path for an agent that starts out in the information state $H = \{u, u, \ldots, u\}$. However, the cost values of each face in our graph at this state can be recovered directly from the costs of the faces

ComputeCost$(C[f_k, i])$

1 Cost = $Cost(f_k, Goal)$;
2 $v = v(h(f_k))$;
3 if $(v = u)$
4 Cost = $p(h(f_k) = o) \cdot C[f_k, i_o] + p(h(f_k) = t) \cdot C[f_k, i_t]$;
5 else
6 Cost = $\min(\text{Cost}, \min_{f_i \in f}(C[f_i, i] + Cost(f_k, f_i)))$;
7 if $(v = t)$
8 Cost = $\min(\text{Cost}, \min_{f_i \in a(f_k)}(C[f_i, i] + CostThru(f_k, f_i)))$;
9 return Cost;

Figure 3.36: The Cost Update Calculation for Face f_k in Information State i.

in the information states which have *exactly one* pinch point of known value. These face costs in turn can be computed from the costs of the faces in the information states which have *two* known pinch points, and so on.

The reason for this is as follows. As soon as the agent moves to a face associated with a pinch point which is of value u, the agent observes what the actual value of that pinch point is. As a result, our agent is constantly increasing its knowledge of the state of the environment, one pinch point at a time. To solve for the values of information state i we must have the values of every information state that is reachable from i. These are exactly the information states that have one more pinch point of known value.

In short, we iterate from the base-case information states where the environment is completely known (all pinch points are of known values) up to information states with increasing numbers of pinch points holding the value u, and finally to the initial robot location in the initial information state (where all pinch points hold the value u).

The costs of each face in the deterministic information states (there are 2^p such states for p pinch points) can be solved for using Dijkstra's search[10]. Each face has its cost initialized to its edge cost to the goal. If no such edge exists (i.e. the face has no path to the goal without needing to traverse some hidden state element), the cost is initialized to infinity.

Once the costs of these states have been determined, the costs of faces of subsequent information states can be solved with value iteration [Bellman, 1957, Bertsekas, 1987], using the cost update (or Bellman backup) shown in Figure 3.36.

In this cost calculation, $C[f_k, i]$ represents the cost of face f_k in information state i, $h(f_k)$ is the pinch point to which face f_k belongs, $v(h_j)$ is the value of pinch point h_j in information state i (one of t, o, or u), i_o and i_t are the information states similar to i in all respects except that $h(f_k)$ is of value o and t, respectively, and $a(f_k)$ is the set of all faces associated with pinch point $h(f_k)$.

[10]If we were only interested in the value of one face it would make sense to use A* rather than Dijkstra's.

The cost update works by finding the complete set of successor faces (combined with information states) from a given face f_k. If f_k is attached to a pinch point with value u (in our information state i), then we use the probability measure associated with this pinch point to generate an expected cost of the current face. This expected cost combines the values of the face in the information states i_o and i_t. If the pinch point is known, then we update our current cost to be the minimum of the cost associated with moving to any adjacent face (and the goal, if reachable). If the pinch point is known *and is traversable* then we can add to our contention the faces on the other side of the current pinch point as these, too, are available successors.

Reachability Analysis

A major drawback of the above approach is that *every* possible information state is examined and has its cost solved for, including states that can never be realized given the initial state the agent resides in.

Consider again our outdoor robot navigation scenario. If there are a number of pinch points in the environment that the robot cannot directly reach (i.e., without going through some other pinch point), it does not make sense for it to process any information states where it holds information concerning these pinch points without knowing the values of the pinch points it would have to pass through to get this information. Such states cannot possibly be encountered given the initial position and information state of the robot.

Reachability analysis has been used extensively by the Markov Decision Processes community (and others) to restrict computation to information states that are physically reachable from the initial state [Boutilier et al., 1998, Blum and Furst, 1995]. The idea is to propagate outwards from the initial state, marking each encountered state as reachable. All states left unmarked can be ignored in our solution derivation.

Incorporating reachability considerations, the algorithm described above changes in two ways. Firstly, an initial propagation step is performed, branching out from the initial state, to mark all the reachable states. Secondly, the iteration phase only considers the states marked in the first step, thus ignoring the irrelevant areas of our information space.

The AO* Algorithm

The number of examined states can be further reduced by performing heuristic-based search over the information space. AO* is a classic search algorithm that performs such a heuristic search over an AND-OR graph [Chang and Slagle, 1971, Rich and Knight, 1992]. AO* searches an AND-OR graph by gradually building a solution tree from the start state through two alternating phases. First, it grows the best partial solution by expanding one of the non-terminal leaf states in its search tree and assigning admissible heuristic costs to

1. The initial solution tree consists solely of the start state, s_{start}, in the original information state H.

2. While the solution tree has some nonterminal leaf state:
   (a) *Expand best partial solution*: Expand a nonterminal leaf state and compute heuristic values for its two children. Add the children to the solution tree, noting whether they are nonterminal.
   (b) *Propagate cost changes and update solution*: Compute an updated cost estimate of the original leaf state given the costs of its children. If its cost has changed, update its parent's cost to reflect this change. If the parent is an OR state, the current child may be replaced if it no longer provides the minimum cost. Continue propagating up the tree until a state is reached whose cost does not change.

3. Return the optimal solution tree.

Figure 3.37: The AO* Algorithm

its children[11]. Then it uses the newly computed costs to propagate cost revisions to the parent state and onwards up the tree. The full algorithm as used is illustrated in Figure 3.37.

The efficiency of AO* is obtained through its use of a heuristic to limit the amount of the AND-OR graph that is examined. The resulting solution tree can often be constructed through observing only a fraction of the complete graph. In our case, the heuristic value of a new state n is computed through solving for the cost of the 'heuristic counterpart' of n: the fully-known state characterized by the best-possible true values the pinch points in n could have. For elements already known in n (i.e. pinch points h_p such that $v(h_p) = t$ or $v(h_p) = o$) the elements are left untouched. Elements still unknown (with $v(h_p) = u$) are assigned the value t. The resulting values are guaranteed to be admissible and computing these values can be done very efficiently.

The PAO* Algorithm

AO* works very well in certain situations, particularly when most of the reachable states are clearly undesirable. This is because its use of a heuristic allows it to focus its search away from highly suboptimal faces. However, given the current problem domain and heuristic, it is possible for the algorithm to examine far more states than necessary. Furthermore,

[11] A state is non-terminal if it is not the goal location and it can transition to a new information state by moving to a neighboring state in the adjacency graph.

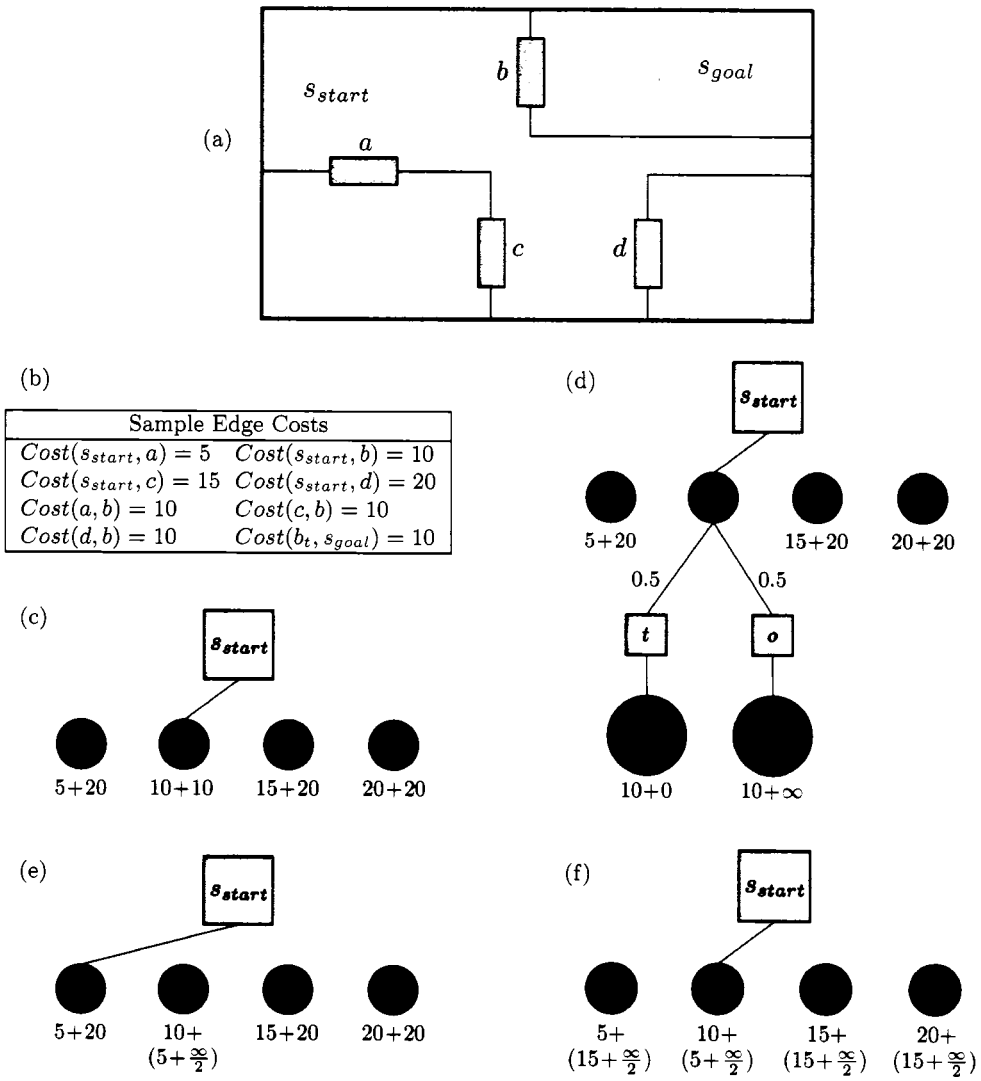


Figure 3.38: Sample planning problem involving a robot, a goal, and four doors which may be open or closed. Some sample edge costs between door faces and points of interest are shown in (b). The first two expansions of the AO* and PAO* solution trees are shown in (c) and (d). The values under each state represent the parent's cost to the state plus the cost of the state itself. The resulting AO* and PAO* solution trees after propagating the cost change from (d) are shown in (e) and (f).

because the partial solution tree is altered during the course of the algorithm, AO* can re-expand the same state several times. In the worst case, this can result in execution times that surpass the naïve approach by orders of magnitude [Ferguson et al., 2004].

To minimize the drawbacks of using partial solution trees while capturing the clear benefits of using heuristic-based search, we have developed the *Propagating AO** (PAO*) algorithm [Ferguson et al., 2004]. The key insight behind PAO* is that cost changes are

rarely isolated. PAO* propagates the effect of cost changes not only upwards to parents in the partial solution tree, but sideways to neighbors (in the complete AND-OR graph), and downwards to children. The resulting approach makes full use of all received information and thus allows for more informed decisions to be made at each stage of the planning process.

The key insight behind PAO* is that cost changes are rarely isolated. If a state updates its cost based on an altered child cost, it is likely this update will affect the costs associated with that state's neighbors. Consider the simple scenario described in Figure 3.38, where a robot (at the position s_{start}) must navigate to a goal within an environment containing 4 doors. Each door may be open or closed and the robot is equipped with a proximity sensor to tell, upon reaching a doorway, which possibility prevails. For clarity we have only dealt with the four reachable faces from the robot's initial position and have shown only relevant edge costs. We have slightly abused notation and used $Cost(b_t, s_{goal})$ to express the cost between face b and the goal (assuming there is no door at b).

Initially, all the faces have heuristic cost values associated with them, corresponding to their lowest possible costs. In this particular scenario, all these costs correspond to paths through face b. As explained previously, these heuristic values are the result of a value iteration over the state of the environment where all the doors are open. Given these initial face costs, the best successor from R is b, giving s_{start} an initial cost of $C[s_{start}] = Cost(s_{start}, b) + C[b] = 20$. So b is placed as the child of R in the partial solution graph (see Figure 3.38(c)).

After a single further expansion of the tree, the cost associated with b changes dramatically. Its two possible outcomes are computed and its resulting cost is forced to reflect the possibility that the adjacent doorway may be blocked, in which case no path to the goal is possible. However, AO* only uses this new information to update the value of the state at the root of the partial solution tree, which in turn chooses a new child (one of the faces whose cost is the original heuristic value - see Figure 3.38(e)). This does not make effective use of the information gained in the previous expansion. Because all the faces depend on b to reach the goal, their costs are affected by any cost changes associated with b. By ignoring this, AO* ends up expanding each of the faces reachable from R one by one in order to arrive at the same cost values that could have been computed directly from this initial expansion.

PAO* propagates information concerning updated costs more thoroughly through the information state space. The complete algorithm is given in Figure 3.39. There are four key differences between its operation and that of AO*.

The first difference allows PAO* to overcome the difficulty AO* faces in domains such as our simple robot navigation example of Figure 3.38. In its propagation of cost changes, PAO* propagates the updated child cost through the *entire* child information state, so that

1. The initial solution tree consists solely of the start state, s_{start}, in the original information state H.

2. While the solution tree has some nonterminal leaf state:

   (a) *Generate fringe state*: Starting from the root, traverse down the solution tree until a nonterminal leaf state is encountered. Along the way, update untraversable child states to have their face costs lower bounded by their parent states.

   (b) *Expand best partial solution*: Expand the nonterminal leaf state and compute cost values for the information states of its children. Traversable child states are given heuristic costs. Untraversable child states inherit their parents' cost values as lower bounds then perform limited value iterations over their heuristic counterparts to potentially increase these values. Add the children to the solution tree, noting whether they are terminal.

   (c) *Propagate cost changes and update solution*: Compute an updated cost of the original leaf state given the costs of its children. If the state's cost has changed, update the cost estimates for its *entire information state* and update its parent's cost to reflect these changes. If the parent is an OR state, the current state may be replaced if it no longer provides the minimum cost. If the state is a traversable child, update the costs associated with the entire parent state to be lower bounded by the current state. Continue propagating up the tree until a state is reached whose cost does not change.

3. Return the optimal solution graph.

Figure 3.39: The PAO* algorithm

dependencies between faces will be reflected in their costs: if one face in an information state uses another to derive its cost, then it makes sense that when the cost of the latter state changes, the cost of the former may be affected. PAO* updates the costs of all such affected states. As a result, the parent of the updated child will be able to use the most accurate information possible in determining its own cost and (currently) optimal child (see Figure 3.38(f)).

Secondly, PAO* propagates cost values *down* the solution tree. Given an AND state with two children corresponding to the two possible true values of the state's pinch point (traversable and obstacle), the cost of the parent state should never be greater than the cost of the obstacle child state. Similarly, the parent should never have a lower cost than the traversable child. This makes intuitive sense: if the true value of a given pinch point is known to be traversable, then we are certainly in at least as desirable a state as if we did not know anything about that pinch point's true value. However, because a face in a given

information state can be reached through a number of different paths, often this will not hold for a given parent and child combination. It is even more likely that *other* faces in the same information state as the face of the child/parent state will have unrealistic costs. To take advantage of this piece of intuition, PAO* updates the face costs of the states associated with obstacle states so that they are lower bounded by their parent state values. The update looks at each face in the child state and assigns it the maximum of the costs assigned to it by the two respective states, parent and child. PAO* performs this update as it traverses down the solution tree to select the next nonterminal state for expansion.

As an example, consider again the simple environment given in Figure 3.38(a). Assume this time the robot starts not from the position marked s_{start} but rather in the room blocked by the doors attached to faces a and c. Let's assume further that the robot initially expands the face on the other side of the door from a (call this face a'). If it then chooses to expand the *traversable* child of a', it will receive an updated value for face b that takes into account its probability of being untraversable and precluding any solution. When it propagates this information back to its parent and on up to state a', suddenly the traversable child of a' has a higher cost than its obstacle child, since the obstacle child still uses the heuristic cost of each unexpanded face (including b). PAO* propagates the updated information to the obstacle child on its next pass down the tree and, as a result, arrives at a much better heuristic estimate of its cost.

The relationship works both ways, and PAO* also updates the face costs of *parent* information states from their *traversable* child states. It performs this update as part of its propagation of cost changes back up the solution tree. These two propagation steps combine to allow information gained at one end of the solution tree to be accessible at the other.

The final difference resides at the state expansion stage. In the AO* expansion of a nonterminal state, indifference is shown towards the nature of the two children. Both are assigned initial heuristic values and these values are then used to update the parent. However, it is possible to exploit the relationship between the face costs of parent and child states described above to produce more realistic values for at least one of the two children. PAO* allows the obstacle child state to inherit the values of the parent state, then performs value iteration over the information state characterized by the heuristic counterpart of the obstacle state. While performing this value iteration, it carefully ensures the resulting costs are not less than the parent costs. This is done by initializing each face with the cost of its edge to the goal (if one exists), then performing standard value iteration. If, at any point during value iteration, the cost of a particular face becomes less than that face's cost in the parent state, the face has its cost fixed to the parent cost for the remainder of the iterations. This inheritance allows all the information concerning the parent state to be retained and utilised by the obstacle child state.

Approach	Complete	Reachability	AO*	PAO*
Examined				
Min	59048	19684	5	5
Max	59048	59048	59048	2146
Avg	59048	52924.8	25272.0	405.8
Expanded				
Min			2	2
Max			26748382	3150
Avg			5794832.5	314.8
Run Time				
Min	1.3278	0.4749	0.0004	0.0002
Max	2.8071	2.5056	1011.5316	0.2827
Avg	1.8250	1.3594	232.5929	0.0302

Table 3.3: The results of our four approaches applied to a test set of 200 environments containing 10 pinch points each. The three criterion displayed are the number of information states examined, the number of states expanded (in the case of AO* and PAO*) and the run time required to find the optimal solution.

Together, these propagation steps combine to allow information gained at one end of the solution tree to be taken advantage of at the other. As a result of these differences, PAO* can be much more efficient than AO* at generating a solution. Further, since these modifications never overestimate the costs of states, the optimality of PAO* follows directly from the optimality of AO*.

PAO* Results

We compared the performance of PAO* to all three alternative approaches discussed above. The algorithms were tested over 20 different fractally generated environments, each containing 10 pinch points. Each fractal environment was generated using a different density to simulate varying degrees of terrain difficulty. For each environment we randomly varied the probabilities associated with each pinch point to produce 10 different test cases.

In each case, the task was to find the optimal path given a start state at one end of the environment and a goal state at the other. Each environment was 200×200 cells in size. The time taken for the initial edge cost propagation is independent of which approach is used and is highly dependent on the size of the environment, so it has been left out of our comparison. On average, this propagation took about 6 seconds. All times reported are for a 1.4 GHz Pentium III Processor.

We used 10 pinch points in our analysis in order to keep the numbers down. Although the relative performance of each approach alters slightly given an increased quantity of hidden state (the advantage of reachability analysis over the complete solution, for example, will increase), we found that 10 pinch points was enough to portray the general trend.

Figure 3.40: A simulated traverse (in blue) through a traversability map extracted from the outdoor environment shown in Figure 3.33. 10 pinch points are shown in green (and circled in red for clarity). This outdoor data was obtained by Omead Amidi and Ryan Miller.

The results from this experiment are presented in Table 3.3. The first criteria used to evaluate the approaches was the number of information states examined. For the complete approach, this is a fixed number, as it exhaustively solves each information state from the deterministic cases upwards. Reachability analysis allows us to reduce the number of examined states quite considerably. AO* at times examines only a fraction of the states, however on occasion it was forced to deal with the complete information space. PAO* was able to keep the number of considered states extremely low, on average looking at only 406 (out of a state space of 59048).

To compare PAO* with AO* more thoroughly, we generated results for the number of states expanded during the run of each algorithm. This corresponds to the number of fringe elements which were further processed to produce their traversable and obstacle children. Because the partial solution graph maintained by these approaches is continually updated and reshaped, a single state can be expanded several times. Thus, the number of expanded states can be much larger than the total number of distinct states examined. AO* on average expanded more than $1.8 \cdot 10^4$ times as many states as PAO*.

The enormous difference in state expansions carried over into the run time results. AO* performed on average much more poorly than the complete solution, although certain environments it was able to solve very quickly. PAO* performed considerably better than any of the other approaches, with an average run time of 0.03 seconds.

The effectiveness of each approach is highly dependent on the nature of the environment in which we wish to plan. We have been interested in solving the navigation problem for outdoor environments and generated our range of test scenarios accordingly. However, for different environments, particularly indoor scenarios, the relative performance of the

approaches may be a little different. In particular, in run time values the approaches would be even more separated, as only a fraction of the information states are reachable when the order of the adjacency list between faces is small (a typical characteristic of indoor environments), and the inter-dependencies between face costs are even stronger. These changes do not affect the overall performance advantage of PAO*, however, which dominated every criterion in every environment we tested (including some indoor scenarios which have not been reported).

We have included in Figure 3.40 one possible resulting traverse for an agent planning optimally in a larger section of the outdoor environment shown in Figure 3.33 containing 10 pinch points[12]. The agent started on the left side of the environment (shown in dark blue) and made its way to the goal at the far right (shown in red). It encountered three pinch points, one of which turned out to be untraversable.

Combining these results with the computation required to extract pinch points from the previous section, we have the entire process conservatively taking 6.7 seconds. This time reflects the worst results reported here and in the previous section for environments with 10 pinch points.

A complete partially observable solution over such an environment would need to contend with 3^{40000} states rather than the 3^{10} used in our PAO* planning. This is currently far too large a problem to be solved optimally. Focusing our computation on pinch points, we are able to solve the planning problem much more efficiently than the full partially observable approach yet still incorporate key areas of uncertainty to produce robust paths. In addition, the complexity of our approach, through both the extraction of pinch points and the subsequent PAO* search for a solution, is highly dependent on both the quality of the information the planning agent holds and the nature of the environment. Thus, when confronted with simple environments and reliable information, it is able to exploit these desirable attributes to produce solutions very quickly. Meanwhile, in more complex environments with less reliable information, our approach incorporates more areas of uncertainty to produce very robust solutions. The resulting approach is fast enough to be used by real systems operating with imperfect information in real environments.

Recently, research related to our pinch point planning has been performed by Nabbe and Hebert [Nabbe and Hebert, 2004] and Likhachev and Stentz [Likhachev and Stentz, 2006]. Nabbe and Hebert present an approach that plans information gathering actions to account for uncertain information. Their algorithm estimates the reduction in path cost that would result from making observations from different locations in the environment and uses this information to plan the next position for the agent to move towards.

[12]In this case, the pinch points were manually specified and given probabilities of 0.5 of being untraversable. Their actual values were generated randomly.

Likhachev and Stentz present an approximation algorithm that extends our pinch point planning algorithm to incorporate huge numbers of pinch points. Their approach works by exploiting a key property of the pinch points, namely, that there is a clear preference for their values. Clearly, if the traversability of a pinch point is unknown, then the agent would always rather find out that the pinch point is traversable than untraversable. Likhachev and Stentz use this preference to develop an efficient planner that executes a series of deterministic searches to construct and refine a solution in an anytime fashion. Their approach does not necessarily provide an optimal solution, but it can handle thousands of unknown areas in the environment and has a number of nice properties [Likhachev and Stentz, 2006].

Both of these recent approaches intelligently reason about the uncertainty in the agent's initial information to provide more robust solutions.

3.8 Discussion

In this chapter, we have described two general approaches for dealing with imperfect information: assumptive-based planning and planning with uncertainty. Assumptive planning is very computationally efficient and works well for mobile robot navigation in open or highly-connected environments, where incorrect assumptions about the terrain in a given area will not have serious consequences on an agent's traverse. Planning with uncertainty is much more computationally expensive but can produce better paths for our agents, particularly when dealing with environments that are uncertain and contain areas that could be expensive to circumvent if found to be untraversable.

In order to address some of the limitations of current techniques, we have presented new algorithms that fall under each of these general approaches.

For assumptive planning, we have introduced algorithms that address three significant shortcomings associated with current approaches. First, we presented Delayed D*, an optimal graph-based replanning algorithm that delays the processing of path cost increases as long as possible. Instead of treating both edge cost increases and edge cost decreases equally, as the D* and D* Lite algorithms do, Delayed D* ignores edge cost increases that do not reside upon the current solution path. As a result, it is able to provide optimal paths more efficiently than existing replanning algorithms, particularly in common mobile robot navigation scenarios.

The second shortcoming of current assumptive planning approaches we addressed concerns the quality of paths produced over grids. Almost all grid-based planners are limited to finding paths that transition only between adjacent grid points. This creates unnatural and often costly paths. To alleviate this problem, we presented Field D*, a replanning algorithm that uses linear interpolation to approximate the path costs of points not sampled on

the grid. This allows paths to transition between any two points on adjacent grid cell *edges*, rather than just between grid cell centers or corners. Thus, the paths produced are less costly and involve less unnecessary turning than those produced using current grid-based approaches.

The third shortcoming we addressed concerns the memory requirements of grid-based path planning. In large environments, planning over a uniform resolution grid requires a huge amount of memory. Further, robots often have information concerning different parts of the environment at different resolutions, but many existing planners require that the entire environment be represented at the highest resolution for which any information is available. To remedy this, we presented Multi-resolution Field D*, an algorithm able to plans and replan over non-uniform resolution grids. The non-uniformity of the grids allows us to represent at a low resolution areas of the environment for which only low resolution information exists. This significantly reduces the memory and often the runtime requirements of the planning task. Furthermore, because of its use of linear interpolation, the paths provided by Multi-resolution Field D* are comparable in quality to those of uniform resolution Field D*. Consequently, it is particularly effective for mobile robot navigation in large outdoor environments. It is our belief that, by combining interpolation with non-uniform representations of the environment, we can 'have our cake and eat it too', with a planner that is extremely efficient in terms of both memory and computation while still producing very direct, low-cost paths.

These algorithms have been extended in a number of ways. For example, a 3D version of the Field D* algorithm has been developed for vehicles operating in the air or underwater [Carsten et al., 2006]. Further, interpolation is currently being incorporated into the TEMPEST mission-level path planner, which takes into account time and energy constraints while generating paths [Tompkins et al., 2004].

We also presented approaches that take into account the uncertainty associated with the initial information concerning the environment and reason about this uncertainty. Because planning with uncertainty can be extremely computationally expensive, we introduced an approach that focuses computation on areas of the environment whose uncertainty is most important given the current navigation task. This approach extracts these key areas, known as pinch points, then incorporates their uncertainty into the planning process. To do this planning efficiently, we introduced a new heuristic search algorithm, PAO*, that generates solutions over AND-OR graphs. PAO* is an extension of the AO* algorithm that exploits information gained during its search as much as possible and, as a result, is able to generate solutions far more efficiently.

The algorithms presented in this chapter provide improved path planning performance for mobile robots navigating imperfectly-known environments, in terms of path quality, computational requirements, and memory requirements.

Chapter 4

Single Agent Planning with Limited Deliberation Time

When an agent must react quickly or the planning problem is complex, computing optimal paths as described in Chapter 2 can be infeasible. For instance, imagine planning a trajectory for a robotic arm operating under real-time constraints in a cluttered environment, or planning smooth trajectories for a vehicle navigating at high speed. It may not be possible to generate optimal paths for such agents in the limited time allotted for planning. Instead, in such situations we must be satisfied with the best solution that can be generated in the time available.

There are a few different approaches we can take to deal with such scenarios. First, we could plan purely locally, and only consider the next single action or small sequence of actions for our agent. This short range planning can be performed very quickly, but it can result in much longer, more costly paths for our agents overall, and can sometimes be catastrophic (for instance, leading them into dead-ends from which they can't escape). Secondly, we could use an efficient global planner that sacrifices solution quality for speed, such as the RRT sampling-based algorithm introduced in Chapter 2. Such approaches will provide global solutions, but the quality of these solutions can be very poor and the time taken to generate them can vary widely. Thirdly, we can try to use some of the heuristic-based planning algorithms we've already seen, along with their nice properties, but modify them in order to have them operate under time constraints.

In this chapter, we describe a set of algorithms that fall under this last category. We begin by introducing a class of approaches known as *anytime* algorithms that are able to operate under time constraints and improve their solutions over time. Within this class, we describe a particularly useful existing graph-based anytime algorithm known as *Anytime Repairing A** that is able to provide, and control, a bound on the quality of its solution at any point in time. This algorithm is very effective at planning under time constraints. However, it relies on perfect information concerning the environment and so is not appro-

priate for planning in imperfectly-known or changing environments. To remedy this, in Section 4.2 we introduce a novel algorithm that combines the anytime behavior of Anytime Repairing A* with the replanning capability of D* and D* Lite. The resulting approach, *Anytime Dynamic A**, is able to improve its solution over time *and* repair its solution when new information is received concerning the environment.

4.1 Anytime Planning

When deliberation time is limited, it is important that our planner is able to produce a solution very quickly, so that the agent always has some action to execute. Sometimes, there may only be a very small amount of time available to generate this solution, for instance if the agent is in jeopardy, while at other times the agent may much more time to plan, for instance if the agent is in wide open space. Further, the difficulty of planning at each step may also vary significantly: at some points it may be obvious what the best plan is, while at others it may be very difficult to generate any feasible plan at all.

One useful class of planning approaches that operate effectively under a range of planning times and problem difficulties are known as *anytime* algorithms. Anytime algorithms typically construct an initial, possibly highly suboptimal, solution very quickly, then improve the quality of this solution while time permits [Zilberstein and Russell, 1995, Dean and Boddy, 1988, Zhou and Hansen, 2002, Likhachev et al., 2003, Horvitz, 1987]. They are thus able to generate solutions that satisfy very tight time constraints when the agent needs to act in a hurry, as well as to produce high-quality solutions when the agent has a little more time to think.

A number of different anytime algorithms exist in the AI community [Ambite and Knoblock, 1997, Zilberstein and Russell, 1993, Hawes, 2002, Pai and Reissell, 1998, Prendinger and Ishizuka, 1998]. Although these are not specifically path planning or graph search algorithms, some general approaches have been devised that can take almost any search algorithm and transform it into an anytime version. For instance, one simple anytime strategy is to iteratively expand the horizon or depth to which the search is performed. The idea is to first search out from s_{start} until the search tree is grown to a specified depth and choose the best solution found out to that depth, using heuristic values for states on the leaves or fringes of the search tree. This is commonly referred to as agent-centered search [Dasgupta et al., 1994, Koenig, 1997] and is similar to the local planning approaches mentioned in Chapter 2. Next, if time allows, the specified depth can be increased and the search tree can be grown out further to generate a better solution. The process repeats until the time available for planning runs out. One benefit of such an approach is that by starting with a small depth, solutions can be generated extremely quickly, so time constraints can be easily satisfied. Unfortunately, this technique suffers from the same limitations as the purely

local planning approaches already discussed. In particular, no bounds on the quality of the solutions can be provided, and in malicious environments or search spaces this approach can lead the agent into areas from which it cannot escape.

Several heuristic-based anytime search algorithms also exist [Bonet and Geffner, 2001, Korf, 1993, Zhou and Hansen, 2002, Edelkamp, 2001, Rabin, 2000, Chakrabarti et al., 1988]. These often rely on the same basic idea as Weighted A*: in many domains, inflating the heuristic values used by A* often provides substantial speed-ups at the cost of solution optimality. Recall from Chapter 2 that if the heuristic values used by A* are multiplied by an inflation factor $\epsilon > 1$, then the cost of the resulting solution obtained is also guaranteed to be within ϵ times the cost of the optimal solution [Pearl, 1984].

Taking advantage of this property, Zhou and Hansen developed the *Anytime A** algorithm. This algorithm begins by performing a Weighted A* search with a large ϵ value. After this initial search is completed, it continues to process states off the queue whose (uninflated) f-values ($g(s) + h(s)$) are less than or equal to the cost of the best solution generated thus far. This algorithm thus quickly produces an initial, ϵ-suboptimal solution, and gradually improves this solution over time. However, their algorithm has no control over the suboptimality bound while the initial solution is improved upon. In contrast, Likhachev, Gordon, and Thrun present an anytime algorithm that performs a succession of A* searches, each with a decreasing inflation factor, where each search reuses efforts from previous searches [Likhachev et al., 2003]. This approach provides suboptimality bounds for each successive search and has been shown to be much more efficient than competing approaches [Likhachev et al., 2003].

Anytime Repairing A*

Likhachev et al.'s algorithm, Anytime Repairing A* (ARA*), uses the notion of consistency introduced in Section 3.1 to limit the processing performed during each search by only considering those states whose costs at the previous search may not be valid given the new ϵ value. The basic idea is to perform as little computation as possible while still guaranteeing the resulting solution will be ϵ-suboptimal.

It begins by performing an A* search with an inflation factor ϵ_0, but during this search it only expands each state at most once[1]. Once a state has been expanded during a particular search, if it becomes inconsistent due to a cost change associated with a neighboring state, then it is not reinserted into the queue of states to be expanded. Instead, it is placed into an *INCONS* list, which contains all inconsistent states already expanded. Then, when the current search terminates, the value of ϵ is reduced and the states in the *INCONS* list are inserted into a fresh priority queue (with new priorities based on the new inflation

[1]It is proved in [Likhachev et al., 2003] that this still guarantees an ϵ_0 suboptimality bound.

```
ComputePath()
 1 while(key(s_goal) > min_{s∈OPEN}(key(s)))
 2   remove s with the smallest key(s) from OPEN;
 3   v(s) = g(s); CLOSED ← CLOSED ∪ {s};
 4   for each successor s' of s
 5     if s' was never visited by ARA* before
 6       v(s') = g(s') = ∞;
 7     if (g(s') > v(s) + c(s, s'))
 8       g(s') = v(s) + c(s, s');
 9       if s' ∉ CLOSED
10         insert/update s' in OPEN with key(s');
11       else
12         insert s' into INCONS;
```

Figure 4.1: The ARA* Algorithm: ComputePath function

factor ϵ) to be used by the next search. This improves the efficiency of each search in two ways. Firstly, by only expanding each state at most once a solution is reached much more quickly. Secondly, by only reconsidering states from the previous search that were inconsistent, much of the previous search effort can be reused. Thus, when the inflation factor is reduced between successive searches, a relatively minor amount of computation is required to generate a new solution.

A simplified, forwards-searching version of the algorithm is given in Figures 4.1 and 4.2. Here, the priority of each state s in the *OPEN* queue is computed as the sum of its cost $g(s)$ and its inflated heuristic value $\epsilon \cdot h(s, s_{goal})$ (Figure 4.2, line 1). Note that since this algorithm is forwards-searching, $g(s)$ is the cost of a path from s_{start} to s, as in the A* algorithm. Also, the heuristic function used must be consistent in order to provide ϵ-suboptimality bounds on the solutions produced by the algorithm. *CLOSED* contains all states already expanded once in the current search, and *INCONS* contains all states that have already been expanded and are inconsistent. Other notation should be consistent with that described earlier.

ARA* has been shown to be much more efficient than competing approaches and has been applied successfully to high-dimensional state spaces, such as kinematic robot arms with up to 20 links [Likhachev et al., 2003]. It has effectively extended the applicability of discrete planning algorithms into much higher dimensions than previously possible.

Limitations

ARA* does a very good job of providing anytime behavior over fairly high-dimensional problems. However, if the agent also holds incomplete or imperfect initial information,

```
key(s)
  1 return g(s) + ε · h(s, s_goal);
Main()
  2 g(s_goal) = v(s_goal) = ∞; v(s_start) = ∞;
  3 g(s_start) = 0; OPEN = CLOSED = INCONS = ∅;
  4 insert s_start into OPEN with key(s_start);
  5 ComputePath();
  6 publish current ε-suboptimal solution;
  7 while ε > 1
  8   decrease ε;
  9   Move states from INCONS into OPEN;
 10   Update the priorities for all s ∈ OPEN according to key(s);
 11   CLOSED = ∅;
 12   ComputePath();
 13   publish current ε-suboptimal solution;
```

Figure 4.2: The ARA* Algorithm: Main function

then the solution will need to be repaired (as discussed in Chapter 3). Currently, the only algorithms that can cope with such scenarios are highly-inflated versions of D* or D* Lite (i.e. we run D* or D* Lite but inflate the heuristic with an $\epsilon > 1$). But these replanning algorithms do not improve the quality of their solutions over time. To cope with time constraints and imperfect initial information, we need a planning algorithm that can both improve and repair its solution over time and in the face of new information. We introduce such an algorithm in the next section.

4.2 Anytime Dynamic A*

When the state space is complex *and* the agent is receiving updated information concerning its environment, neither the replanning algorithms introduced in Chapter 3 nor the static anytime planners introduced above will suffice. We need an algorithm that is able to *both* replan when new information is received *and* improve its solution over time.

As two motivating examples, consider motion planning for a kinematic arm in a populated environment and for an outdoor vehicle moving at high speed. A planner for either of these tasks would ideally be able to replan efficiently when new information is received indicating that the environment has changed. It would also need to generate suboptimal solutions, as optimality may not be possible given its limited deliberation time. In light of this, it would ideally improve upon its solutions as time allows.

A paucity of algorithms exist that provide such performance, and to the best of our knowledge, no search-based algorithms whatsoever. Chien et al. [Chien et al., 2000] present

a symbolic planner that returns partial plans and improves and repairs these plans as changes in the environment are observed. Ferrer [Ferrer, 2002] presents a symbolic approach that repairs the current plan by replacing parts of it with alternative subplans when new information is received, then improves the resulting plan over time. This latter approach provides complete plans, but provides no bounds on the suboptimality of these plans.

Given the strong fundamental similarities between the forwards-searching versions of the D* Lite and ARA* algorithms, it seems appropriate to look at whether the two could be combined into a single anytime, incremental replanning algorithm that could provide the sort of performance required in our motivating motion planning examples.

To this end, we have developed *Anytime Dynamic A** (Anytime D*), an anytime replanning algorithm that combines the anytime capability of ARA* with the replanning capability of D* Lite [Likhachev et al., 2005a]. Anytime D* tunes the quality of its solution based on available search time, at every step reusing previous search efforts. When updated information regarding the environment is received, the algorithm incrementally repairs its previous solution. The result is an approach that combines the benefits of anytime and incremental planners to provide efficient, bounded solutions to complex path planning problems in partially-known or dynamic environments.

Anytime D* performs a series of searches using decreasing inflation factors to generate a series of solutions with improved bounds, as with ARA*. When there are changes in the environment affecting the cost of edges in the graph, locally affected states are placed on the *OPEN* queue with priorities equal to the minimum of their previous key value and their new key value, as with D* Lite. States on the queue are then processed until the current solution is guaranteed to be ϵ-suboptimal.

This approach is able to both repair and improve the solution at the same time. If no changes are taking place in the environment, the algorithm reduces to the ARA* algorithm. If the inflation factor ϵ is set to 1, then the algorithm reduces to D* Lite.

The Anytime D* Algorithm

A forwards-searching version of the algorithm is presented in Figures 4.3 and 4.4. The *Main* function first sets the inflation factor ϵ to a sufficiently high value ϵ_0, so that an initial, suboptimal plan can be generated quickly (Figure 4.4, lines 6 to 9). Then, unless changes in edge costs are detected, the Main function decreases ϵ and improves the quality of its solution until it is guaranteed to be optimal, that is, $\epsilon = 1$ (Figure 4.4 lines 22 to 26). This phase is exactly the same as for ARA*: each time ϵ is decreased, all inconsistent states are moved from *INCONS* to *OPEN* and *CLOSED* is made empty.

When changes in edge costs are detected, there is a chance that the current solution will no longer be ϵ-suboptimal. If the changes are substantial, then it may be computationally

UpdateSetMembership(s)

1 if ($v(s) \neq g(s)$)
2 if ($s \notin CLOSED$) insert/update s in $OPEN$ with key(s);
3 else if ($s \notin INCONS$) insert s into $INCONS$;
4 else
5 if ($s \in OPEN$) remove s from $OPEN$;
6 else if ($s \in INCONS$) remove s from $INCONS$;

ComputePath()

7 while(key(s_{goal}) > $\min_{s \in OPEN}$(key(s)) or $v(s_{goal}) < g(s_{goal})$)
8 remove s with the smallest key(s) from $OPEN$;
9 if ($v(s) > g(s)$)
10 $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
11 for each successor s' of s
12 if s' was never visited by Anytime D* before
13 $v(s') = g(s') = \infty$; $bp(s') =$ **null**;
14 if ($g(s') > v(s) + c(s, s')$)
15 $bp(s') = s$;
16 $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
17 else
18 $v(s) = \infty$; UpdateSetMembership(s);
19 for each successor s' of s
20 if s' was never visited by Anytime D* before
21 $v(s') = g(s') = \infty$; $bp(s') =$ **null**;
22 if ($bp(s') = s$)
23 $bp(s') = \text{argmin}_{s'' \in Pred(s')} v(s'') + c(s'', s')$;
24 $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);

Figure 4.3: Anytime D*: ComputePath function

expensive to repair the current solution to regain ϵ-suboptimality. In such a case, the algorithm increases ϵ so that a less optimal solution can be produced quickly (Figure 4.4, lines 20 to 21). Because edge cost increases may cause some states to become underconsistent, a possibility not present in ARA*, states need to be inserted into the *OPEN* queue with a key value reflecting the minimum of their old cost and their new cost. Further, in order to guarantee that underconsistent states propagate their new costs to their affected neighbors, their key values must use uninflated heuristic values. This means that different key values must be computed for underconsistent states than for overconsistent states (Figure 4.3, lines 1 to 4). As with ARA*, the heuristic function used to contribute to these key values needs to be consistent.

By incorporating these considerations, Anytime D* is able to handle both changes in edge costs and changes to the inflation factor ϵ. It can also be used to interleave planning

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + ε · h(s); g(s)];
 3 else
 4   return [v(s) + h(s); v(s)];
Main()
 5 g(s_goal) = v(s_goal) = ∞; v(s_start) = ∞; bp(s_goal) = bp(s_start) = null;
 6 g(s_start) = 0; OPEN = CLOSED = INCONS = ∅; ε = ε_0;
 7 insert s_start into OPEN with key(s_start);
 8 forever
 9   ComputePath();
10   publish ε-suboptimal solution;
11   if ε = 1
12     wait for changes in edge costs;
13   for all directed edges (u, v) with changed edge costs
14     update the edge cost c(u, v);
15     if v ≠ s_start
16       if v was never visited by Anytime D* before
17         v(v) = g(v) = ∞; bp(v) = null;
18       bp(v) = arg min_{s''∈Pred(v)} v(s'') + c(s'', v);
19       g(v) = v(bp(v)) + c(bp(v), v); UpdateSetMembership(v);
20   if significant edge cost changes were observed
21     increase ε or re-plan from scratch (i.e., re-execute Main function);
22   else if (ε > 1)
23     decrease ε;
24   Move states from INCONS into OPEN;
25   Update the priorities for all s ∈ OPEN according to key(s);
26   CLOSED = ∅;
```

Figure 4.4: Anytime D*: Main function

and execution. To do this, it is useful to perform the search backwards, so that as the state of the agent s_{start} changes, the costs associated with states in the graph are still valid. For more details on why searching backwards is a sensible idea when agents are moving, see Section 3.1.

An Example

Figure 4.5 presents an illustration of several of the search algorithms we have discussed on a simple grid world planning problem. In this example we have an eight-connected grid where black cells represent obstacles and white cells represent free space. Each cell is treated as a state in our search space, with transitions available between adjacent cells, exactly as in standard grid planning (see Section 2.1). The cell marked R (initially in the bottom right

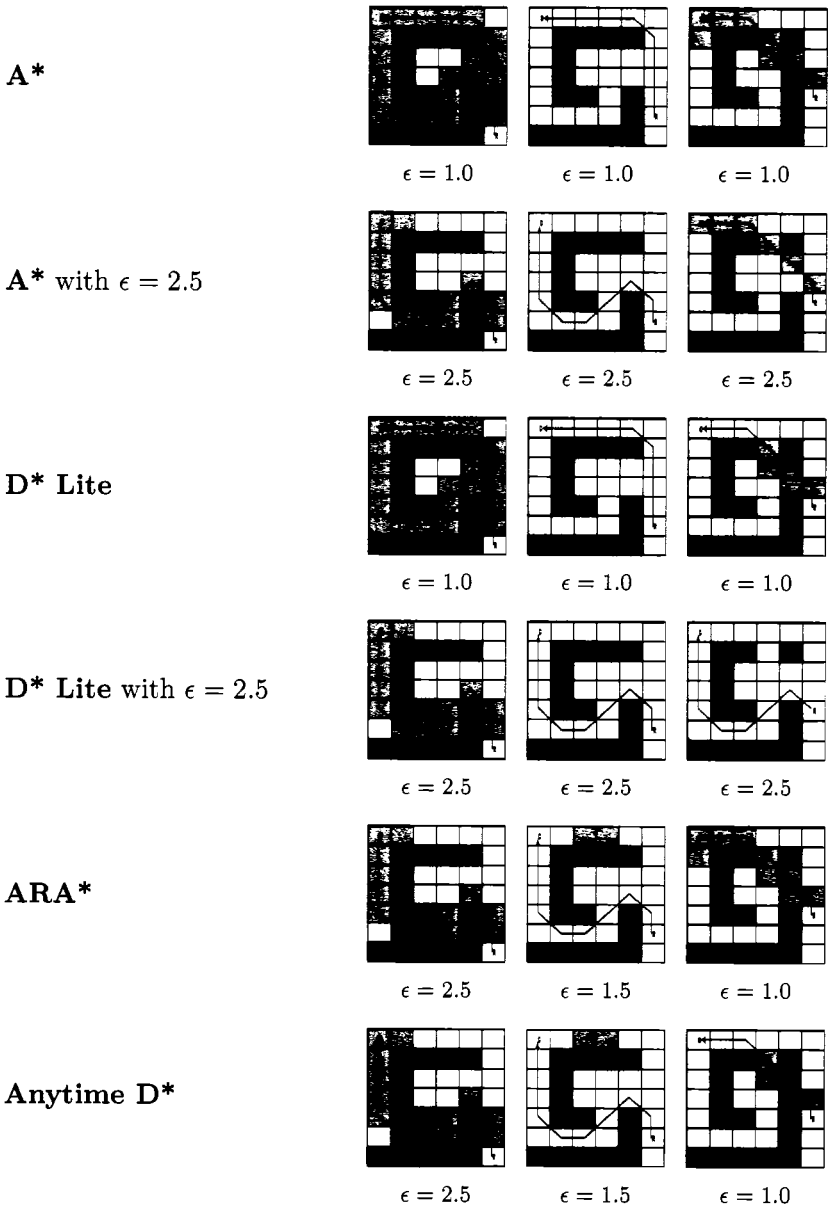


Figure 4.5: A simple 8-connected robot navigation example. The robot starts in the bottom right cell and plans a path to the upper left cell. After it has moved two steps along its path, it observes a gap in the top wall. The states expanded by each of six algorithms (A*, A* with an inflation factor ϵ, D* Lite, D* Lite with an inflation factor ϵ, ARA*, and Anytime D*) are shown (shaded) at each of the first three robot positions.

corner of the environment) denotes the position of an agent navigating this environment towards a goal cell, marked G (in the upper left corner of the environment). The cost of moving from one cell to any non-obstacle neighboring cell is one. The heuristic used by each algorithm is the larger of the x (horizontal) and y (vertical) distances from the current cell to the cell occupied by the agent. The cells expanded by each algorithm for each subsequent agent position are shown in grey (each algorithm has been optimized not to expand the agent cell). The resulting paths are shown as dark grey arrows.

The first approach shown is Backwards A*, that is, A* with its search focused from the goal state to the start state. The initial search performed by A* provides an optimal path for the agent. After the agent takes two steps along this path, it receives information indicating that one of the cells in the top wall is in fact free space. It then replans from scratch using A* to generate a new, optimal path to the goal. The combined total number of cells expanded at each of the first three agent positions is 31.

The second approach is Backwards A* with an inflation factor of $\epsilon = 2.5$. This approach produces an initial suboptimal solution very quickly. When the agent receives the new information regarding the top wall, this approach replans from scratch using its inflation factor and produces a new path, which happens to be optimal. The total number of cells expanded is only 19, but the solution is only guaranteed to be ϵ-suboptimal at each stage.

The third approach is D* Lite, and the fourth is D* Lite with an inflation factor of $\epsilon = 2.5$[2]. The bounds on the quality of the solutions returned by these respective approaches are equivalent to those returned by the first two. However, because D* Lite reuses previous search results, it is able to produce its solutions with fewer overall cell expansions. D* Lite without an inflation factor expands 27 cells (almost all in its initial solution generation) and always maintains an optimal solution, and D* Lite with an inflation factor of 2.5 expands 13 cells but produces solutions that are suboptimal every time it replans.

The final row of the figure shows the results of ARA* and Anytime D*. Each of these approaches begins by computing a suboptimal solution using an inflation factor of $\epsilon = 2.5$. While the agent moves one step along this path, this solution is improved by reducing the value of ϵ to 1.5 and reusing the results of the previous search. The path cost of this improved result is guaranteed to be at most 1.5 times the cost of an optimal path. Up to this point, both ARA* and Anytime D* have expanded the same 15 cells each. However, when the robot moves one more step and finds out the top wall is broken, each approach reacts differently. Because ARA* cannot incorporate edge cost changes, it must replan from scratch with this new information. Using an inflation factor of 1.0 it produces an optimal solution after expanding 9 cells (in fact this solution would have been produced regardless of the inflation factor used). Anytime D*, on the other hand, is able to repair its

[2]The D* Lite algorithm is described in detail in Section 3.1.

previous solution given the new information and lower its inflation factor at the same time. Thus, the only cells that are expanded are the 5 whose cost is directly affected by the new information and that reside between the agent and the goal.

Overall, the total number of cells expanded by Anytime D* is 20. This is 4 less than the 24 required by ARA* to produce an optimal solution, and much less than the 27 required by D* Lite. Because Anytime D* reuses previous solutions in the same way as ARA* and repairs invalidated solutions in the same way as D* Lite, it is able to provide anytime solutions in partially-known or changing environments very efficiently.

Theoretical Properties

The Anytime D* algorithm is guaranteed to terminate with an ϵ-suboptimal solution. It is also guaranteed to only expand each state s at most twice during each call to *ComputePath*, and only if s was inconsistent before *ComputePath* was called or its v-value is changed during the function . We include proofs of these results in the appendix; these were originally presented in [Likhachev et al., 2005b].

Theorem 3. *When the* ComputePath *function exits, the cost of the path from* s_{start} *to* s_{goal} *defined by backpointers is no larger than* $\epsilon * g^*(s_{goal})$.

The above theorem guarantees ϵ-suboptimality of the solution returned by the *ComputePath* function. The next theorem guarantees that Anytime D* will not expand states an arbitrary number of times.

Theorem 4. *No state is expanded more than twice during any particular execution of the* ComputePath *function. A state can be expanded at most once as underconsistent and at most once as overconsistent.*

The last theorem we include here shows that Anytime D* will not expand states unnecessarily.

Theorem 5. *A state s is expanded by the* ComputePath *function only if either it is inconsistent before* ComputePath *is called or its v-value is altered by* ComputePath *at some point during its execution.*

Theorem 5 also highlights the computational advantage of Anytime D* over D* Lite and ARA*. Because Anytime D* only processes exactly the states that were either inconsistent at the beginning of the current search or made inconsistent during the current search, it is able to produce solutions very efficiently. Neither D* Lite nor ARA* are able to both improve and repair existing solutions in this manner.

Anytime D* Results

One of the original motivations for developing Anytime D* was smooth trajectory planning for outdoor mobile robots operating in imperfectly-known environments, where velocity

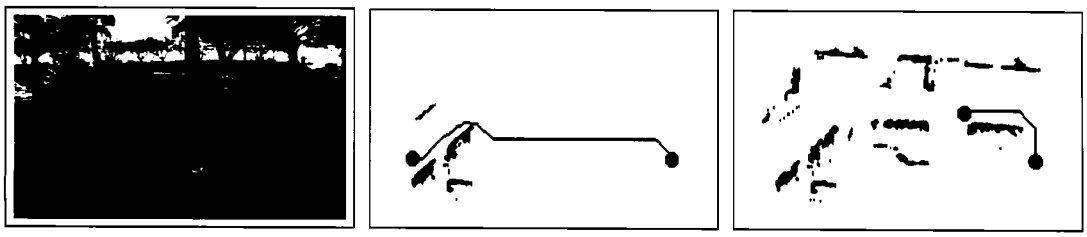


Figure 4.6: The ATRV robotic platform. Also shown are two images of the robot moving from the left side to the right side of an initially-unknown outdoor environment using Anytime D* for updating and improving its solution path.

considerations are important for generating smooth, fast trajectories. We can frame this problem as a search over a state space involving four dimensions: the $(x.y)$ position of the robot, the robot's orientation, and the robot's velocity. Solving this initial 4D search in large environments can be computationally costly, and an optimal solution may be infeasible if the initial processing time of the robot is limited.

Once the robot starts moving, it is highly unlikely that it will be able to replan an optimal path if it receives new information about the environment. But if the environment is only partially-known or dynamic, either of which is common in the areas we are interested in traversing, new information will certainly be received. As a result, the robot needs to be able to quickly generate suboptimal solutions when new information is gathered, then improve these solutions as much as possible given its processing constraints.

Anytime D* has been used to provide this capability to two robotic platforms currently used for outdoor navigation[3]. Figure 4.6 shows the first of these platforms, an ATRV vehicle equipped with two laser range finders for mapping and an inertial measurement unit for position estimation. Also shown in Figure 4.6 are two images of the map and path generated by the robot as it traversed from one side of an initially-unknown environment to the other. The 4D state space for this problem has roughly 20 million states, however Anytime D* was able to provide fast, safe trajectories in real-time. Anytime D* has also been implemented on a Segway Robotic Mobility Platform, shown in Figure 4.7. Using Anytime D*, this vehicle has successfully navigated back and forth across a substantial part of the Stanford campus. For both the ATRV and Segway implementations, an efficient 2D (x, y) planner was used to provide heuristic values to the 4D search.

To further evaluate the performance of Anytime D*, we compared it to ARA* and D* Lite (with an inflation factor of $\epsilon = 20$) on a simulated 3 degree of freedom (DOF) robotic arm manipulating an end-effector through a changing environmen. In this set of experiments, the base of the arm is fixed, and the task is to move into a particular goal

[3]The implementation of Anytime D* on these vehicles was performed by Maxim Likhachev.

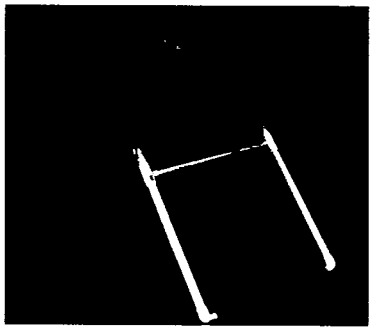


Figure 4.7: The Segway Robotic Mobility Platform

configuration while navigating the end-effector around both static and moving obstacles. We used a manufacturing-like scenario for testing, where the links of the arm exist in an obstacle-free plane, but the end-effector projects down into a cluttered space (such as a conveyor belt moving goods down a production line). Figures 4.8 and 4.9 show the environments used and the results generated by each algorithm.

In each experiment, we started with a known map of the end-effector environment. As the arm traversed each step of its trajectory, however, there was some probability $\mathcal{P}^o$ that an obstacle would appear in its path, forcing the planner to repair its previous solution. All three algorithms performed their searches in a backwards direction to account for the movement of the arm.

We have included results from two different initial environments and several different values of $\mathcal{P}^o$, ranging from $\mathcal{P}^o = 0.04$ to $\mathcal{P}^o = 0.2$. In these experiments, the agent was given a fixed amount of time for deliberation, $\mathcal{T}^d = 1.0$ seconds, at each step along its path. The cost of moving each link was nonuniform: the link closest to the end-effector had a movement cost of 1, the middle link had a cost of 4, and the lower link had a cost of 9. The heuristic used by all algorithms was the maximum of two quantities; the first was the cost of a 2D path from the current end-effector position to its position at the state in question, accounting for all the currently known obstacles on the way; the second was the maximum angular difference between the joint angles at the current configuration and the joint angles at the state in question. This heuristic is admissible and consistent.

In each experiment, we compared the cost of the path traversed by ARA* with $\epsilon_0 = 20$ and D* Lite with $\epsilon = 20$ to that of Anytime D* with $\epsilon_0 = 20$, as well as the number of states expanded by each approach. Our first environment had only one general route that the end-effector could take to get to its goal configuration, so the difference in path cost between the algorithms was due to manipulating the end-effector along this general path more or less efficiently. Our second experiment presented two qualitatively different routes the end-effector could take to the goal. One of these had a shorter distance in terms of

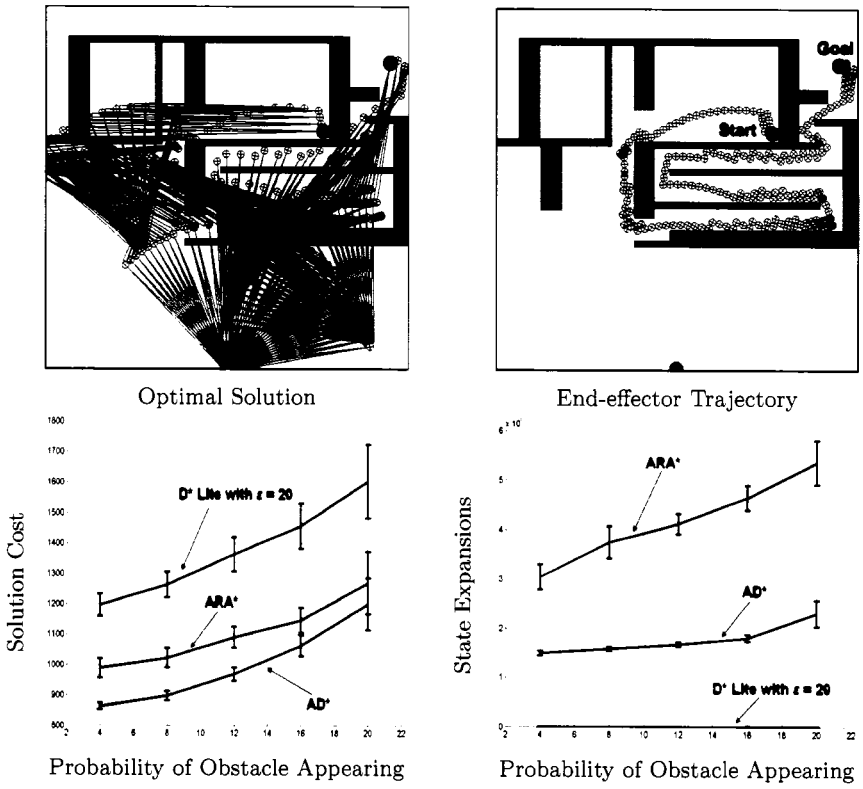


Figure 4.8: The environment used in our first experiment, along with the optimal solution and the end-effector trajectory (without any randomly-appearing obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.

end-effector grid cells but was narrower, while the other was longer but broader, allowing for the links to move in a much cheaper way to get to the goal.

Each environment consisted of a 50 × 50 grid, and the state space for each consisted of slightly more than 2 million states. The results for the experiments, along with 95% confidence intervals, can be found in Figures 4.8 and 4.9. As can be seen from these graphs, Anytime D* was able to generate significantly better trajectories than ARA* while processing far fewer states. D* Lite processed very few states, but its overall solution quality was much worse than either of the anytime approaches. This is because it is unable to improve its suboptimality bound.

We have also included results focusing exclusively on the anytime behavior of Anytime D*. To generate these results, we repeated the above experiments without any randomly-appearing obstacles (i.e., $\mathcal{P}^o = 0$). We kept the deliberation time available at each step, $\mathcal{T}^d$, set at the same value as in the original experiments (1.0 seconds). Figure 4.10 shows

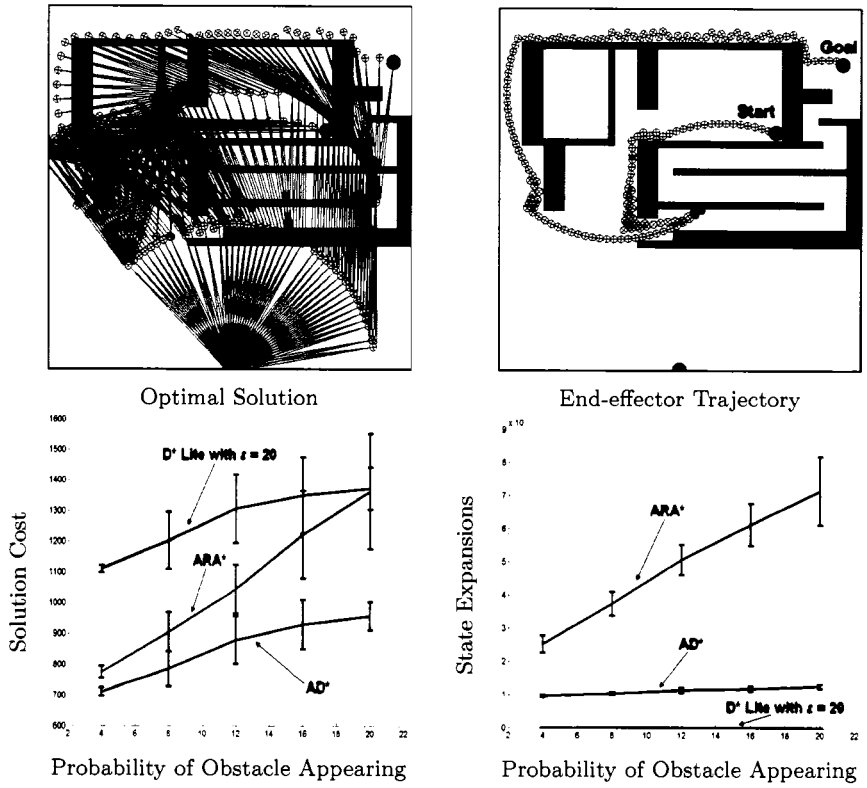


Figure 4.9: The environment used in our second experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.

the total path cost (the cost of the executed trajectory so far plus the cost of the remaining path under the current plan) as a function of how many steps the agent has taken along its path. Since the agent plans before each step, the number of steps taken corresponds to the number of planning episodes performed. These graphs show how the quality of the solution improves over time. We have included only the first 20 steps, as in both cases Anytime D* has converged to the optimal solution by this point.

We also ran the original experiments using D* Lite with no inflation factor and unlimited deliberation time to get an indication of the cost of an optimal path. On average, the path traversed by Anytime D* was about 10% more costly than the optimal path, and it expanded roughly the same number of states as D* Lite with no inflation factor. This is particularly encouraging: not only is the solution generated by Anytime D* very close to optimal, but it is providing this solution in an anytime fashion for roughly the same total processing as would be required to generate the solution in a single search.

There are a few details and extensions of the algorithm worth expanding on. Firstly,

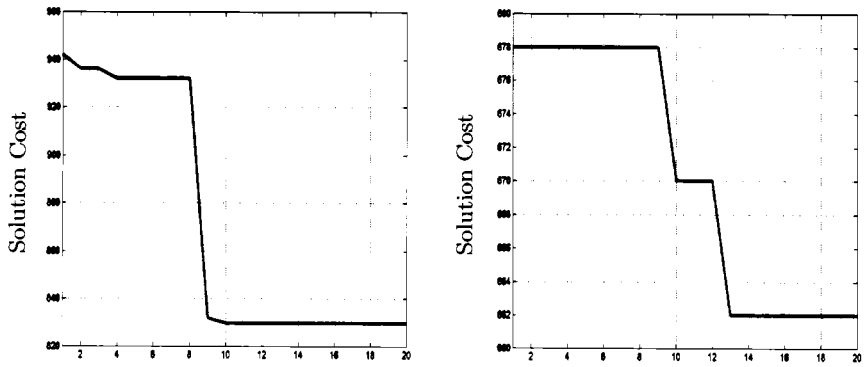


Figure 4.10: An illustration of the anytime behavior of Anytime D*. Each graph shows the total path cost (the cost of the executed trajectory so far plus the cost of the remaining path under the current plan) as a function of how many steps the agent has taken along its path, for the static path planning problem depicted to the left of the graph. Also shown are the optimal end-effector trajectories for each problem.

lines 20 - 21 of the *Main* function (Figure 4.4) state that if significant changes in edge costs are observed, then either ϵ should be increased or we should replan from scratch. This is an important consideration, as it is possible that repairing a previous solution will involve significantly more processing than planning over from scratch. Exactly what constitutes "significant changes" is application-dependent. For our outdoor navigation platforms, we look to see how much the 2D heuristic cost from the current state to the goal has changed: if this change is large, there is a good chance replanning will be time consuming. In our simulated robotic arm experiments, we never replanned from scratch, since we were always able to replan incrementally in the allowed deliberation time. However, in general it is worth taking into account how much of the search tree has become inconsistent, as well as how long it has been since we last replanned from scratch. If a large portion of the search tree has been affected and the last complete replanning episode was quite some time ago, it is probably worth scrapping the search tree and starting fresh. This is particularly true in very high-dimensional spaces where the dimensionality is derived from the complexity of the agent rather than the environment, since changes in the environment can affect a huge number of states.

Secondly, every time we change ϵ the entire *OPEN* queue needs to be reordered to take into account the new key values of the states residing upon it (Figure 4.4 line 25). This can be a rather expensive operation. It is possible to avoid this full queue reorder by extending an idea originally presented along with the D* algorithm [Stentz, 1995] and briefly mentioned in Section 3.1, in which a bias term is added to the key value of each state

being placed on the queue. This bias is used to ensure that those states whose priorities in the queue are based on old, incorrect key values are at least as high as they should be in the queue. In other words, by adding a fixed value to the key of each new state placed on the queue, the old states are given a relative advantage in their queue placement. When a state is popped off the queue whose key value is not in line with the current bias term, it is placed back on the queue with an updated key value. The intuition is that only a small number of the states previously on the queue may ever make it to the top, so it can be much more efficient to only reorder the ones that do. We can use the same idea when ϵ decreases (from ϵ^o to ϵ^n, say) to increase the bias term by $(\epsilon^o - \epsilon^n) \cdot \max_{s \in OPEN} h(s, s_{goal})$. Note that this assumes the algorithm is searching in a forwards direction. The key value of each state becomes

$$\text{key}(s) = [\min(v(s), g(s)) + \epsilon \cdot h(s, s_{goal}) + \textbf{bias}, \min(v(s), g(s))].$$

By using the maximum heuristic value present in the queue to update the bias term, we are guaranteeing that each state already on the queue will be at least as elevated on the queue as it should be relative to the new states being added.

Finally, it may be possible to reduce the effect of underconsistent states in our repair of previous solution paths. With the current version of Anytime D*, underconsistent states need to be placed on the queue with a key value that uses an uninflated heuristic value. This is because they could reside on the old solution path and their true effect on the start state may be much more than the inflated heuristic would suggest. This means, however, that the underconsistent states quickly rise to the top of the queue and are processed before many overconsistent states. At times, these underconsistent states may not have any effect on the value of the start state (for instance when they do not reside upon the current solution path). In Section 3.2 we introduced the Delayed D* replanning algorithm, which delays the processing of underconsistent states as long as possible. This algorithm can be significantly more efficient than the D* and D* Lite algorithms. Incorporating the ideas behind Delayed D* into the Anytime D* algorithm could provide even greater efficiency gains, since underconsistent states are so expensive for Anytime D* to process.

4.3 Discussion

In this chapter we have presented various approaches for path planning under time constraints. In particular, we have described a class of algorithms known as *anytime* algorithms that are able to quickly generate an initial solution, then improve this solution as deliberation time allows. We discussed one such algorithm, Likhachev et al.'s *ARA** algorithm [Likhachev et al., 2003], that is extremely effective for path planning in complex search spaces when deliberation time is limited. ARA* performs a series of weighted A* searches,

each with a decreasing inflation factor ϵ, to generate a series of improved solutions. Each search reuses information from previous searches to avoid repeating any computation. As a result, it is able to efficiently provide anytime performance and can provide bounds on the quality of its solution at any stage.

However, the ARA* algorithm cannot cope with changes to the underlying search graph, such as might be encountered when the initial information available to an agent is imperfect. In such cases, ARA* must reset its inflation factor ϵ to its initial value and plan a new solution from scratch. This can be very computationally expensive, particularly if new information is being received frequently.

In order to address the problem of path planning with imperfect information *and* with time constraints, we presented the *Anytime D** algorithm. Anytime D* combines the anytime behavior of ARA* with the replanning capability of D* Lite. It performs a series of searches using a decreasing inflation factor ϵ, as with ARA*. When new information is received, it incrementally repairs its current solution, as with D* Lite. The result is an approach that combines the benefits of both anytime and replanning algorithms to provide efficient, bounded solutions to complex path planning problems in partially-known or changing environments. We have provided theoretical properties of the algorithm as well as results in the domains of mobile robot navigation and robotic arm manipulation. To our knowledge, Anytime D* is the only heuristic search algorithm that provides both anytime and replanning capabilities.

Chapter 5

Single-agent Planning in Dynamic Environments

When the environment contains some dynamic elements, such as moving obstacles or other agents, the task of planning a safe path for an agent becomes more complicated, as it involves reasoning about the behavior of these dynamic entities. Sometimes it is possible to ignore the dynamic elements by treating them as static obstacles, then approach path planning in dynamic environments as the same problem as path planning in partially-known environments, and solve it using the assumptive algorithms introduced in Section 3.1. However, in general there is a qualitative difference between these two path planning problems that motivates using different solution techniques for each.

Consider an agent on a sidewalk trying to cross a street. Assume that there is some car coming down the street towards the agent. Now, at this point in time, the agent can observe that the street in front of it is empty. Using standard planning techniques, it could plan an unobstructed path across the road. However, once it begins to move along this path, the car could catch up to the agent and cause a collision. The issue here is that standard planning and replanning algorithms used for navigation assume that the environment is static, so that if an agent observes that the road is clear *right now*, then it concludes that it will be clear *forever*. Now, replanning algorithms allow for new information to be incorporated so that plans can be updated, but when changes are happening quickly it may not be possible to replan in time. Further, if the agent has information concerning the dynamic elements of the environment, then in order to produce a path that is optimal with respect to this information it must incorporate this information into its planning. So, even when replanning algorithms are able to replan in time to account for environment dynamics, they can produce paths that are grossly suboptimal.

The problem we focus on in this chapter is the following. Imagine we have a mobile robot navigating a complex, outdoor environment in which adversaries or other agents exist. This environment may contain terrain of varying degrees of traversability, as well as

desirable and undesirable areas based on other metrics (such as proximity to adversaries or friendly agents, communication access, or resources). Imagine further that this agent is trying to reach some goal location in the environment, while incurring the minimum possible combined cost according to all of the above metrics, as well as (perhaps) time. Specifically, imagine the agent is trying to minimize the cost of its trajectory according to some function $\mathcal{C}$, defined as:

$$\mathcal{C}(\text{path}) = w_t \cdot t_{\text{path}} + w_c \cdot c_{\text{path}},$$

where t_{path} is the time taken to traverse the path, c_{path} is the cost of the path based on all relevant metrics other than time, and w_t and w_c are the weight factors specifying the extent to which each of these values contributes to the overall cost of the path ($w_t + w_c = 1$).

In a real world scenario it is likely that the agent will not have perfect information concerning the environment, and there may be dynamic elements in the environment that are not under the control of the agent. It is important that the agent is able to plan given various degrees of uncertainty regarding both the static and dynamic elements of the environment. Further, as the agent navigates through this environment, it receives updated information through onboard sensors concerning both the environment itself and the dynamic elements within. Thus, it is important that either the agent can repair its previous solution to account for the latest information, or that its initial solution provides a plan for the current contingency. Any required onboard planning must be performed in a very timely manner, as the best solutions may require immediate action and so that solutions are not out-of-date when they are finally generated.

In this chapter, we describe current planning approaches for dealing with dynamic environments and evaluate their applicability to this general problem scenario. We begin by describing how planning in dynamic environments can be framed as a search through state-time space, where the trajectories of dynamic elements can be represented and reasoned about. Next, we describe a number of limitations of existing path planning algorithms applied to dynamic environments, and we show how the Anytime D* algorithm presented in Chapter 4 can be applied to this domain, alleviating some of these limitations. We introduce a modified version of this algorithm tailored to dynamic environments and provide results from an outdoor mobile robot navigation scenario involving dynamic obstacles.

5.1 Planning in State-Time Space

In order to plan a path that avoids dynamic obstacles, an agent needs to be able to reason about and represent where the dynamic obstacles will be at future points in time. For instance, if an agent is planning a path straight across a road, then it needs to calculate where the vehicles on the road will be while the agent is making its way to the other side.

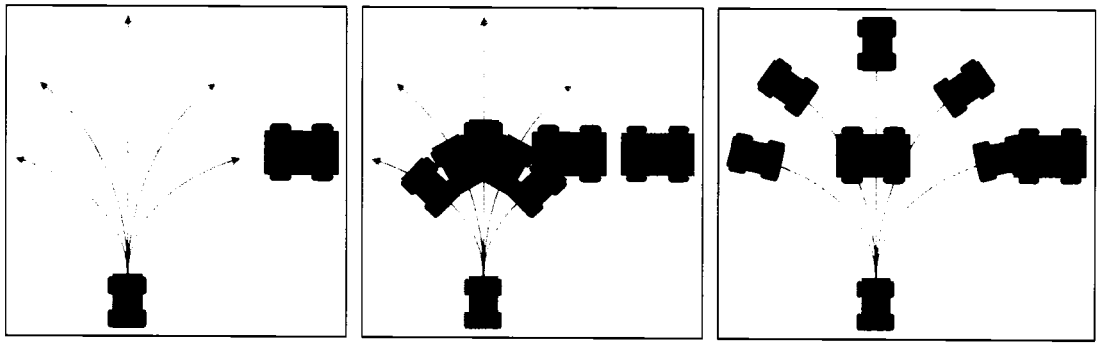


Figure 5.1: Local planning amidst dynamic obstacles. If an agent (facing upwards) assumes the dynamic obstacle (traveling in from the right) is static when choosing its next action (potential actions shown as the arcs emanating out from the agent), it may select an action that will have it collide with the obstacle at some future point in time. Instead, it needs to estimate the position of the dynamic obstacle in the future and use these estimates to select an action that will avoid the obstacle at all times. The three images show the potential position of the agent based on its available actions, as well as the position of the dynamic obstacle, at three stages in time (the color of each agent reflects the time).

In order for the agent to be sure its path is safe, it is not enough to consider just the initial positions of these vehicles, as they will move during the agent's traverse. Figure 5.1 illustrates the importance of this sort of reasoning.

One way to perform such reasoning is to plan in state-time space, where the original state space of the planning problem is extended by a dimension to include time. In this extended space, a state consists of a pair (n, t), where n is a state from the original state space and t is a point in time. State-time space allows for the representation of dynamic obstacles: the area of the original state space that a dynamic obstacle intersects or invalidates can vary with time, so that some state n that is initially affected by the obstacle (at time $t = 0$) may later be free of the obstacle after the obstacle moves (at some later $t' > t$).

This state-time space allows us to explicitly model and take into account the trajectories of dynamic objects when performing initial planning. For example, Kindel et al. [Kindel et al., 2000, Hsu et al., 2002] present a state-time space approach for computing paths for a cylindrical robot amidst hand-propelled obstacles on a frictionless 2D testbed. Their planner builds a new roadmap representation of state-time space for each planning query and is similar in nature to the RRT algorithm discussed in Section 2.2. Discrete approaches also exist for computing paths through state-time space [Fiorini and Shiller, 1996, Fujimura, 1991, Fujimura, 1995, Shih et al., 1990, Pan and Luo, 1991, Kant and Zucker, 1986, Reif and Sharir, 1985, Tompkins et al., 2004]. For instance, the Tempest algorithm extends D* to operate over state-time space and has been effective at incorporating time-varying costs into the planning process [Tompkins et al., 2004]. Some of these approaches are also

able to compute time-optimal trajectories that take into account the dynamic constraints of the planning agent as well as the environment (e.g. [Shih et al., 1990, Fiorini and Shiller, 1996]). However, these algorithms are typically limited to simple environments where the static and dynamic elements are perfectly known a priori. In general, discrete approaches are limited to simple, low-dimensional state spaces or limit the motion of the agent to reside upon small graph representations of the environment.

Because planning in state-time space involves increasing the dimensionality of the planning problem, it can be thought of just another example of a high-dimensional planning problem. However, because of its importance, this particular high-dimensional problem has attracted a great deal of attention from robotics researchers. Further, because time is one of the dimensions of the problem, time spent planning alters the state of the system. As a result, onboard deliberation must be performed carefully and solutions must be returned quickly.

To ensure solutions are generated efficiently, recently researchers have looked at reducing the complexity of the problem by first constructing a path [Fraichard, 1999] or a PRM [van den Berg and Overmars, 2004] based on the static elements of the environment, then planning a collision-free trajectory on this path or roadmap that takes into account the dynamic obstacles. When the dynamics of the problem are not known beforehand, we have a very similar situation to that of an agent holding an imperfect map of the environment, which was discussed in Section 3.1. In this case, a PRM can be used to generate an initial path, then this PRM can be updated if dynamic obstacles present themselves along this path as it is traversed by the agent [Leven and Hutchinson, 2002, Jaillet and Simeon, 2004]. This update phase can consist of locally reconnecting two nodes using RRTs when an obstacle violates their adjoining edge or, when this fails, growing the PRM further to provide new paths [Jaillet and Simeon, 2004]. As soon as any collision-free path exists to the goal on the PRM, the update phase can terminate. These approaches work well when the static elements of the environment can be represented efficiently by a PRM and the dynamic elements have little impact on the free-space connectivity.

POMDPs can also be used to represent the problem of planning in partially-known dynamic environments. However, as was the case when dealing with partially-known static environments, solving the resulting POMDP is extremely expensive. Instead, the future trajectories of the partially-known dynamic elements are often approximated based on their initial positions and velocities, or a worst-case behavior is assumed for each dynamic element so that the path planned for the agent is guaranteed not to collide with any of these elements [Vasquez et al., 2004, Petty and Fraichard, 2005].

Limitations

Current algorithms used to plan paths through dynamic environments have a number of limitations. As stated earlier, discrete algorithms are often restricted to low-dimensional state spaces, as the memory and computation involved in planning increase dramatically with each added dimension. Further, these algorithms require some discretization of the state space, which is not always easy to come by. Thus, current discrete approaches for planning in dynamic environments are either limited to very small graph or path representations (e.g. [Fujimura, 1991, Fujimura, 1995]) or are optimal and thus may struggle to provide real-time performance when the dynamic elements are only partially-known beforehand (e.g. [Shih et al., 1990, Stentz, 1995, Fiorini and Shiller, 1996, Tompkins et al., 2004]).

Sampling-based approaches to planning in dynamic environments are able to provide solutions quickly. However, as discussed in Section 2.2, they too have a number of limitations, particularly in regards to replanning capability and solution quality. The variants of these algorithms used for planning in dynamic environments share these limitations. They either require perfect initial information regarding the dynamic (and static) elements (e.g. [Fraichard, 1999, van den Berg and Overmars, 2004]), or assume the dynamic elements are static and perform local repair when changes are observed (e.g. [Leven and Hutchinson, 2002, Jaillet and Simeon, 2004]).

In short, none of the approaches mentioned above effectively addresses the general case, where an agent may have (i) perfect information, (ii) imperfect information, *and/or* (iii) no information, regarding the dynamic elements of the environment, and the agent needs to quickly repair its trajectory when new information is received concerning *either* the dynamic *or* static elements of the environment (or have already generated a contingency plan at the outset). Further, most of these approaches assume that the only factor influencing the overall quality of a solution is the time taken for the agent to traverse the resulting path. In fact, as specified in our motivating problem, we are often concerned with minimizing a more general cost associated with the path that may incorporate, for example, traversal risk, stealth, and visibility, as well as time of traverse.

In the following sections we describe how the Anytime D* algorithm can be used to tackle this general problem.

5.2 Applying Anytime D* to Dynamic Environments

In order to navigate partially-known, dynamic environments, an agent needs to be able to generate an initial plan, then efficiently update this plan as new information is received concerning either the dynamic or static elements of the environment. Since taking dynamic

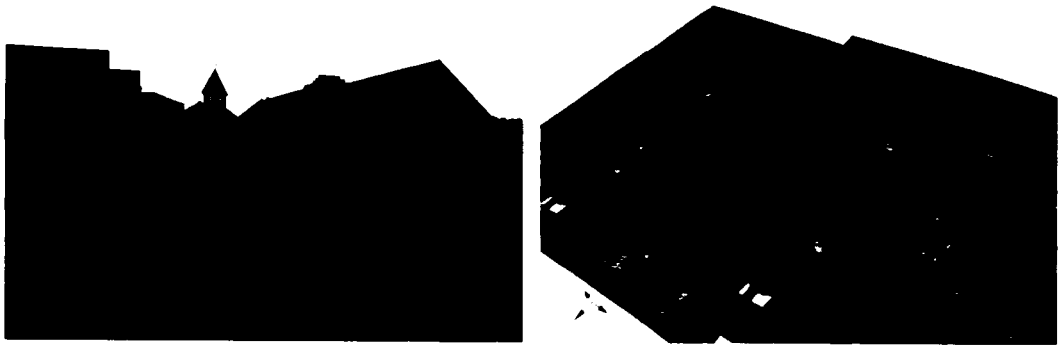


Figure 5.2: Outdoor data acquired from Fort Benning. On the left is a sample photo of the area. On the right is a 3D model of the environment used in our experiments.

elements into account during planning can be computationally expensive and the agent is operating under strict time constraints, it is important that the agent is able to plan in an anytime fashion. Further, since the trajectories of the dynamic elements may be imperfectly-known, it is important that the agent is able to repair its plans when new information is received. In light of these two considerations, the Anytime D* algorithm appears to be well-suited to solving this problem, as it combines both anytime and replanning capabilities.

In order to apply Anytime D* to this planning problem, we need to extract a graph over which Anytime D* can plan. To do this, we first use efficient sampling-based approaches to construct a roadmap encoding the static elements in the environment. This roadmap represents the connectivity of the free space without considering any of the dynamic elements. We use sampling-based approaches to construct this roadmap rather than discrete approaches because, as discussed in Section 2.2, sampling-based approaches are extremely effective at efficiently generating compact representations of the configuration space. Further, they can encode any internal constraints of the agent (such as its turning radius, kinematic limitations, etc), which can be especially important when planning in dynamic environments as it means that the time required to traverse each edge is easier to estimate.

Once we have the roadmap representing the static portion of the problem, we then use Anytime D* to plan an initial trajectory over this roadmap in state-time space, taking into account any known dynamic obstacles. This trajectory is planned in an anytime fashion and is continually improved until the time for deliberation is exhausted. Next, the agent begins to execute its trajectory. As it moves, its trajectory continues to be improved upon. When changes are observed, either to the static or dynamic elements of the environment, the current trajectory is repaired to reflect these changes. This repair is also performed in an anytime fashion, so that a valid solution can be extracted as soon as possible. We

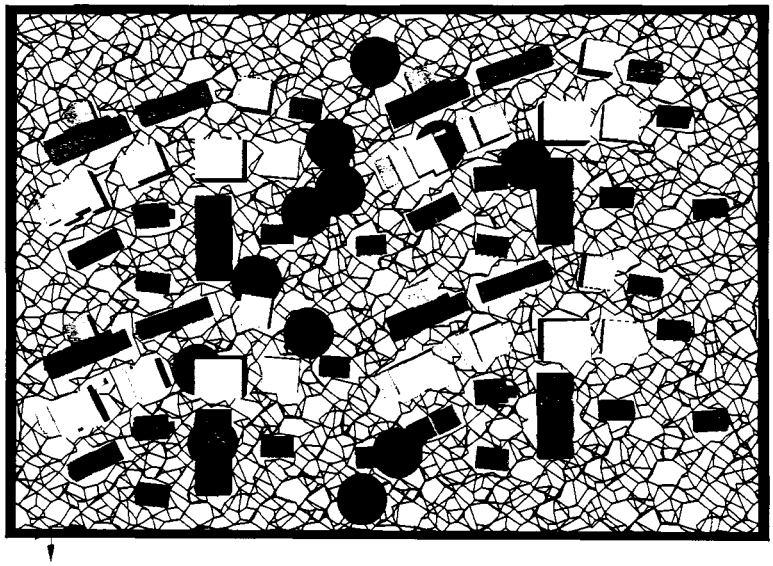


Figure 5.3: A roadmap overlaid onto the Fort Benning data (in black), along with 12 dynamic obstacles (in red). This particular roadmap assumes a holonomic vehicle.

describe each of these steps in more detail below.

Constructing the roadmap

As in other recent approaches dealing with dynamic environments (e.g. [van den Berg and Overmars, 2004, Jaillet and Simeon, 2004]), we begin by creating a PRM that takes into account the static portion of the environment. This PRM encodes any internal constraints placed on the agent's motion (such as degrees of freedom, kinematic limitations, etc) and takes into account the known costs associated with traversing different areas of the environment. It also includes cycles to allow for many alternative routes to the goal [Nieuwenhuisen and Overmars, 2004]. The objective in this initial phase is to reduce the continuous planning space into a discrete graph that is compact enough to be planned over while still being extensive enough to provide low-cost paths. Figure 5.2 presents a sample outdoor environment used for testing, and Figure 5.3 shows a sample PRM constructed from this environment.

Planning over the roadmap

We then plan a path over this PRM from the agent's initial location to its goal location, taking into account any known dynamic elements. To do this, we add the time dimension to our search space, so that each state s consists of a node on the PRM n and a time t.

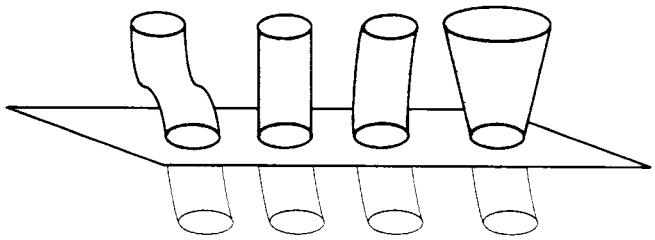


Figure 5.4: Different dynamic obstacle representations based on initial information. On the far left is a known trajectory, on the center-left an assumed-static obstacle, on the center-right an extrapolated trajectory based on previous motion, and on the far right a worst-case trajectory based on current position and maximum velocity.

This allows us to represent trajectories of dynamic elements. We discretize the time-axis into small steps of δ_t, and allow transitions from state (n,t) to state $(n,t+\delta_t)$ and to states $(n',t+c_t(n,n'))$, where n' is a successor of n in the roadmap and $c_t(n,n')$ is the time it takes to traverse the edge between them. This allows the agent to wait at a particular location as well as to transition to an adjacent roadmap location. The total cost of transitioning between state $s=(n,t)$ and some successor $s'=(n',t')$ is defined as:

$$c(s,s') = w_t \cdot (t'-t) + w_c \cdot c_r(n,n'),$$

where $c_r(n,n')$ is the cost of traversing the edge between n and n' in the roadmap.

Because the agent may not have perfect information concerning the dynamic elements in the environment, it is important that it adequately copes with partial information. There are a number of existing methods for dealing with this scenario [Vasquez et al., 2004, Petty and Fraichard, 2005]. In particular, we can estimate future trajectories based on current behavior, or we can assume worst-case trajectories. Whichever of these we choose, we end up with some trajectory or set of trajectories that we can represent as 3D objects in our state-time space (see Figure 5.4). We can then avoid these objects as we plan a trajectory for the agent.

Planning a least-cost path through this space can be computationally expensive, and although the agent may have time to generate its initial path, if the agent is continuously receiving new information as it moves, replanning least-cost paths over and over again from scratch may be infeasible. It is here that the replanning capability of Anytime D* provides benefit. In order to make this replanning as efficient as possible, the search is performed in a backwards direction from the goal to the start[1]. Since we don't know in advance at what time the goal will be reached, we seed the search with multiple goal states. For our

[1]See Section 3.1 for details on why this is important. Briefly, if we plan backwards then when the agent moves, the stored path costs are still valid because they correspond to paths to the goal, whereas if we plan forwards then these costs are invalid as they correspond to paths from an outdated position of the agent.

implementation,

$$\begin{aligned} GOALS = [&(s_{goal}, h_t(s_{start}, s_{goal})), \\ &(s_{goal}, h_t(s_{start}, s_{goal}) + \delta_t), \\ &\vdots \\ &(s_{goal}, \textit{max-arrival-time})] \end{aligned}$$

where s_{goal} is the goal position in the PRM and *max-arrival-time* is the maximum time allowed for traveling to the goal. To improve the efficiency of the search, we use two important heuristic values. First, we compute the minimum possible *time* $h_t(s_{start}, n)$ for traversing from the current agent position s_{start} to any particular position n in the PRM. Second, we compute the minimum possible *cost* $h_c(s_{start}, n)$ from the current start position to any particular position n on the PRM. These heuristic values can be efficiently computed using a fast graph search algorithm, such as Dijkstra's or A* search[2]. We then use the time heuristic to prune states added to the search and the cost heuristic to focus the search.

Specifically, if we are searching backwards from the goal states and come across some state $s = (n_s, t_s)$ with $t_s - t_{start} < h_t(s_{start}, n_s)$ we know that it is not possible to plan a trajectory from the initial location and initial time to this location by time t_s, so this state cannot be part of a solution and can be ignored. If the state passes this test, then we insert it into our search queue with a priority based on a heuristic estimate of the overall cost of a path through this state:

$$\text{key}(s) = g(s) + w_t \cdot (t_s - t_{start}) + w_c \cdot h_c(s_{start}, n_s),$$

where $g(s)$ is the current cost-to-goal value of state s. This overall heuristic estimate serves the same purpose as the f-value in classic A* search: it focuses the search towards the most relevant areas of the search space.

However, as already mentioned, the agent may not have time to plan and replan optimal paths across the PRM. Instead, it may need to be satisfied with the best solutions that can be generated in the time available. To this end, we make use of the anytime capability of Anytime D* and inflate the heuristic value for each state by $\epsilon > 1$. We then improve the quality of this solution while deliberation time allows, by decreasing ϵ. At any point during its processing, the current solution can be extracted and traversed. Then, as the agent begins its execution, Anytime D* is able to continually improve the solution.

We have included pseudocode of a backwards searching version of Anytime D*, along with modifications required to plan to multiple goals, in Figures 5.5 and 5.6. We have also

[2]If we are indeed computing the time/cost to *every* position on the PRM then it makes sense to use Dijkstra's rather than A*. However, we may only be interested in a small portion of the PRM (for instance if the agent starts out close to its goal), in which case a focused search like A* may be beneficial.

```
UpdateSetMembership(s)
 1 if (v(s) ≠ g(s))
 2   if (s ∉ CLOSED) insert/update s in OPEN with key(s);
 3   else if (s ∉ INCONS) insert s into INCONS;
 4 else
 5   if (s ∈ OPEN) remove s from OPEN;
 6   else if (s ∈ INCONS) remove s from INCONS;

ComputePath()
 7 while(key(s_start) > min_{s∈OPEN}(key(s)) or v(s_start) < g(s_start))
 8   remove s with the smallest key(s) from OPEN;
 9   if (v(s) > g(s))
10     v(s) = g(s); CLOSED←CLOSED ∪ {s};
11     for each predecessor s' of s
12       if s' was never visited by Anytime D* before
13         v(s') = g(s') = ∞; bp(s') = null;
14       if (g(s') > c(s', s) + v(s))
15         bp(s') = s;
16         g(s') = c(s', bp(s')) + v(bp(s')); UpdateSetMembership(s');
17   else
18     v(s) = ∞; UpdateSetMembership(s);
19     for each predecessor s' of s
20       if s' was never visited by Anytime D* before
21         v(s') = g(s') = ∞; bp(s') = null;
22       if (bp(s') = s)
23         bp(s') = argmin_{s''∈Succ(s')} c(s', s'') + v(s'');
24         g(s') = c(s', bp(s')) + v(bp(s')); UpdateSetMembership(s');
```

Figure 5.5: The Anytime D* Algorithm: Backwards-searching version of ComputePath function

included the modified key value calculation used in our current planning scenario (Figure 5.6 lines 2 and 4) and the construction of the roadmap encoding the static portion of the environment (Figure 5.6 lines 5 and 6).

Repairing the Plan

While the agent is traveling through the environment, it will be receiving updated information regarding its surroundings through its onboard sensors. As a result, its current solution trajectory may be invalidated due to this new information. However, it would be prohibitively expensive to replan a new trajectory from scratch every time new information arrives. Instead, we make use of the replanning capability of Anytime D* to repair the previous solution. This replanning is also performed in an anytime fashion, so that solutions

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + ε · (w_t · (t_s − t_start) + w_c · h_c(s_start, n_s)); g(s)];
 3 else
 4   return [v(s) + w_t · (t_s − t_start) + w_c · h_c(s_start, n_s); v(s)];

Main()
 5 construct PRM of static portion of environment;
 6 use PRM and Dijkstra's to extract predecessor and successor functions,
   heuristic functions h_c and h_t, and goal list GOALS;
 7 v(s_start) = ∞; bp(s_start) = null; g(s_start) = 0;
 8 OPEN = CLOSED = INCONS = ∅; ε = ε_0;
 9 for each s_goal in GOALS
10   v(s_goal) = ∞; g(s_goal) = 0; bp(s_goal) = null;
11   insert s_goal into OPEN with key(s_goal);
12 fork(MoveAgent());
13 while (s_start ∉ GOALS)
14   ComputePath();
15   publish ε-suboptimal solution;
16   if ε = 1
17     wait for changes in edge costs;
18   for all directed edges (u, v) with changed edge costs
19     update the edge cost c(u, v);
20     if u ∉ GOALS
21       if u was never visited by Anytime D* before
22         v(u) = g(u) = ∞; bp(u) = null;
23       bp(u) = arg min_{s'' ∈ Succ(u)} c(u, s'') + v(s'');
24       g(u) = c(u, bp(u)) + v(bp(v)); UpdateSetMembership(u);
25   if significant edge cost changes were observed
26     increase ε or re-plan from scratch (i.e., re-execute Main function);
27   else if (ε > 1)
28     decrease ε;
29   Move states from INCONS into OPEN;
30   Update the priorities for all s ∈ OPEN according to key(s);
31   CLOSED = ∅;
```

Figure 5.6: The Anytime D* Algorithm: (Modified) Main function

can be improved and repaired at the same time, allowing for true interleaving of planning, execution, and observation. Pseudocode of the agent traverse function is provided in Figure 5.7.

Each time the agent moves, we update the heuristic values $h_c(n)$ and $h_t(n)$ of states n on the roadmap based on the current position of the agent (Figure 5.7 line 3). This can be done very quickly as it only concerns the static roadmap. We then find the states in the

```
MoveAgent()
 1 while (s_start ∉ GOALS)
 2   update s_start to be successor of s_start in current solution path;
 3   use Dijkstra's to recompute h_c and h_t given the new value of s_start;
 4   while agent is not at s_start
 5     move agent towards s_start;
 6     if new information is received concerning the static environment
 7       update the affected edges in the PRM;
 8       update the successor and predecessor functions;
 9       use Dijkstra's to recompute h_c and h_t;
10       mark the affected edges in the Anytime D* search as changed
11     if new information is received concerning the dynamic elements
12       mark the affected edges in the Anytime D* search as changed
```

Figure 5.7: The Anytime D* Agent Traverse function

search tree that have been affected by any new information and update these states. If new information is received concerning the static elements in the environment, the roadmap is updated to reflect this new information and any edges in our state-time space search that are affected are marked as changed (Figure 5.7 lines 6 to 10). If the new information concerns only the dynamic elements, then the roadmap is left alone and just the affected edges in our state-time space search are marked as changed (Figure 5.7 lines 11 to 12). The marked edges are then updated by Anytime D* and their new values are used to update the current solution (Figure 5.6 lines 18 to 24). The entire process repeats until the agent has made it to the goal.

Modeling Dynamic Obstacles

As mentioned in the previous section, when the agent observes changes regarding the trajectory of the dynamic obstacles, we need to find all the edges in state-time space considered thus far whose collision status has changed. This consists of all the edges that were previously valid but have been invalidated by the new trajectories of the dynamic obstacles, as well as all the edges that were invalidated by the previous trajectories and are now valid based on the new trajectories. This can potentially be a rather expensive calculation, but it can be performed efficiently if we are careful about how we represent the agent and the dynamic obstacles in our state-time space. For instance, in our outdoor navigation scenario, if we model both the agent and the dynamic obstacles as discs moving in the plane, this calculation can be very fast. To do this, we first add the radius of the agent to the radii of each dynamic obstacle, so that we can treat the agent as a single point in state-time space. Next, we mathematically describe the state-time volumes carved out by each dynamic ob-

stacle. For instance, if the estimated trajectory of a dynamic obstacle is an extrapolation of its current velocity, this volume becomes a slanted cylinder, which can be described as

$$((x - x_0) - (t - t_0)v_x)^2 + ((y - y_0) - (t - t_0)v_y)^2 = r^2,$$

where (x_0, y_0) is the current position of the obstacle, (v_x, v_y) its current velocity, r its radius, and t_0 the current time. If we assume the obstacles to be static, this volume becomes a vertical cylinder, and if we model worst-case trajectories, this volume becomes a cone (provided that the maximal velocity of the obstacle is given). In any case, it is easy to check whether or not a given edge in state-time space intersects these volumes. Experiments show that it is also very fast; in the tests described below we found that over 50000 edges can be checked within 0.01 seconds.

As these estimated future trajectories of dynamic obstacles are only approximations, they will change over time. For instance, if we use the extrapolation method outlined above, whenever an obstacle changes its velocity we should re-check all edges against its new estimated trajectory. However, given an indication of the frequency of trajectory changes (the dynamism of the environment) we can set some *horizon* on the validity of the estimated trajectory such that only edges in the near future are collision-checked. Edges in the far future are simply considered to be collision-free. As time goes by, these edges are eventually checked as well. This facilitates replanning as it enables faster collision-checking and results in less of state-time space becoming (unnecessarily) invalidated.

Dynamic Environment Results

We experimented with the above approach in the environment shown in Figures 5.2 and 5.3. In our experiments, the agent had to move from the lower-rightmost node in the roadmap to the upper-leftmost node. The roadmap consisted of 4000 vertices and 6877 edges. Twelve disc-shaped dynamic obstacles (see Figure 5.3) started in the center of the scene and spread out in random directions with random velocities. During the agent's traverse the obstacles randomly changed their course. In our experiments the total number of course changes varied around 100. When an obstacle reached the boundary of the environment, it reversed its direction. Both the static and dynamic obstacles heavily impeded the agent in its attempt to reach the goal. For the sake of experimentation, the cost values for traversing edges in the roadmap were chosen to be random variations of their length. The time axis was discretized into intervals of 0.1 seconds.

The initial value of ϵ was set to 10, and was gradually decreased to 1 as deliberation time allowed. After every 0.1 seconds, the collision status of each edge in state-time space encountered thus far was checked to see if it had changed with respect to the dynamic obstacles. Each time this occured, ϵ was reset to 10 to quickly repair the path. Meanwhile,

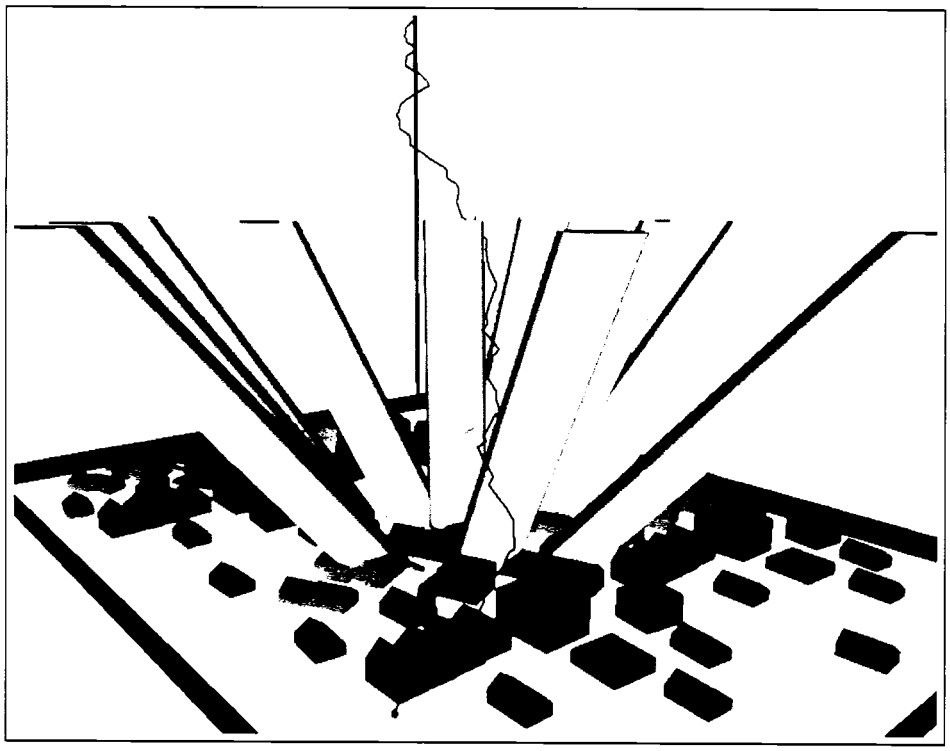


Figure 5.8: An example path (in black) planned through state-time space from the initial agent position (in green) to the goal (shown as a vertical blue line extending from the goal location upwards through time). Also shown are the extrapolated trajectories of the dynamic obstacles (in yellow). The underlying PRM has been left out for clarity but can be seen in Figure 5.3.

the position of the agent along its path was updated every 0.1 seconds according to the best available path. This continued until the agent reached its goal. These experiments were performed on a Pentium IV 3.0GHz with 1 Gbyte of memory.

In our experiments we used two models for estimating the future course of the obstacles: the extrapolation method and the assumed-static method (see Figure 5.4). In the extrapolation method we assume that the current course of an obstacle (a linear motion) is also its future course. This gives a slanted cylindrical obstacle in state-time space, which is avoided during planning. In the assumed-static case, we assume the obstacles to be static, giving vertical cylindrical state-time obstacles. In each iteration of the algorithm the position of the obstacle is updated according to its actual trajectory. For both models the horizon was set to 10 seconds. We also implemented the worst-case model (assuming a maximum velocity for each obstacle), but this model proved too conservative for the current problem, as realistic velocities for the obstacles resulted in their worst-case conical volumes becoming so large that no feasible paths existed. Figure 5.8 shows an example path planned through state-time space, along with extrapolated trajectories of the dynamic obstacles.

In our experiments we compared the PRM-based Anytime D* method (in which ϵ is regularly reset to 10) to a PRM-based D* Lite method (in which ϵ is always 1). The Anytime D* method was run in real-time, so that the agent moved along its current path regardless of whether it had finished repairing or improving the path. The D* Lite method was not run in real-time and the agent was allowed as much time for planning as it needed in between movements (in other words, time was 'paused' for the planner). We included this latter approach simply to demonstrate the relative efficiency and solution quality of Anytime D* compared to an optimal planner.

Each run consisted of a complete traverse of the agent from its initial location to its goal. During the course of this traverse, the path was improved and repaired several times. We measured the maximum time needed to improve or repair the path during each run. As the obstacle trajectories were randomly generated, we performed 50 runs using each method with the same random sequence, and averaged the maximal planning times. Additionally, we measured the average overall cost of the path, and the number of times the generated path was not safe (i.e. where there was a collision between the agent and the dynamic obstacles). Results are reported for both the extrapolation model and the static model in Table 5.1.

From the results it can be seen that for both models the maximal amount of time needed to replan the path was on average three to four times less for Anytime D* than for D* Lite. The path quality did not suffer much from the anytime-characteristic: for both Anytime D* and D* Lite the average path costs were about the same.

If we compare the extrapolation model (in which we use information of the current velocity of the obstacles) to the assumed-static case, we see that the assumed static case is quite dangerous. In approximately half of the cases, the agent hits an obstacle. In the extrapolation case about one fifth of the runs result in a collision. These collisions are explained by the radical course changes the obstacles can make; if the agent moves alongside an obstacle, and suddenly the obstacle decides to take a sharp turn, the agent may not have enough time to escape a collision (also because the velocity of the obstacle is unbounded). This can be improved by using more realistic obstacle behavior in combination with slightly more conservative trajectory estimation techniques.

The overall approach can be made even more efficient by using an expanding horizon for the dynamic obstacles. Specifically, when producing an initial plan or performing a repair we set the horizon for each obstacle to zero, so that a path is produced very quickly. Then, as deliberation time allows, we improve the accuracy and robustness of this path by gradually increasing the horizon. This modification may significantly reduce the time required to perform initial repairs on an existing solution.

We have found combining Anytime D* with sampling-based representations of the configuration space to be an effective approach for planning in dynamic environments. In our

Obstacle Model	Approach	Cost	Max.time	Invalid
Extrapolation	Anytime D*	81.07	0.19s	22%
Extrapolation	D* Lite	80.20	0.74s	18%
Assumed-Static	Anytime D*	85.09	0.22s	52%
Assumed-Static	D* Lite	81.08	0.67s	58%

Table 5.1: Anytime D* and D* Lite Results (averaged over 50 runs)

experiments we focused on a variant of the problem in which the trajectories of the dynamic elements were unknown. However, the approach is very general and can be applied to any instances of the general problem described at the beginning of this chapter. By combining anytime and replanning capabilities, Anytime D* is well-suited to producing high quality solutions and quickly generating executable trajectories, both of which are important in this problem domain. Further, as shown in this chapter it can be combined with any graph-based representation, allowing us to take advantage of recent and future developments in roadmap construction algorithms.

5.3 Discussion

In this chapter, we have described the general problem of path planning in dynamic environments. We have discussed existing approaches designed to solve this problem and pointed out a number of limitations of these approaches, particularly in regards to coping with imperfect initial information.

To address these limitations, we have shown how the Anytime D* algorithm introduced in Section 4.2 can be employed to provide anytime path planning and replanning in partially-known, dynamic environments. To begin with, a roadmap is constructed representing the static portion of the planning problem. This roadmap is then used to plan a low cost path for the agent that avoids the dynamic elements. As the agent traverses this path, it is able to update both the static and dynamic elements of the environment based on new information received. Using Anytime D* it is able to efficiently repair its current solution to reflect this new information and then improve it while deliberation time allows. The resulting approach handles the case where an agent may have (i) perfect information, (ii) imperfect information, *and/or* (iii) no information, regarding the dynamic elements of the environment, and the agent needs to quickly repair its trajectory when new information is received concerning *either* the dynamic *or* static elements of the environment. We have shown it to be capable of solving large instances of the navigation problem in dynamic, non-uniform cost environments.

Our approach combines research in discrete planning and sampling-based representations. We are unaware of any approaches to date that combine the strong body of work on discrete replanning algorithms with compact, sampling-based representations of the en-

vironment, and yet we believe the union of these two areas of research can lead to very effective solutions to a wide range of problems.

Part II

Multi-agent Planning

Chapter 6

Multi-agent Path Planning

Several tasks exist for which we would employ a team of agents rather than a single agent. Some of these tasks may be impossible to solve without using a team; others may be solved more effectively or efficiently using multiple agents. In robotics, researchers have investigated using teams of robots to solve both of these types of problems.

For instance, robot teams have been used for playing multirobot games [Bowling et al., 2004, Vail and Veloso, 2003], security sweeping and monitoring [Kalra et al., 2005, Gerkey et al., 2005], formation control [Naffin and Sukhatme, 2004, Pirjanian et al., 2002], box pushing [Brown and Jennings, 1995, Gerkey and Matarić, 2002b], and collaborative manipulation and movement of objects [Chaimowicz et al., 2001, Khatib et al., 1996, Simmons et al., 2000, Osumi et al., 1998, Ota et al., 1995, Kosuge et al., 1999, Hara et al., 1996, Wang et al., 1996, Jennings et al., 1997, Lin and Hsu, 1995, Huntsberger et al., 2003]. In these examples, a single agent cannot solve the problem and teams are used out of necessity.

Robot teams have also been used to provide better or more robust solutions to single agent problems, such as exploration [Burgard et al., 2000, Zlot et al., 2002], tracking [Stroupe, 2003, Kim et al., 2001], reconnaissance [Zlot and Stentz, 2003], and emergency handling [Østergaard et al., 2001]. For these problems, involving a team can reduce the time taken to complete the task, as well as provide improved robustness through redundancy and potentially reduce the complexity required of each agent.

In this chapter, we describe several approaches to path planning for multi-agent teams. We begin by introducing centralized algorithms, which treat the entire team as a single complex agent and then generate plans for this agent. Next, we discuss distributed algorithms, which plan for each agent individually and use coordination techniques to combine these plans. We then describe a class of algorithms that combine ideas from both centralized and distributed approaches.

We conclude the chapter by evaluating the suitability of each approach for our motivating example of constrained exploration and we provide an outline for the remainder of the thesis.

6.1 Centralized Planning

One approach to multi-agent path planning is to plan for the entire team together. To do this, the team is treated as a single complex agent and a path is planned for this single agent. This is known as centralized planning.

By planning for the entire team at once, centralized planning approaches have the potential to generate high quality solutions. However, the complexity of the planning problem increases rapidly with the size of the team and so centralized approaches are typically only used when dealing with small teams or simple problems. Since the dimensionality of the planning problem can be large even for small teams, the sampling-based algorithms described in Section 2.2 are well suited to centralized path planning.

Centralized planning has been used by researchers in robotics for various applications. Khatib et al. [Khatib et al., 1996], Ota et al. [Ota et al., 1995], and Osumi et al. [Osumi et al., 1998] all employ a centralized planner to coordinate complex agents to perform manipulation tasks. Bowling et al. [Bowling et al., 2004] plan centrally for a team of robots playing soccer. Brummit and Stentz [Brumitt and Stentz, 1996] use a centralized planner to control a set of robots exploring key areas in an environment. Koes et al. [Koes et al., 2005, Koes et al., 2006] use a centralized planner to coordinate a team of robots performing search and rescue in partially-known environments.

One centralized approach that is particularly related to our motivating constrained exploration example is work performed by Schouwenaars et al. [Schouwenaars et al., 2006]. In this work, a centralized planner based on mixed integer linear programming is used to plan for a team of agents performing constrained exploration. However, this approach suffers from a number of limitations. In particular, a fixed ordering on the line-of-sight constraints for the team is imposed, so that each agent must maintain line-of-sight communication with its two adjacent neighbors in this ordering. This severely restricts the quality of the solutions generated (the whole team is restricted to following the same homotopic class of paths) and may fail to generate solutions to complex instances of the problem. Further, in their approach only one goal is planned to, for the single agent at the furthest extent of the team (i.e. the agent with no higher neighbor in the imposed ordering). This approach is also unable to replan efficiently when new information is received or incorporate non-uniform cost areas of the environment. We will discuss the importance of these capabilities in subsequent chapters.

The RRT algorithm described in Section 2.2 can also be used to generate centralized solutions to our constrained exploration problem. This algorithm is able to plan through the full configuration space of this problem, without restricting the agents to maintain any fixed ordering or restricting the planning to a single goal. For example, Figure 6.1(a) shows an RRT and the corresponding set of paths for 3 robots moving through a 300×600

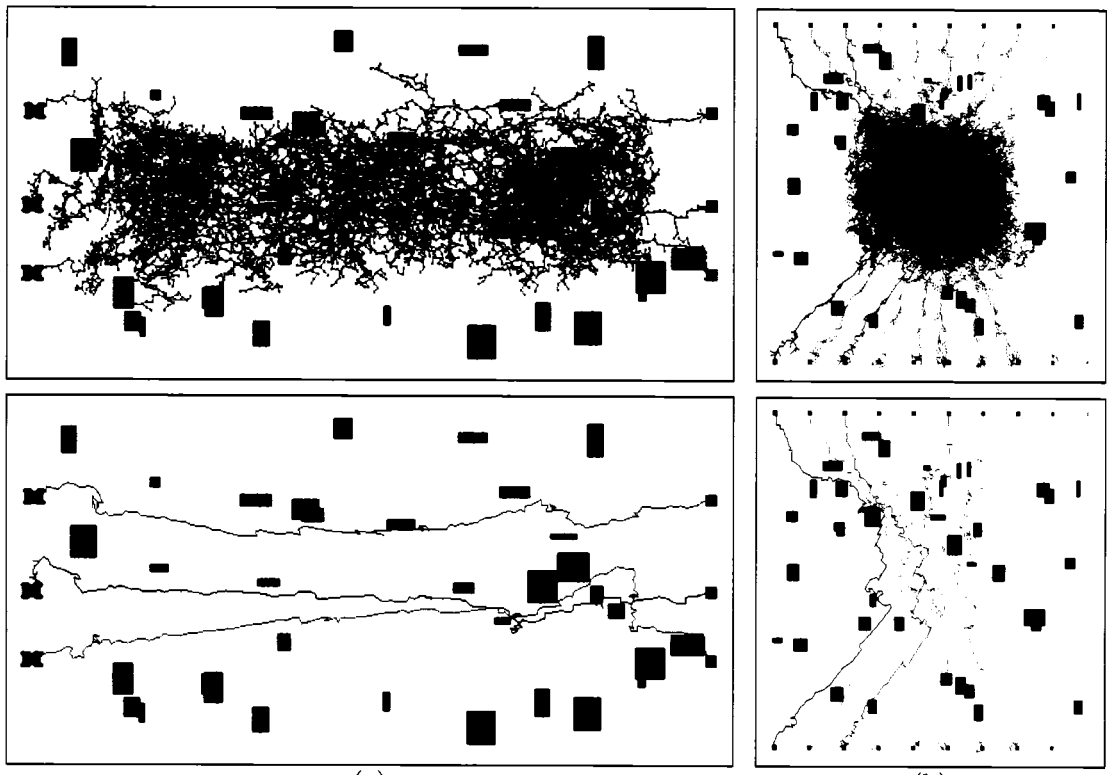


Figure 6.1: Using RRTs for Constrained Exploration. (a) A path is planned for three robots from one side to the other of an environment, while maintaining line-of-sight communication constraints among the team. The top image shows the RRT. In order to show the 6D RRT in only two dimensions, it has been split into three 2D trees. Each 2D tree encodes all the information about a single robot from the 6D tree, including edges between parent and child nodes. The red tree corresponds to the positions of robot 1 in the 6D RRT, the blue to robot 2, and the green to robot 3. The black regions depict areas through which communication cannot be made. The bottom image shows the corresponding solution path for each robot (before any smoothing). (b) A similar scenario involving 10 robots.

environment containing obstacles through which communication cannot be made. Here, the RRT began from the initial configuration of the team and grew a tree out through the configuration space, at each stage ensuring that new branches and nodes in the tree did not violate the line-of-sight connectivity constraint of the team. The size of the state space for this problem was 6×10^{15}, however the RRT was able to find a path in under a second[1]. In fact, even for medium-sized teams, RRTs are able to generate paths in a reasonable amount of time: a similar scenario involving ten robots is shown in Figure 6.1(b). This problem had a state space of 10^{54} and took 13 seconds to solve.

[1]Using a 1.5 GHz Powerbook G4.

6.2 Distributed Planning

When dealing with large teams, planning in a centralized fashion can be intractable. In such situations, distributed planning approaches are extremely popular. Distributed approaches have each agent plan individually, and are thus far less affected by increasing team sizes.

Since distributed planning approaches have each agent perform its own planning, they can be extremely efficient. However, depending on how complex the planning performed by each agent is, the overall team solutions that are produced can be highly suboptimal. Further, when there are complex constraints on the behavior of the team—as in our constrained exploration problem—distributed approaches can fail to provide any valid solution at all.

A host of different distributed approaches exist, ranging from those where each agent performs very simple, local planning, to those where each agent takes into account a variety of information concerning other agents and performs complex reasoning. Brown and Jennings [Brown and Jennings, 1995] present a simple approach for getting two robots to carry a box to a specified location, where one robot plans a path and steers the box along this path (ignoring the other robot) and the other robot reacts to where the box currently is and supports it in that direction. In this work, as in other related approaches [Hara et al., 1996, Kosuge et al., 1999], very simple planning is performed by each agent.

Slightly more complex reasoning is used by Huntsberger et al. to solve the tasks of collaborative object transport and cliff descent [Huntsberger et al., 2003, Pirjanian et al., 2002]. In the latter task, a team of agents belays one of their teammates down a cliff, and each agent decides its best course of action to provide support to the descending teammate. Similar local reasoning is used to provide formation control for multi-robot teams, where the overall team constraints can be approximated by local constraints, such as a minimum and maximum distance between an agent and its closest teammate [Parker, 1993, Yamaguchi, 1997, Balch and Hybinette, 2000, Fredslund and Matarić, 2001].

To provide improved team performance, Stroupe developed a distributed approach in which each agent took into account the current, and expected future, behavior of its teammates when generating its next action [Stroupe, 2003]. This approach was later extended by Kalra et al. [Kalra et al., 2005] to provide extended planning for each agent beyond the next immediate action. These approaches provide improved team performance by enabling each agent to incorporate more information when planning.

Distributed approaches have also been used for various instances of the constrained exploration problem. Nguyen et al. [Nguyen et al., 2003] describe a simple approach for providing a communication link between one agent and a base station by using a set of agents as relay nodes. In their approach, the agents acting as relay nodes execute a simple set of behaviors: an ordering is imposed on the agents, and each agent follows the

agent ahead of it in this ordering until communication with the agent behind it starts to weaken, at which point it stops. This produces a set of 'bread-crumb' agents that provide the necessary communication link for the original agent. However, depending on the complexity of the environment, a large number of agents may be required to create this bread-crumb trail. Wagner and Arkin [Wagner and Arkin, 2004] present an approach for performing constrained reconnaissance, where a team of robots explore an area while maintaining communication across the team. In their approach, routes and roles are planned offline by hand, and then each agent selects between different plans during execution in a reactive fashion. Because of this prior manual planning, this method is limited to small collections of fairly simple tasks and is unable to cope well with imperfect initial information.

6.3 Hybrid Planning

Since centralized and distributed approaches to multi-agent planning have complementary advantages, it makes sense to look at how aspects of each could be combined into a hybrid approach that would provide improved overall performance.

One way to do this is to use a distributed approach to generate plans for each individual agent, then have each agent re-evaluate its own plan in light of the plans of its teammates. Azarm et al. [Azarm and Schmidt, 1996] use this idea to plan paths for teams of agents: each agent plans its own path and then as the agents traverse their paths, potential collisions are detected by each agent and paths are modified to prevent these collisions. Bennewitz et al. [Bennewitz et al., 2001] use a similar idea to plan collision-free paths for teams of robots. Their approach orders the agents and then plans paths for each agent in order, taking into account the paths already planned for the previous agents to prevent any collisions. Similar to these two ideas is the fixed-path planning strategy of O'Donnell and Lozano-Pérez [O'Donnell and Lozano-Pérez, 1989], in which paths are planned for each agent independently, then a centralized planner schedules the movement of all the agents along their respective paths to ensure there are no collisions. Bererton et al. [Bererton et al., 2003, Bererton and Gordon, 2004] present a related approach that has each agent plan its own path, but a centralized planner is used to update the cost functions used by each agent in order to arrive at good overall team solutions.

More complex interaction between the agents can be achieved by using more active coordination strategies, where the agents share information or responsibilities to accomplish the task. Vail and Veloso [Vail and Veloso, 2003], Chaimowicz et al. [Chaimowicz et al., 2001, Chaimowicz et al., 2004], and Naffin and Sukhatme [Naffin and Sukhatme, 2004] have the agents on a team negotiate to assume different roles in the domains of robot soccer, box pushing, and formation control, respecively. Each role imparts a set of responsibilities and prescribed relationships between different roles enable the team to coordinate effectively.

For example, in Vail and Veloso's work, four soccer playing robots on a team negotiate for the roles of "goalie," "supporting defender," "supporting attacker," and "primary attacker." Each of these roles requires very different behavior and how two agents coordinate with each other depends heavily on their respective roles. Clark et al. [Clark et al., 2003] present a hybrid approach where centralized planning is performed for dynamically-formed subgroups of agents. Their approach enables the agents to decide which subgroups to form based on the relative positioning of the agents in the team.

One very popular class of techniques that also use negotiation between agents to improve the performance of distributed approaches are *market-based approaches.* Based on the Contract Net Protocal of Smith [Smith, 1980] and originally proposed by Stentz and Dias [Stentz and Dias, 1999], market-based approaches model the team of agents as a market economy, in which individual agents act in a self-interested fashion. Each agent tries to maximize its own revenue function, defined as the reward it obtains for completing different tasks minus the cost it expends performing the tasks. The set of tasks for the team are auctioned out to the agents, with agents placing a bid for each task based on their estimate of how much revenue they will earn from the task. The idea behind the approach is that, by having each agent maximize its individual revenue function, the overall team performance will be high. The challenge in market-based approaches is thus transforming team revenue functions into individual agent revenue functions in such a way that maximizing the latter will also maximize, or at least provide a good value for, the former.

Market-based approaches have been used by several researchers in several different domains, including exploration [Zlot and Stentz, 2003, Dias et al., 2004a], box pushing [Gerkey and Matarić, 2002a], and security sweeping [Kalra et al., 2005]. The approach maintains the efficiency and robustness of distributed approaches while also providing improved team performance. They have also been extended to provide centralized planning over subteams of agents [Dias et al., 2004b, Kalra et al., 2005]. In particular, Kalra et al. [Kalra et al., 2005] present an approach called Hoplites that allows for joint plans to be auctioned over the market, so that an agent can construct a centralized plan for itself and some group of its teammates, then try to purchase its teammates participation in this plan. This allows the agents to form sub-teams and coordinate closely when their current task is complex, yet separate again when it this close coordination is no longer necessary. This approach has been applied to the constrained exploration problem and presents a very nice combination of distributed and centralized planning. However, in order to construct good overall team solutions, it relies heavily on a good centralized planner to compute the sub-team solutions that can then be traded over the market.

Thus, when dealing with complex multi-agent path planning problems where the actions of each agent may be highly coupled, such as constrained exploration, some form of centralized planning is required. The beauty of hybrid techniques such as Hoplites is their ability

to only employ such planning when absolutely necessary, and even then only for sub-teams of agents. However, effective centralized planning algorithms are certainly required at some stage of the planning process.

In the remainder of this thesis, we focus on centralized algorithms for multi-agent path planning. In particular, we concentrate on the sampling-based RRT algorithm to cope with the high-dimensional state spaces involved when planning for teams of agents. We apply this algorithm to our motivating constrained exploration problem, which is very difficult to solve in complex instances without high quality centralized plans. We also present a number of extensions to this algorithm that improve its ability to plan with imperfect initial information and improve the quality of its solutions. We apply these extensions to various instances of the constrained exploration problem, including the cases of imperfect initial information concerning the environment, areas of non-uniform traversal cost existing within the environment, and dynamic obstacles moving about the environment.

As mentioned earlier, because the dimensionality of the joint planning problem increases with every agent added to the team, centralized planners are limited to reasonably small sized teams of agents. The algorithms we develop over the next few chapters are no exception. However, by combining these algorithms with recent hybrid approaches for multi-agent planning, such as Hoplites, high quality solutions can be provided for very large teams.

Chapter 7

Multi-agent Planning with Imperfect Information

The RRT algorithm discussed in Chapters 2 and 6 is able to efficiently provide solutions to problems involving vast, high-dimensional configuration spaces that would be intractable using discrete approaches. As mentioned, it is thus well-suited to coordinated planning for teams of agents. In Chapter 6 we saw how it can be used to generate paths for a team of agents performing constrained exploration.

However, this algorithm is unable to cope with new information concerning the environment. As discussed in Chapter 3, our initial information regarding the environment is rarely perfect and so it is important that our agents are able to update their information and solutions over time. In such scenarios, RRT-based approaches typically abandon the current solution and grow a new RRT from scratch. This can be a very time-consuming operation, particularly if the planning problem is complex. On the other hand, we have already described several sophisticated discrete replanning algorithms that are able to efficiently repair the previous solution when such changes occur without needing to replan from scratch.

In this chapter, we present a replanning algorithm for repairing RRTs when new information concerning the configuration space is received. Instead of abandoning the current RRT entirely, our approach mimics discrete replanning algorithms by efficiently removing just the newly-invalid parts and maintaining the rest. It then grows the remaining tree until a new solution is found. The resulting algorithm, Dynamic Rapidly-exploring Random Trees, is a sampling-based analog of the widely-used D* family of discrete replanning algorithms.

We begin by describing a number of extensions that have been made to RRTs to facilitate replanning. We then introduce the Dynamic Rapidly-exploring Random Tree algorithm and how it can be used for multi-agent planning with imperfect initial information. We go on

to present a number of comparative results from our constrained exploration domain and conclude by discussing a number of possible extensions.

7.1 The Extended RRT Algorithm

The essential RRT algorithm was introduced in Section 2.2 and is reproduced in Figure 7.1. The algorithm works by growing a search tree out from the initial configuration towards the goal configuration. As mentioned previously, this process can be made more efficient by biasing the search towards the goal: in the *ChooseTarget* function, we let q_{target} be the goal with probability p and choose it randomly with probability $1 - p$. This focuses the growth of the tree towards the goal.

In many domains where plans are executed by agents or teams of agents, planning and execution are interleaved: the team creates a tree, executes the returned path for some number of steps or until it is no longer valid based on new information, then grows a new tree, and so on until the goal is reached. The Execution-extended RRT (ERRT) algorithm is a further extension that is useful in these scenarios because it reuses information from previous planning episodes when generating new trees [Bruce and Veloso, 2002]. This algorithm is also included in Figure 7.1, with ERRT modifications to the RRT algorithm shown highlighted in red. After a plan is returned, the ERRT algorithm stores some of the nodes from the solution path in a waypoint cache (line 23). In future searches, the target node q_{target} is set to one of these waypoint nodes with some probability (lines 12; 15 to 16), thus biasing the growth of the tree towards previously successful solutions.

By biasing the search towards areas of the configuration space that were previously useful in providing solutions, the ERRT algorithm can be much more efficient at generating a series of solutions than the original RRT algorithm. When small changes to the configuration space are occuring, as might be the case when the team is receiving new information concerning the environment during its traverse, such biasing can greatly reduce the computation time required to arrive at a new solution. However, even the ERRT algorithm rebuilds an RRT from scratch every time new information invalidates the current solution, regardless of how much of the solution is affected. This usually results in far more work than is necessary for generating a new solution.

On the other hand, researchers have developed methods of reusing as much previous computation as possible when performing multiple-query path planning with Probabilistic Roadmaps (PRMs) [Li and Shie, 2002, Kallmann and Matarić, 2004]. In particular, the Reconfigurable Random Forest (RRF) approach of Li and Shie creates a roadmap of the configuration space using several different RRTs, each rooted at different locations [Li and Shie, 2002]. Periodically, the individual RRTs are checked to see if they can be connected together, as in the RRT-Connect algorithm [Kuffner and LaValle, 2000]. When changes

```
InitializeRRT(rrt T)
  1 T.add(q_start);
ReinitializeRRT(rrt T)
  2 T.cleartree();
  3 T.add(q_start);
GrowRRT(rrt T)
  4 q_new = q_start;
  5 while (Distance(q_new, q_goal) > distance-threshold)
  6   q_target = ChooseTarget();
  7   q_nearest = NearestNeighbor(q_target, T);
  8   q_new = Extend(q_nearest, q_target, T);
  9   if (q_new ≠null)
 10     T.add(q_new)
ChooseTarget()
 11   p = RandomReal([0.0, 1.0]);
 12   i = RandomInt([1, num-waypoints);
 13   if (p < goal-sampling-prob)
 14     return q_goal;
 15   else if (p < goal-sampling-prob + waypoint-prob)
 16     return WaypointCache[i];
 17   else
 18     return RandomConfiguration();
Main()
 19 InitializeRRT(tree);
 20 GrowRRT(tree);
 21 forever
 22   wait for changes to configuration space
 23   CacheCurrentWaypoints(tree);
 24   ReinitializeRRT(tree);
 25   GrowRRT(tree);
```

Figure 7.1: The (Extended) RRT Algorithm.

are made to the configuration space, the newly-invalid edges in the forest are removed and new trees are formed from the branches that were connected to these edges. This approach effectively updates roadmaps of the environment and is useful for multiple-query path planning in partially-known or changing environments.

In the following section we describe a related extension of the RRT algorithm that allows for efficient repair of the tree when changes are made to the configuration space. As with RRFs, this approach prunes sections of the tree that are no longer valid. However, it maintains only a single tree rather than an entire forest and thus it is particularly suited

to single-shot path planning problems. The resulting approach is simple to implement and can be much more efficient than replanning from scratch when small changes are made to the configuration space. Further, as we will show, it can be used to develop an incremental replanning algorithm for teams of robots navigating partially-known environments and is particularly well-suited to our constrained exploration problem.

7.2 Dynamic Rapidly-exploring Random Trees

Discrete replanning algorithms such as D* and D* Lite efficiently repair previous solutions when changes occur in the environment [Stentz, 1995, Koenig and Likhachev, 2002]. As mentioned in Section 3.1, they do this by determining which parts of the solution are still valid and which parts need to be recomputed. We can use this same basic idea to improve the efficiency of replanning with sampling-based algorithms such as RRTs.

The general process is illustrated in Figure 7.2. We begin with an RRT generated from an initial configuration to a goal configuration (Figure 7.2(a)). When changes occur to the configuration space (e.g. through receiving new information), we mark all the parts of the RRT that are invalidated by these changes (Figure 7.2(b) and (c)). We then trim the tree to remove all these invalid parts (Figure 7.2(d)). At this point, all the nodes and edges remaining in the tree are guaranteed to be valid, but the tree may no longer reach the goal. Finally, we grow the tree out until the goal is reached once more (Figure 7.2(e)).

We call this approach *Dynamic Rapidly-exploring Random Trees* (Dynamic RRTs). Pseudocode for trimming and regrowing the tree is presented in Figure 7.3. When an obstacle is added to the configuration space, first the edges in the current tree that intersect this obstacle are found (line 21). Each of these edges will have two endpoint nodes in the tree, with one of these endpoint nodes being the parent of the other in the RRT. In other words, one of these nodes (the parent) will have added the other (the child) to the tree through an *Extend* operation. The child endpoint node of each edge is then marked as invalid (line 5) to signify that the edge from its parent no longer represents a feasible action.

After all the child endpoint nodes of the affected edges have been marked, the solution path is checked for invalid nodes. If any are found, the RRT needs to be regrown. This involves trimming the tree and growing the trimmed tree out to the goal (lines 10 and 11). Trimming the tree involves stepping through the RRT in the order in which nodes were added and marking all child nodes as invalid whose parent nodes are invalid (lines 1 to 9). This effectively breaks off branches where they directly collide with new obstacles and removes all nodes on these branches. The tree is then re-initialized to contain only the remaining valid nodes.

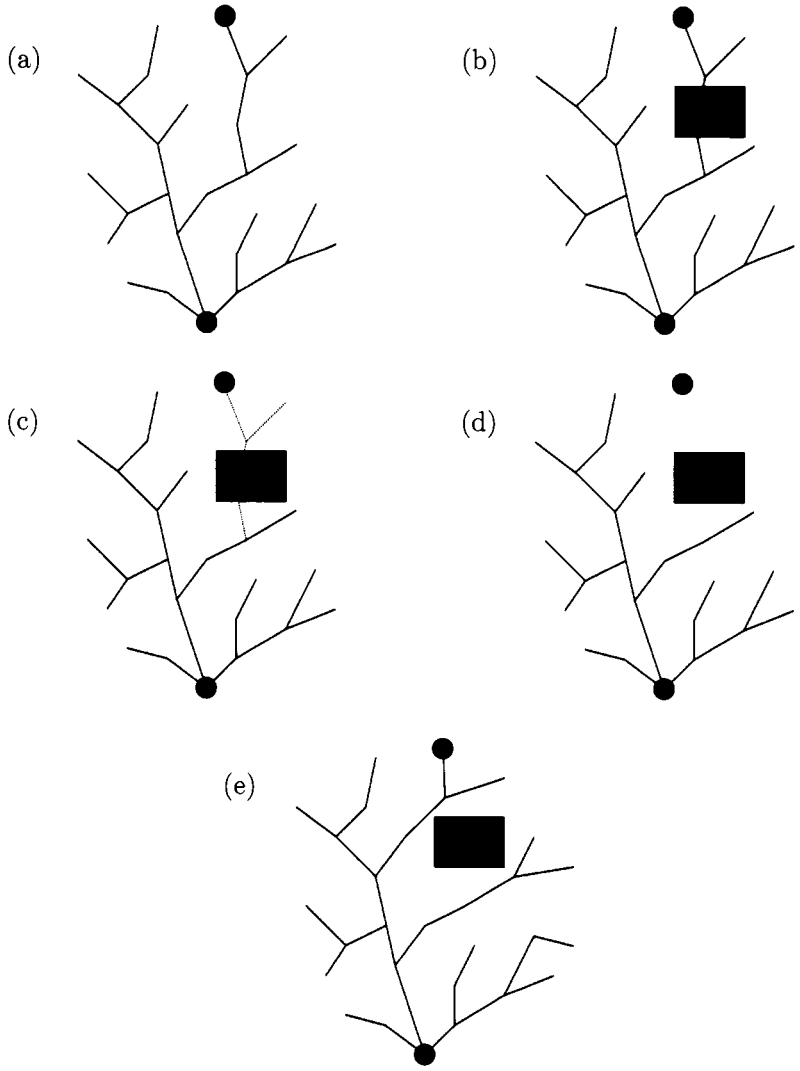


Figure 7.2: Replanning with RRTs. (a) An initial RRT generated from a start position (red) to a goal position (blue). (b) A new obstacle is added to the configuration space. (c) Parts of the previous tree that are invalidated by the new obstacle are marked. (d) The tree is trimmed: invalid parts are removed. (e) The trimmed tree is grown until a new solution is generated.

```
TrimRRT(rrt T)
 1  Q = ∅; i = 1;
 2  while (i < T.size())
 3    q_i = T.node(i); q_p = T.parent(q_i);
 4    if (q_p.flag = INVALID)
 5      q_i.flag = INVALID;
 6    if (q_i.flag ≠ INVALID)
 7      Q = Q ∪ {q_i};
 8    i = i + 1;
 9  T = CreateTreeFromNodes(Q);

RegrowRRT(rrt T)
10  TrimRRT(T);
11  GrowRRT(T);

InvalidateNodes(rrtT, obstacle o)
12  E = FindAffectedEdges(T, o);
13  for each edge e ∈ E
14    q_e = ChildEndpointNode(e);
15    q_e.flag = INVALID;

Main()
16  InitializeRRT(tree);
17  GrowRRT(tree);
18  forever
19    wait for changes to configuration space
20    for each new obstacle o
21      InvalidateNodes(tree, o);
22    if solution path contains an invalid node
23      RegrowRRT(tree);
```

Figure 7.3: The Dynamic Rapidly-exploring Random Tree Algorithm

Once the tree has been trimmed, it can be grown out to the goal. This can be performed in exactly the same manner as the basic RRT algorithm for initial construction. However, depending on how the configuration space has changed, it may be more efficient to focus the growth towards areas that have been affected, as discussed in the following section.

Dynamic RRTs can be used to provide a sampling-based analog of the D* algorithm for teams of mobile robots navigating imperfectly-known environments. To do this, we first reverse the direction of growth of the RRT so that it grows from the desired configuration towards the current robot configuration. Recall from Section 3.1 this is a common modification made by discrete replanning algorithms and allows us to reuse the previous tree when the current team configuration changes (as the team moves through the environment) and new obstacles appear. Otherwise, the root of the tree would constantly be changing

```
Main()
 1  q_start = q_goal; q_goal = q_initial;
 2  InitRRT(tree);
 3  GrowRRT(tree);
 4  while (q_goal ≠ q_start)
 5    q_goal = tree.parent(q_goal);
 6    Move to q_goal and check for new obstacles;
 7    if any new obstacles are observed
 8      for each new obstacle o
 9        InvalidateNodes(tree, o);
10      if solution path contains an invalid node
11        RegrowRRT(tree);
```

Figure 7.4: Using Dynamic RRTs to Interleave Planning and Execution

and so the entire tree would need to be re-grown. Further, since observations are typically being made in the vicinity of the team (through onboard sensors), this modification allows us to maintain more of the previous tree during repair, as only the tips of the tree will be affected by newly-observed obstacles.

In order to plan in a backwards direction with RRTs, the *Extend* function needs to generate actions in reverse. In other words, when extending the tree from a configuration $q_{nearest}$ to q_{target} (Figure 7.1 line 8), we generate a set of actions that have $q_{nearest}$ as their resulting configuration, rather than their initial configuration, and choose from this set the action that begins at the closest configuration to q_{target}. This is typically straightforward when dealing with reversible actions or fairly simple action sets.

Figure 7.4 presents pseudocode for using the Dynamic RRT algorithm to interleave planning and execution for mobile robot navigation in partially-known environments. The initial configuration of the agent is $q_{initial}$ and the desired goal configuration is q_{goal}. Note that this algorithm can be used for both single agents and multi-agent teams.

Application to Constrained Exploration

In Section 6.1 we showed how RRTs can be used to plan paths for teams of agents performing constrained exploration. However, this approach assumed that the environment was perfectly-known to begin with. When exploring real environments it is likely that only partial information will be available concerning the environment and so solutions will need to be updated to reflect new information received by the team during its traverse.

Because of its ability to efficiently repair previous solutions when changes to the configuration space are observed, the Dynamic RRT algorithm is well-suited to the task of constrained exploration in partially-known environments. To apply Dynamic RRTs to this

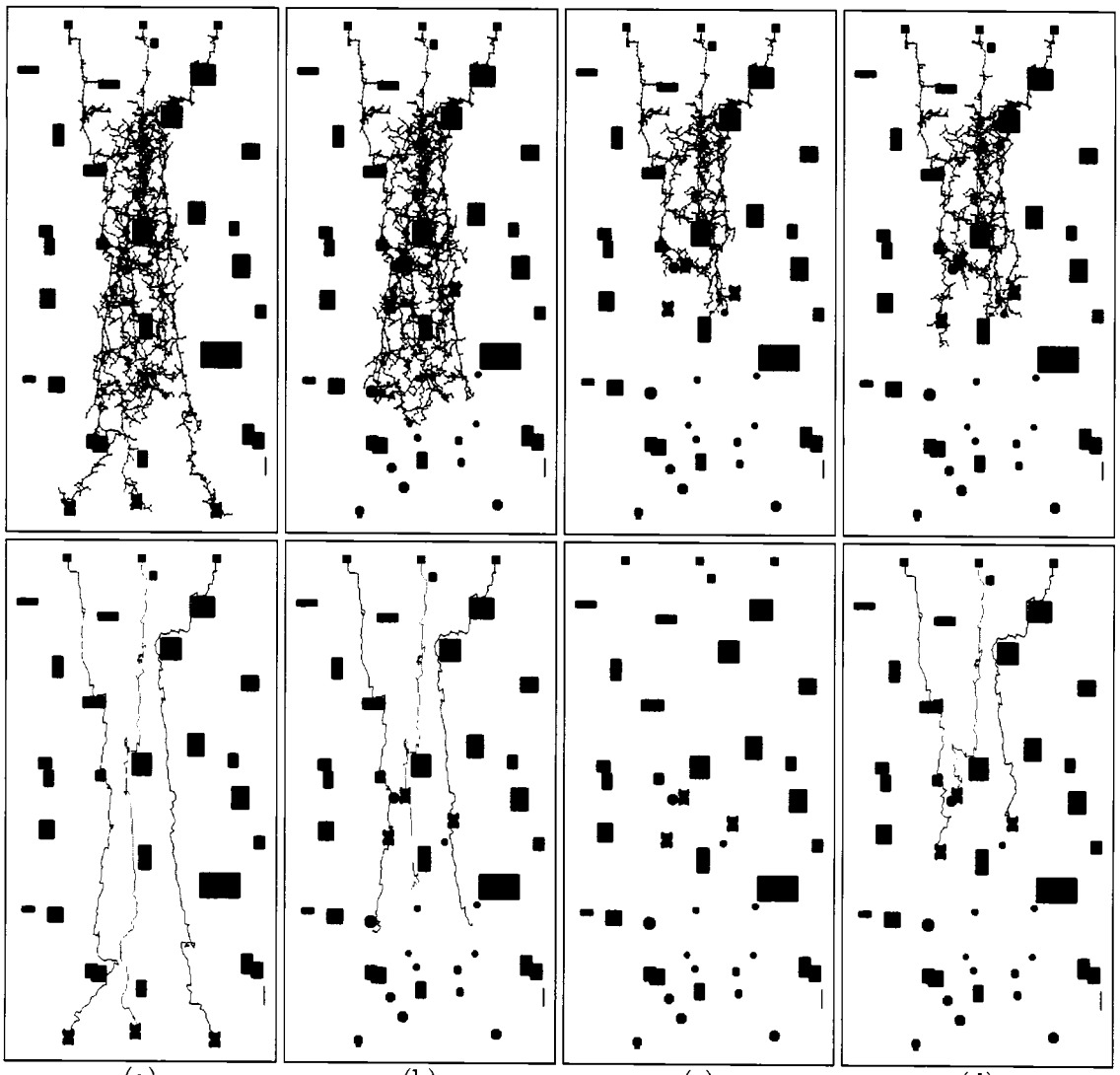


Figure 7.5: Applying Dynamic RRTs to the Constrained Exploration Problem. (a) The initial search tree (top) and solution path (bottom) planned for three agents navigating the environment while maintaining line-of-sight communication across the team at all times. Communication obstacles are shown in black. (b) As the agents traverse their paths, navigation obstacles are observed (in grey). At this point in the path, one of these obstacles (in front of the red agent) intersects the solution path. (c) The search tree is trimmed to remove the portions of the tree affected by the newly-observed navigation obstacles. (d) The remaining tree is re-grown to generate a new solution for the team.

problem, we grew the search tree in a backwards direction, from the desired goal configuration of the team towards its initial configuration. When new information was received, in the form of obstacles in the configuration space, we found and removed the affected portions of the search tree and re-grew the remaining portions to generate a new solution.

We concentrated on a version of the constrained exploration problem where the areas through which communication is not allowed (e.g. adversary areas) are known a priori, but the nature of the environment in terms of traversability is not. Thus, the team begins with an empty traversability map of the environment and as it navigates along its path, it observes the true nature of the environment and updates its traversability map to reflect newly-observed navigation obstacles. These obstacles have to be avoided by the team, and so the current solution path may need to be repaired to account for these obstacles. However, we assume the observed obstacles have no affect on communication (they can be thought of as rocks or ditches).

Figure 7.5 shows an example traverse by a team of three agents navigating from an initial team position at one end of an environment to a desired team position at the other. In this figure, the areas through which communication is forbidden appear in black, and navigation obstacles appear in grey as they are observed by the agents. To begin with, an initial RRT is grown from the goal configuration to the initial team configuration to provide an initial solution path for the team (Figure 7.5(a)). Next, the team begins to execute this solution, observing navigation obstacles as they progress through the environment. When one of these navigation obstacles intersects the current solution (Figure 7.5(b)), the affected nodes and edges in the current search tree are trimmed (Figure 7.5(c)) and the remaining, unaffected portions of the tree are grown to produce a new solution (Figure 7.5(d)). Because navigation obstacles are observed in the vicinity of the agents in the team, it is likely that they will affect portions of the current tree that are near the initial configuration. By directing the search backwards from the goal configuration to the initial configuration, rather than forwards, the amount of the tree that subsequently needs to be trimmed is reduced considerably.

Dynamic RRT Results

To test the performance of Dynamic RRTs against current approaches, we ran a number of experiments simulating a small team of robots performing constrained exploration in partially-known environments, as described above. As the robots moved through the environment, they received new information concerning the traversability of areas within some sensor field of view. If obstacles were encountered along their solution paths, a new RRT was generated that took into account the new obstacles. For comparison, this RRT was generated using both the ERRT approach and the Dynamic RRT approach.

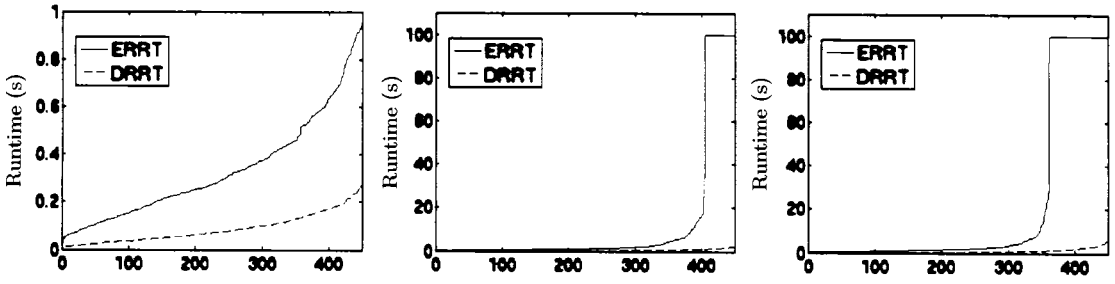


Figure 7.6: Runtimes for the ERRT (red) and Dynamic RRT (blue; labeled DRRT) approaches in our three constrained exploration scenarios. The x-axis of each graph represents the runs performed (each run is a full traverse through the environment). For each approach, the runs are ordered in terms of runtime.

To efficiently determine which edges in the RRT collided with a new obstacle (for the Dynamic RRT approach), for each (x, y) position in our discretized map we kept track of the nodes in the RRT that had one of the robots located at that position. Then, when a new obstacle appeared at location (x', y'), we could quickly find any nodes that were invalidated. To find edges that were invalidated, we simply carved a 2D circle out with center (x', y') and radius equal to the maximum length of any edge in the tree, then checked if any nodes of the tree resided within this circle *and* had an edge that intersected the new obstacle. Because the edges in our tree were typically quite small (on the order of 5 map cells in distance for each robot), this was very fast. The overhead of maintaining the list of nodes for each (x, y) position and determining which are affected when new obstacles are observed is included in our runtime comparisons.

We also focused re-growth of the Dynamic RRT to areas that had been affected by changes to the configuration space. Specifically, with some probability we chose a random sample point in the vicinity of the recently-affected area of the configuration space. This has the effect of focusing growth of the tree in the region just trimmed, which is also close to the current configuration of the team. In our results, this probability was set equal to the probability used by the ERRT approach to select a previous way-point, which was 0.4. Each approach selected a sample very close to q_{goal} with probability 0.1. We found these parameters gave the best performance for both the ERRT and Dynamic RRT approaches.

Results for teams ranging from a single robot to three robots are shown in Fig. 7.6. For each case, the task was to navigate across a 300 $\times$ 600 environment containing randomly generated obstacles. For the single robot case, as the robot traversed the environment, new obstacles appeared at random: with each step the robot took along its path there was some chance a randomly placed obstacle would appear within the robot's sensor range (25

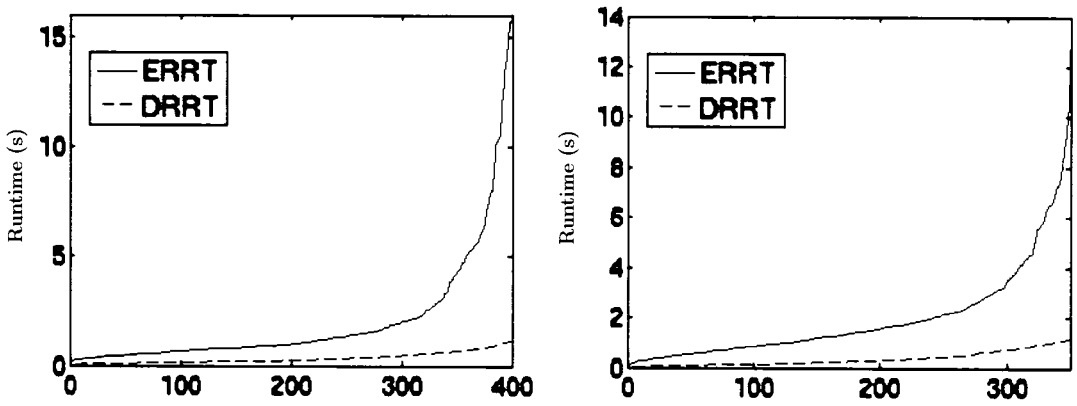


Figure 7.7: Runtimes for the ERRT (red) and Dynamic RRT (blue; labeled DRRT) approaches for the quickest 400 and 350 runs for our two-robot and three-robot teams, respectively. These graphs highlight a portion of the results shown in Figure 7.6.

cells), potentially requiring that the robot repair its previous solution path. We repeated the process 5 times each for 100 different environments.

To ensure each approach was fairly compared, we had them operate over exactly the same traverses. To do this, we ran the Dynamic RRT approach first and recorded the path taken and new obstacles encountered, and then had the ERRT approach operate over the same path and obstacles. Since the agent did not move very far in between replanning episodes, this did not place the ERRT approach at a disadvantage. We also used kd-trees to efficiently compute nearest neighbors in both approaches.

For the multirobot cases, the situation was similar, except that the robots had to maintain line-of-sight communication between team members. The team started with a map specifying the areas of the environment through which communication was not possible. These areas acted as line-of-sight connectivity obstacles: if a straight-line connecting two robots passed through one of these areas, the robots were not within direct line-of-sight communication of each other. Further, as the robots moved, they observed navigation obstacles that they had to avoid but which did not affect communication. These multirobot experiments were designed to simulate the constrained exploration task, where the areas of the environment through which communication is not possible (e.g. adversary areas) are known a priori, but the nature of the environment (e.g. in terms of terrain or navigability) is not.

Navigation obstacles also appeared at random in the multirobot experiments: for the two-robot case an obstacle would appear within each robot's sensor range with probability 0.4 and for the three-robot case an obstacle would appear in front of each robot's sensor range with probability 0.15. Again, one hundred different random environments were used

Number of robots	Nodes added by ERRTs	Nodes added by Dynamic RRTs
1	5443	933
2	14299	2232
3	12855	2692

Table 7.1: The average total number of nodes added to the search trees over the course of each traverse, for each team size.

with each traversed 5 times. Because of their use of randomness, the same problem may take RRTs vastly different amounts of computation depending on the random values generated. Thus, in order to see clearly the relative performance of the algorithms, we have shown the most efficient 90% of the 500 runs for each approach (and, for the same reason, we have limited the runtime of each run to 100 seconds[1]). We have also included, in Figure 7.7, the most efficient 400 and 350 runs for the two-robot and three-robot teams, respectively, to show more clearly the difference in performance of the two approaches. All the runtimes reported are for a 1.5 GHz Powerbook G4.

On average, Dynamic RRTs outperformed ERRTs in each scenario by a factor of 5, in both the average number of nodes added to the tree during each traverse and in the computation time required. Table 7.1 presents the average number of nodes added to the search tree during each traverse, for both ERRTs and Dynamic RRTs. The average time spent per replanning episode for the three robot team was 146 milliseconds for ERRTs and 27 milliseconds for Dynamic RRTs.

There are a number of extensions that can be made to the basic Dynamic RRT algorithm. As discussed earlier, for Dynamic RRTs to be effective for mobile robot navigation, we need to direct the tree growth backwards from the goal configuration to the initial configuration. However, if we want to model the complex kinematics of the robot(s), then this may not be possible. In discrete planning, usually this is performed by combining an approximate global backwards-searching path planner such as D* with a local forwards-searching planner that incorporates the kinematic constraints of the vehicle (see Section 2.1). Using Dynamic RRTs, we could address this issue by augmenting this backwards-directed global RRT with a forwards-directed local RRT that encodes all the kinematic and dynamic constraints of the vehicle. Then, when new information is received while traversing through the environment, the global RRT can be repaired using Dynamic RRT ideas, and a new local RRT can be grown to the global RRT. This is analogous to how bi-directional RRTs are used for planning [Kuffner and LaValle, 2000], but the idea is to use different granularities in our RRTs so that we achieve the same benefits as the coupling of discrete local arc-based planners with

[1]We also limited the number of nodes added to any single tree to be 30000. If no solution was found within this limit we marked the run as a failure and set the time to be the default maximum of 100 seconds.

discrete global planners in the mobile robot navigation realm. Some other useful extensions to Dynamic RRTs will be described in the next two chapters.

7.3 Discussion

In this chapter, we have presented Dynamic Rapidly-exploring Random Trees (Dynamic RRTs), a sampling-based replanning algorithm that efficiently repairs its solution when changes are made to the configuration space. Like the RRT algorithm, Dynamic RRTs initially grow a search tree out through the configuration space to generate a solution. However, when the configuration space is altered and this solution becomes invalid, the Dynamic RRT algorithm efficiently removes just the newly-invalid parts of the search tree and re-grows a solution from what remains. We have demonstrated its effectiveness for single agent and multi-agent path planning in partially-known environments.

The basic Dynamic RRT approach can be thought of as a single-tree variant of the RRF approach described in Section 7.1. However, by only maintaining a single tree, Dynamic RRTs offer a number of advantages in regards to the problem of multi-agent path planning and replanning. Firstly, Dynamic RRTs are simple to implement and involve very few parameters. Because there is only one tree, we need not worry about which tree to grow next, when to try to connect different trees together, how to trim the different trees, and so on. Dynamic RRTs clearly extend basic single-tree RRTs to partially-known or dynamic environments.

Secondly, when edges are invalidated in the tree, Dynamic RRTs remove the entire affected branch rather than using the branch to create a new tree. This can be beneficial in navigation scenarios as often either these branches are very small (since changes are typically taking place in the vicinity of the robot(s) and so near the leaves of the tree), or the branches extend out into areas of the configuration space already passed by the robot(s) and no longer useful for planning. Further, by using biased sampling to focus towards recently invalidated edges, parts of the pruned branches that may have been useful will be re-grown quickly by the Dynamic RRT.

We have shown how Dynamic RRTs can be used to provide a sampling-based analog of the widely-used D* family of discrete replanning algorithms. To our knowledge, this is the first effort to incorporate the principles behind D* into sampling-based algorithms. Our results demonstrate the usefulness of this combination; consequently, we believe Dynamic RRTs are a good substitute for D* in high-dimensional problems such as multi-agent constrained exploration.

Chapter 8

Multi-agent Planning with Limited Deliberation Time

The efficiency of RRTs is due to their ability to explore vast regions of the configuration space without dealing with all possible configurations. In graph-search terminology, RRTs are akin to a best-first search, where the heuristic function used to select which state to expand next is simply the distance to a (continuously changing) randomly selected location in the configuration space. This can be very efficient; however, it also means that the resulting solution may be highly suboptimal.

In many situations, in particular when dealing with non-uniform cost configuration spaces, the difficulty of executing different solution paths may vary significantly. It is thus important that solution cost is taken into account during the search process. With multiple-query sampling-based algorithms like PRMs it is possible to improve the quality of the solutions by increasing the density of the roadmap. However, this can be very costly in terms of both memory and computation, and is really only worth it if we are interested in performing multiple queries. For single-shot planning problems, what we need is a method for improving the quality of solutions generated by single-shot planning algorithms such as RRTs, and ideally a method that is efficient in terms of both memory and computation.

Nevertheless, few efforts have been made to incorporate cost considerations into RRTs in order to produce better solutions. One notable exception is work by Urmson and Simmons [Urmson and Simmons, 2003], who developed a series of modified versions of the RRT algorithm that select tree nodes for expansion based on the cost of their current path from the initial node. Their approaches were able to produce less costly solutions through both uniform and non-uniform cost configuration spaces. However, as might be imagined, this improvement in solution quality usually came at a computational price: the approach that produced the best solutions required significantly more computation than the unmodified RRT algorithm.

For teams of agents operating in the real world under time constraints, it is important

both that high quality solutions are produced and that these solutions can be produced within the time available for planning. As mentioned in Chapter 4, anytime approaches are particularly suited to this task, as they quickly find an initial, highly suboptimal plan, and then improve this plan until time runs out.

When deliberation time is limited, the configuration space is very high-dimensional, *and* the team is receiving updated environment information, it is important to be able to not only improve the quality of the current solution but also repair the solution when new information arrives that invalidates it.

In this chapter, we present two algorithms for addressing these scenarios. We begin by introducing extensions to the RRT algorithm that improve the quality of the solutions returned. We then introduce an anytime framework, called *Anytime RRTs*, for providing solutions very quickly, then improving these solutions while deliberation time allows. We then combine this with our sampling-based replanning algorithm Dynamic RRTs presented in the previous chapter. The result is *Anytime Dynamic RRTs*, a sampling-based anytime, replanning algorithm suited to high-dimensional planning in non-uniform cost, partially-known environments. We present results for both of these algorithms applied to the constrained exploration problem, along with an implementation on a team of outdoor vehicles. We conclude with discussions.

8.1 Improving the Solution Quality of RRTs

As mentioned earlier, the original RRT algorithm does not take into account solution cost during its search. Thus, it can produce paths that are grossly suboptimal, particularly in non-uniform cost search spaces. To improve upon this, Urmson and Simmons presented three modified versions of the RRT algorithm that take cost into account when selecting nodes in the tree for extension [Urmson and Simmons, 2003]. They replaced the simple *NearestNeighbor* function (Figure 2.8 line 5) with a function that found the k nearest neighbors to the current q_{target} point, then selected from these k nodes either (1) the closest node $q_{nearest}$ to q_{target}, as long as an estimate of the cost of a path from q_{start} through $q_{nearest}$ to q_{goal} is less than some probabilistic threshold r, or (2) the first of the k nodes (ordered by estimated path cost) whose current estimated path cost is less than r, or (3) the node with the minimum estimated path cost, as long as this cost was less than r. These 3 different selection methods resulted in 3 different algorithms.

To provide a better understanding of the approach used by Urmson and Simmons, we have included pseudocode of their second selection method, known as the *Iterative k-Nearest Neighbor RRT* algorithm, in Figure 8.1. This approach selects a node in the tree to expand based on its *Quality* measure, which is an indication of how likely the node is to reside on a low-cost solution path. To compute this value, an estimate of the cost of a path through this

```
Quality(configuration q, rrt T)
 1  v = 1 - (EstimatedPathCost(q) - T.optcost) / (T.maxcost - T.optcost);
 2  if (v < T.floorprob)
 3    v = T.floorprob;
 4  return v;

ChooseTargetAndTreeNodes(rrt T)
 5  forever
 6    q_target = ChooseTarget();
 7    Q_near = kNearestNeighbors(q_target, k, T);
 8    while Q_near is not empty
 9      remove configuration q_tree with maximum Quality(q_tree) from Q_near
10      p = RandomReal([0.0, 1.0]);
11      if (p < Quality(q_tree))
12        return {q_target, q_tree};

GrowRRT(rrt T)
13  q_new = q_start;
14  T.optcost = EstimatedOptimalPathCost(q_start, q_goal);
15  while (Distance(q_new, q_goal) > distance-threshold)
16    {q_target, q_tree} = ChooseTargetAndTreeNodes(T);
17    q_new = Extend(q_tree, q_target, T);
18    if (q_new ≠null)
19      T.add(q_new)
20      if (EstimatedPathCost(q_new) < T.maxcost)
21        T.maxcost = EstimatedPathCost(q_new);
```

Figure 8.1: The Iterative k-Nearest Neighbor RRT Algorithm.

node is computed, consisting of the current cost through the tree to the node combined with an estimate of the cost of a path from the node to the goal configuration (this combination makes up the *EstimatedPathCost* function). Next, this estimate is used to come up with a value, between 0 and 1, of the relative value of the node in the tree. This is computed by comparing the estimated path cost through the node to the (estimated) cost of an optimal path and the most expensive path cost associated with any node already in the tree (line 1). The higher this resulting value, the more promising the node. The value is forced to be at least as great as some minimum threshold ($T.floorprob$) to allow for some exploration to occur (line 3).

Once this quality value has been computed, a random number between 0 and 1 is generated and if this number is less than the quality value, the current node q is returned as the next node in the tree to extend. If this node can be extended, the extended configuration q_{new} is added to the tree. If the estimated path cost of q_{new} is the largest of any node

currently in the tree, it is stored to be used for computing relative *Quality* values of other nodes, as mentioned above (lines 20 and 21).

By taking into account the cost associated with reaching nodes in the tree from q_{start} and combining this with an estimate of the cost of reaching q_{goal}, Urmson and Simmons' modified RRT algorithms are able to produce less-costly solutions than the original RRT algorithm. Of their three different selection methods, the third method produced the best overall solutions, but, perhaps unsurprisingly, it required significantly more computation time than the RRT algorithm.

We are interested in using RRTs for the navigation of multi-agent teams in partially-known outdoor environments. Because our agents are acting in the real world, there may be situations where plans must be generated and executed extremely quickly. There may also be other situations where there is more time for deliberation and better plans are desired. The agents may not know a priori how much planning time is available and they certainly will not know how long it will take a particular algorithm to generate a solution. Thus, it is useful to generate a series of solutions and then employ the best of these solutions when an action has to be executed.

As discussed in Chapter 4, there exist a number of efficient discrete algorithms that can provide this performance [Likhachev et al., 2003, Zhou and Hansen, 2002]. For example, the ARA* algorithm quickly creates an initial, highly-suboptimal solution by running an inflated A* search with a high suboptimality bound ϵ, then improves this solution by repeatedly running new searches with decreasing values of ϵ [Likhachev et al., 2003]. After each search terminates, the cost of the most recent solution is guaranteed to be at most ϵ times the cost of an optimal solution.

In order to plan through complex, very high-dimensional configuration spaces (as encountered in the constrained exploration task), we would like an RRT-based analog of these discrete anytime algorithms. In the following section, we present a sampling-based anytime algorithm that efficiently constructs an initial solution using the standard RRT algorithm, then improves this result by generating a series of solutions, each guaranteed to be better than all the previous ones by some improvement factor ϵ_f. These successive solutions are generated by using modified versions of the RRT algorithm that take into account cost—both of nodes in the current tree and of previous solutions—to influence the sampling of the search space, the selection of nodes in the tree for extension, and the extension operation itself.

8.2 Anytime Rapidly-exploring Random Trees

Discrete anytime algorithms such as ARA* achieve their anytime performance by efficiently generating a series of solutions, where each solution is better than the previous ones. At any

point in time the best solution found thus far can be returned, along with a suboptimality bound on the quality of this solution.

We can use this same basic idea to create a sampling-based anytime algorithm. We start out by generating an initial RRT without any cost considerations. We then record the cost C_s of the solution returned by this RRT. Next, we generate a new RRT and ensure that its solution is better than the previous solution by limiting the nodes added to the tree to only those that could possibly contribute to a solution with a lower overall cost than C_s. We can also multiply our cost bound C_s by some factor $(1 - \epsilon_f)$, where $0 \leq \epsilon_f < 1$, to ensure that the next solution will be at least ϵ_f times less expensive than our previous solution. In such a case, ϵ_f is known as the solution improvement factor. We then update our cost bound C_s based on the cost of the new solution and repeat the process until time for planning runs out.

Such an approach guarantees that each solution produced will be better than all the previous solutions. However, it does not guarantee that new solutions will be able to be produced. In order for this approach to be truly effective we require dependable methods for generating each successive low-cost solution. We rely upon novel node sampling, node selection, and node extension operations that incorporate cost considerations and variable bias factors to efficiently produce solutions satisfying a specified cost bound. We discuss each of these operations in turn.

Node Sampling

If we are only interested in generating a solution that is cheaper than some upper bound value C_s, then we can use this upper bound to influence the sampling process used by the RRT algorithm. Rather than randomly sampling the entire configuration space, we restrict our sampling to just those areas of the configuration space that could possible provide a solution satisfying the upper bound. Given a node q_{target} in the configuration space, we can check whether q_{target} could be part of such a solution by calculating a heuristic cost from the initial node q_{start} to q_{target}, $h(q_{start}, q_{target})$, as well as a heuristic cost from q_{target} to the goal node q_{goal}, $h(q_{target}, q_{goal})$. If these heuristic values do not overestimate the costs of optimal paths between these nodes then the combination of these heuristic values gives us a lower bound on the cost of any path from q_{start} through q_{target} to q_{goal}. If this lower bound cost is greater than our upper bound C_s, then there is no way q_{target} could be part of a solution satisfying our upper bound and so q_{target} can be ignored.

However, depending on the complexity of the configuration space and what heuristics are used, this approach could make it very difficult for the RRT to find a solution. For example, if there are narrow passages in the configuration space between large obstacles, then it may be very difficult to sample nodes inside the passages. This is a well-known

problem with the original RRT algorithm, but it could be exacerbated by disregarding any samples that fall inside configuration space obstacles. Depending on how the heuristic deals with such samples, the above approach could make it even more difficult for the tree to grow down any narrow passages. Further, by reducing our consideration of sample points to only those whose heuristic values are promising, we are in effect cutting off large chunks of the configuration space. This is entirely the point of restricting our sampling, but it can also introduce new narrow passages. For example, imagine an obstacle that resides near the edge of the promising configurations, as determined by our heuristic values. It may be possible to plan a path around this obstacle, but finding such a path may be difficult using our restricted sampling approach, as very few samples exist that will pull the tree towards this edge. Thus, it is important to use conservative heuristic estimates and not disregard points that reside in configuration space obstacles.

One way of implementing this restricted sampling idea is to continue generating random samples q_{target} from the configuration space until we find one whose combined heuristic cost is less than our upper bound. This approach is illustrated in Figure 8.2(b). Another method is to use the heuristic functions to do the sampling itself, so that every node sampled will satisfy the upper bound. On the whole, this restricted sampling technique saves us a lot of unnecessary computation spent on irrelevant areas of the configuration space.

Node Selection

Once a sample node q_{target} has been generated using the above technique, we then select a node from the tree q_{tree} to extend out towards the sample node. In the original RRT algorithm, the closest node in the tree to q_{target} is selected to be q_{tree}. However, as Urmson and Simmons show [Urmson and Simmons, 2003], much cheaper solutions can be obtained if we modify this selection process to incorporate cost considerations.

Our selection approach is based on their ideas but uses bias factors to vary over time the influence of cost, so that initially nodes are selected based purely on their distance from q_{target}, and in subsequent searches this gradually changes so that some combination of distance from q_{target} and the cost of the node is considered, eventually leading to purely cost-based selection. We accomplish this by using a distance bias parameter d_b and a cost bias parameter c_b. First, the k nearest neighbor nodes in the tree to q_{target} are computed. Next, these nodes are ordered in increasing node selection cost, where

$$\textbf{NodeSelectionCost}(q) = d_b \cdot \textbf{Distance}(q, q_{target}) + c_b \cdot c(q_{start}, q),$$

where $c(q_{start}, q)$ is the cost of the current path from q_{start} to q in the tree. We then process these nodes in order until one is found from which a valid extension can be made (the extension process is described below). Figure 8.2(c) shows an illustration of the node

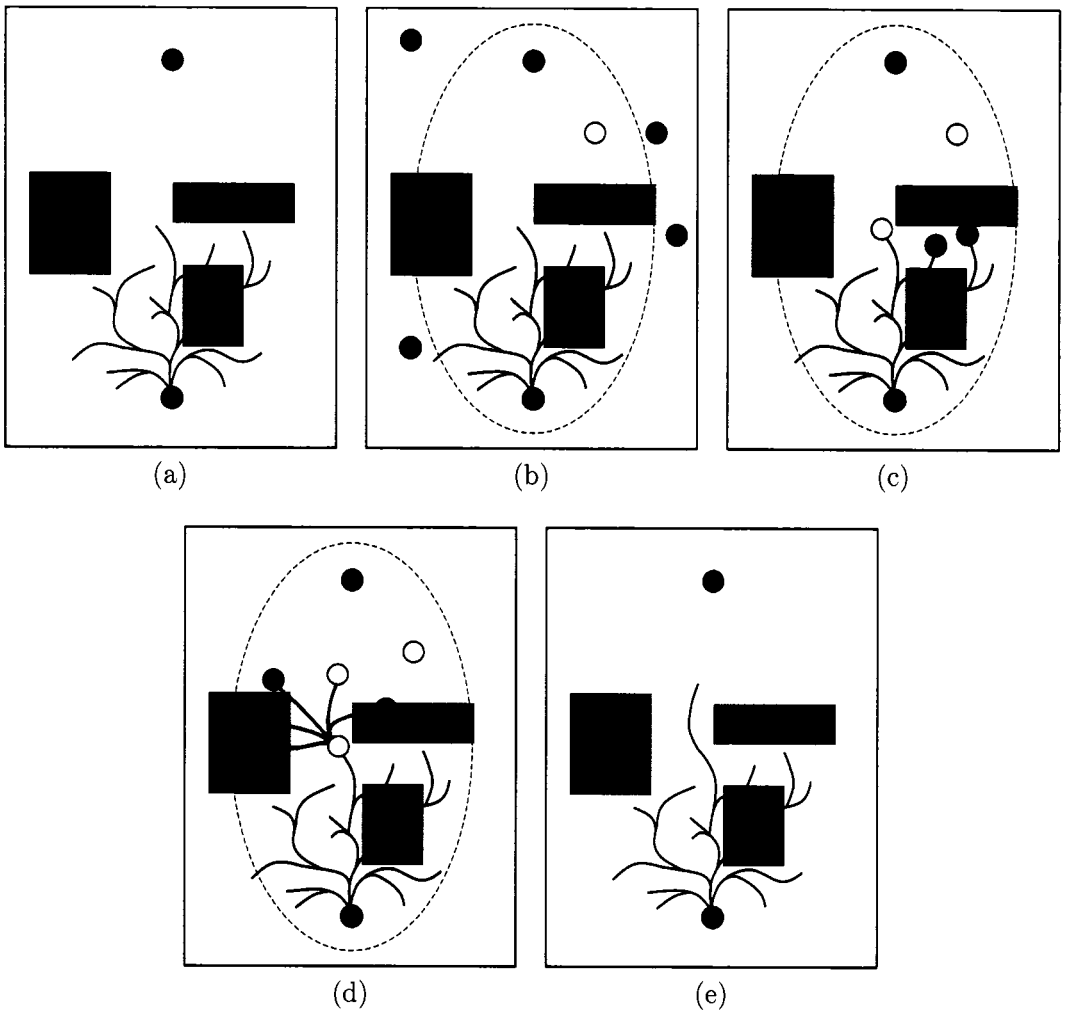


Figure 8.2: Anytime RRT Planning. Given a partial RRT (shown in (a)) being grown from an initial configuration (the bottom filled circle) to a goal configuration (the top filled circle), this illustration shows how the Anytime RRT approach samples, selects, and extends the tree. To begin with, we assume some previous solution has already been generated. (b) When using the Anytime RRT approach to sample the configuration space, only areas that could potentially lead to an improved solution are considered (indicated by the shaded oval). Thus, the black nodes are rejected while the white node is accepted. (c) When selecting the next node in the tree to extend, the k nodes closest to the sample point are found and ordered according to both their distance from the sample point and the cost of their path from the start node. Here, $k = 3$ and the white node and two black nodes are the closest; the white node is selected first since its path is less expensive than those of the two black nodes. (d) The tree node from (c) is then extended by generating a set of possible extensions and then probabilistically choosing the one that is least expensive. The cost of the extension to the white node is cheaper than the extensions to the black nodes, so the white node is chosen as the next element to be added to the tree. (e) After checking that the sum of the cost of the path from the start node through the tree to the new element and the heuristic cost of a path from the new element to the goal is less than the solution bound, the element is added to the tree.

selection process.

Initially, $d_b = 1$ and $c_b = 0$ and our node selection operates exactly as a nearest neighbor lookup. Between each successful search, d_b is reduced by some value δ_d and c_b is increased by some value δ_c, so that the cost of the tree nodes becomes increasingly important as time progresses. This has the effect of producing more costly solutions early on, when we are most concerned with ensuring that valid solutions are available, and then providing cheaper solutions if there is extra time available for planning.

Node Extension

When a node in the tree is selected for extension, we take into account the nature of the configuration space in its vicinity to produce a new, low-cost branch of the tree. There are two different approaches we use to do this. The first approach is to generate a set of possible extensions that lead from the tree node q_{tree} in the general direction of the sample node q_{target} and then take the cheapest of these extensions. This results in a new node q_{new} which is added to the tree if it could potentially contribute to a solution satisfying our upper bound C_s, i.e., if

$$c(q_{start}, q_{tree}) + c(q_{tree}, q_{new}) + h(q_{new}, q_{goal}) \leq C_s,$$

where $c(q_{start}, q_{tree})$ is the cost of the current path from q_{start} to q_{tree} in the tree (as before), and $c(q_{tree}, q_{new})$ is the cost of the extension just constructed from q_{tree} to q_{new}. If q_{new} does not satisfy our solution bound then a new set of extensions that do not lead as directly to the sample node is considered. This process continues, with each subsequent set of extensions 'fanning out' further from the sample node, until either a q_{new} is generated that satisfies our bound or some maximum number of attempts have been made.

The second approach is to use a large initial set of possible extensions and take the cheapest of these extensions. Again, we check that the corresponding new node q_{new} satisfies our solution bound before adding it to the tree. This approach is illustrated in Figure 8.2(d). Because the first of these approaches is generally faster and the second produces less costly solutions, both are useful in our anytime framework: the first can be used to efficiently produce RRTs at early stages of our planning, while the second can be used to produce later trees with less costly solutions. It can also be beneficial to include some randomness in the extension operation: with some probability a random extension operation is selected and tested against our solution bound.

Pseudocode of the basic Anytime RRT approach is given in Figures 8.3 and 8.4. For space and simplicity we have not included all the features mentioned in the previous discussion. In particular, our actual implementation begins by generating a standard RRT and uses this to provide an initial solution and bound, and the *ExtendToTarget* function is mod-

```
ReinitializeRRT(rrt T)
 1 T.cleartree();
 2 T.add(q_start);
GrowRRT(rrt T)
 3 q_new = q_start; time = 0;
 4 while (Distance(q_new, q_goal) > distance-threshold)
 5   q_target = ChooseTarget(T);
 6   if (q_target ≠ null)
 7     q_new = ExtendToTarget(q_target, T);
 8     if (q_new ≠ null)
 9       T.add(q_new);
10   UpdateTime(time);
11   if (time > max-time-per-rrt)
12     return null;
13 return T.c(q_start, q_new);
ChooseTarget(rrt T)
14 p = RandomReal([0.0, 1.0]);
15 if (p < goal-sampling-prob)
16   return q_goal;
17 else
18   q_new = RandomConfiguration();
19   attempts = 0;
20   while (h(q_start, q_new) + h(q_new, q_goal) > T.C_s)
21     q_new = RandomConfiguration();
22     attempts = attempts + 1;
23     if (attempts > max-sample-attempts)
24       return null;
25   return q_new;
```

Figure 8.3: The Anytime RRT Algorithm: GrowRRT and ChooseTarget functions

ified over time to switch between our different extension approaches. In the pseudocode, *UpdateTime*(*time*) increments an elapsed time counter, *kNearestNeighbors*(q_{target}, k, T) returns the k closest nodes in the tree T to a sample node q_{target}, *GenerateExtensions* generates a set of extension operations (in the manner discussed earlier) and returns the nodes at the endpoints of these extensions, and *PostCurrentSolution* publishes the current solution so that it can be executed if deliberation time runs out. The plaintext identifiers (e.g. 'num-neighbors') are constants. Other identifiers (e.g. d_b) are as described earlier, prefixed with the tree identifier T to illustrate that they are variables associated with the tree[1].

[1]We will be dealing with multiple trees in later sections, at which point such associations will be important.

```
NodeSelectionCost(rrt T, configuration q, configuration q_target)
  1 return T.d_b · Distance(q, q_target) + T.c_b · T.c(q_start, q);
ExtendToTarget(rrt T, configuration q_target)
  2 Q_near = kNearestNeighbors(q_target, k, T);
  3 while Q_near is not empty
  4   remove configuration q_tree with minimum NodeSelectionCost(T, q_tree, q_target) from Q_near;
  5   Q_ext = GenerateExtensions(q_tree, q_target);
  6   q_new = argmin_{q∈Q_ext} c(q_tree, q);
  7   T.c(q_start, q_new) = T.c(q_start, q_tree) + c(q_tree, q_new);
  8   if (T.c(q_start, q_new) + h(q_new, q_goal) ≤ T.C_s)
  9     return q_new;
 10 return null;
Main()
 11 T.d_b = 1; T.c_b = 0; T.C_s = ∞;
 12 forever
 13   ReinitializeRRT(T);
 14   T.C_n = GrowRRT(T);
 15   if (T.C_n ≠ null)
 16     PostCurrentSolution(T);
 17     T.C_s = (1 − ε_f) · T.C_n;
 18     T.d_b = T.d_b − δ_d;
 19     if (T.d_b < 0)
 20       T.d_b = 0;
 21     T.c_b = T.c_b + δ_c;
 22     if (T.c_b > 1)
 23       T.c_b = 1;
```

Figure 8.4: The Anytime RRT Algorithm: ExtendToTarget and Main functions

Because the Anytime RRT approach uses a solution bound to influence the growth of each RRT, it has a number of nice properties. We include the major ones here; proofs of these can be found in the appendix. Firstly, because the algorithm restricts nodes added to the tree based on the solution bound $\mathcal{C}_s$, it is guaranteed not to produce any solution whose cost is greater than $\mathcal{C}_s$.

Theorem 6. *When the solution bound is $\mathcal{C}_s$, the cost of any solution generated by the Anytime RRT algorithm will be less than or equal to $\mathcal{C}_s$.*

By updating the solution bound each time a new path is generated, the algorithm is also able to guarantee that the next solution will be better than the previous one, by at least our user-defined solution improvement factor ϵ_f.

Theorem 7. *Each time a solution is posted by the Anytime RRT algorithm, the cost of this solution will be at most $(1 - \epsilon_f)$ times the cost of the previous solution posted.*

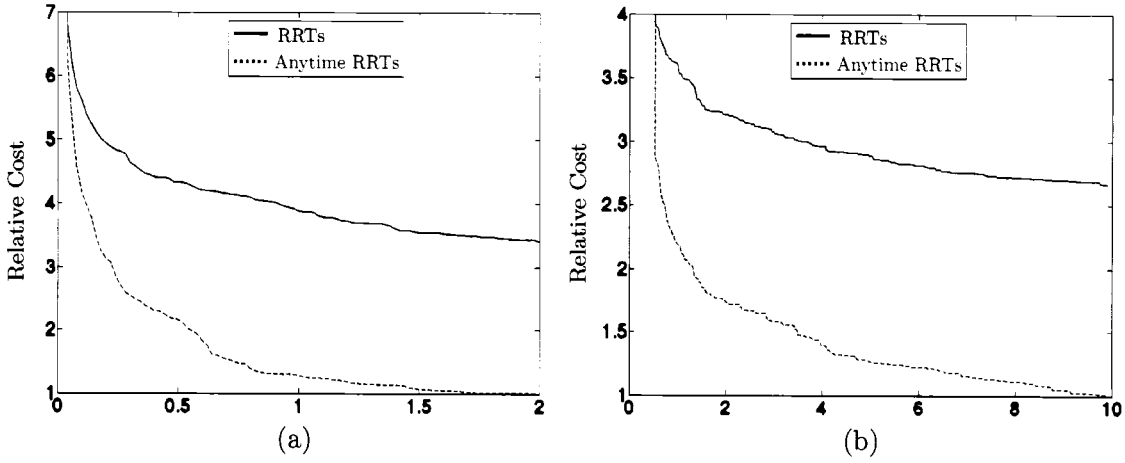


Figure 8.5: Anytime RRT results. Average relative solution cost as a function of time for our (a) single robot, and (b) three robot runs. The minimum-cost solution generated by each of the approaches (relative to the best overall solution generated) is presented at each point in time.

Together, these properties ensure that solutions produced by the Anytime RRT algorithm improve over time and that the rate of improvement is at least as good as the solution improvement factor ϵ_f.

Corollary 1. *If $\epsilon_f > 0$ then the solutions posted by the Anytime RRT algorithm will have associated costs that are strictly decreasing. Moreover, these costs will decrease by at least a factor of ϵ_f between each successive solution.*

Anytime RRT Results

The primary motivation behind this algorithm was efficient multirobot path planning in both uniform and non-uniform cost environments. In particular, we are interested in the constrained exploration task in environments where there may be areas that we would like the team to avoid for various reasons (e.g. traversal risk, stealth, visibility, etc).

To analyse the performance of Anytime RRTs for both single and multi-agent planning, we generated two different sets of experiments. In the first, we planned paths for a single agent across a 300×600 non-uniform cost environment in which there was a set of randomly placed obstacles and a set of randomly placed, random cost areas (see Figure 8.6 for an example such environment). We compared the solutions generated by Anytime RRTs to those generated by a series of standard RRTs. Each approach was allowed to run for a total time of 2 seconds (on a 1.5 GHz Powerbook G4), and each individual RRT was allowed to take up to 0.5 seconds. We repeated this process for 100 different environments. Figure 8.5(a) plots the average cost of the best solutions generated versus planning time for these

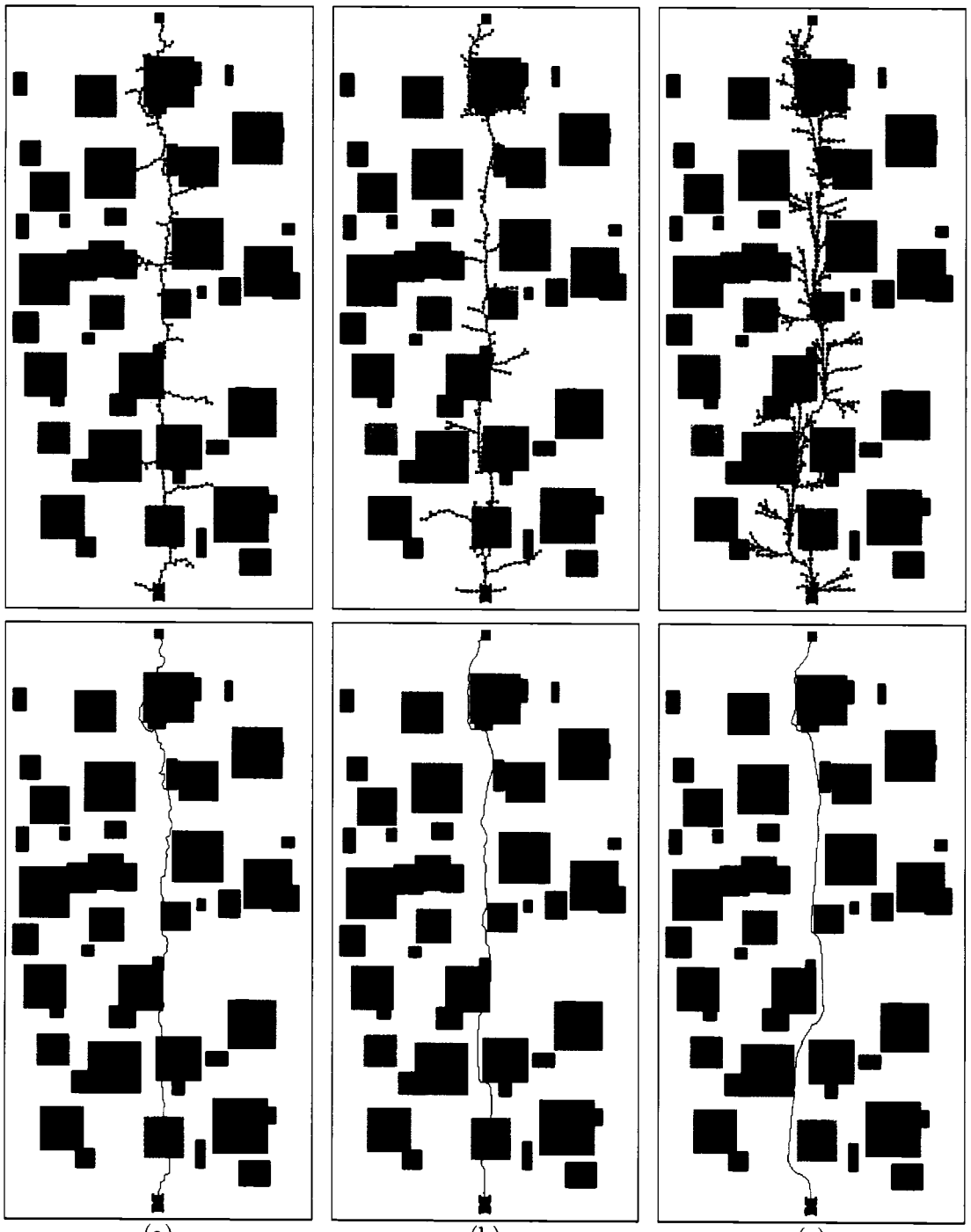


Figure 8.6: Anytime RRTs used for single robot path planning. The initial configuration is shown by the vehicle at the bottom of the environment, the goal is the square at the top. Shaded regions represent higher-cost areas to traverse through; black regions represent obstacles. (a) Initial RRT generated without cost consideration. The top image shows the tree while the bottom image shows the solution path. (b) Fifth RRT generated, using costs of previous solutions and nodes of the current tree to bias the growth of the current tree. (c) Final RRT generated. These images correspond to the results presented in Figure 8.5(a).

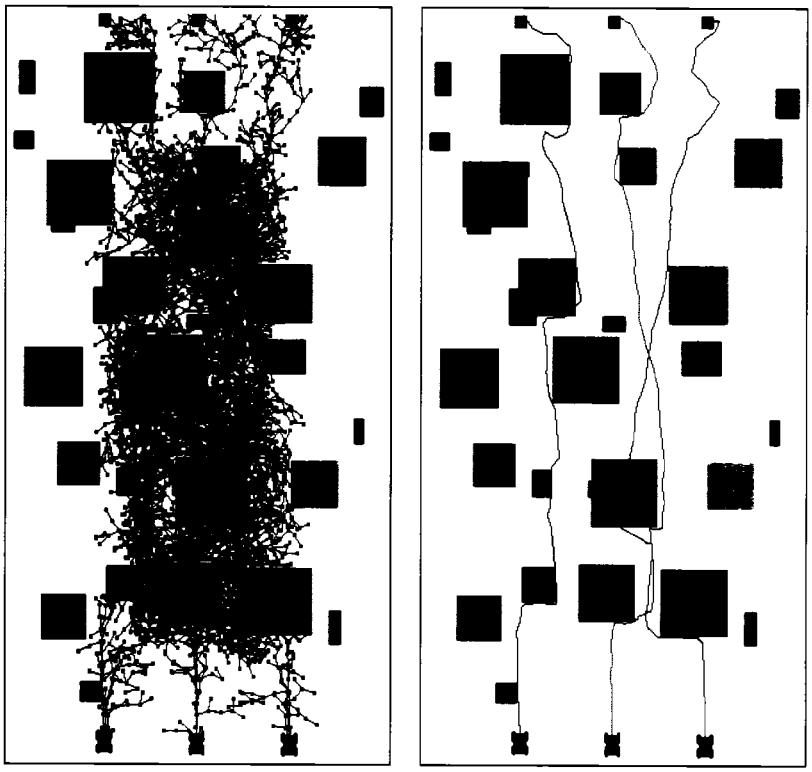


Figure 8.7: Anytime RRTs used for multirobot constrained exploration. On the left is one of the trees produced by our anytime approach. On the right is the corresponding path.

experiments. These results have been normalized: for each run, the best solution produced by the Anytime RRT approach was recorded and all solution costs were then computed relative to this cost.

Our second set of experiments had a similar setup, except that we planned joint paths for a team of 3 agents performing constrained exploration. Again, we randomly generated 100 different non-uniform cost environments, and the obstacles in the environment acted as both navigation obstacles (i.e., no agent could have its path intersect any of these obstacles) and communication obstacles (i.e., if the direct line between two agents intersected one of these obstacles at any point in the agents' paths, line-of-sight communication was broken between these agents). We allowed the planner 10 seconds of planning time, and each individual tree was allowed up to 2 seconds. Results from this second set of experiments are shown in Figure 8.5(b); as with the first experiment, these results have been normalized.

For both these experiments, the Anytime RRT approach began by growing a standard RRT using the standard nearest neighbor and extension operators. Then, it slowly decreased d_b by $\delta_d = 0.1$ each iteration and increased c_b by $\delta_c = 0.1$. It switched its extension operator from the former approach mentioned in Section 8.2 to the latter approach after the third

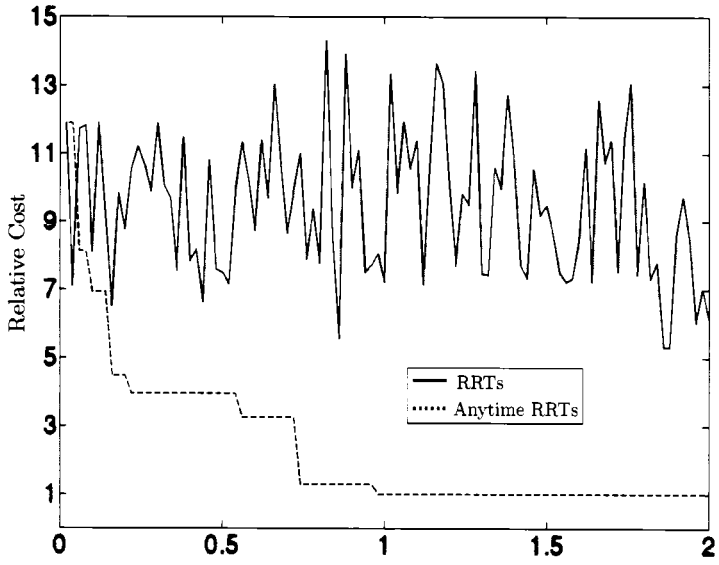


Figure 8.8: Example Anytime RRT results from a single run in one of the single robot environments. Shown are the RRT solution costs generated as time progresses. Note that the regular RRT approach does not use previous solution costs to influence the growth of future trees.

solution was found. ϵ_f was set to 0.1 so that each successive solution was guaranteed to be 10% less costly than the previous solution. For our heuristics we used Euclidean distance and our available extensions were straight-line segments for each agent.

There are a couple points worth noting from the results of these experiments. Firstly, both the regular RRT approach and the Anytime RRT approach generate the same initial solution (the graphs start from the same point in the upper left), so that both are able to provide an initial, valid solution in the minimum possible time. This is important for occasions where the available planning time may turn out to be very small and an agent has to act quickly.

Secondly, the Anytime RRT approach produces much better solutions, and is able to improve upon initial solutions much more quickly than the regular RRT approach. Overall, the best solutions generated by the regular RRT approach at the end of the maximum time allowed for planning were on average 3.6 and 2.8 times more expensive than the corresponding Anytime RRT solutions for the single agent and multi-agent cases, respectively. But the intermediate solutions produced by Anytime RRTs were also much better: Anytime RRTs were able to quickly and continually reduce the solution cost so that at any point in time they provided less costly solutions.

To provide a more detailed look at the behavior of each approach during a single run,

Figure 8.8 shows the results for a single environment in our single agent planning scenario. Here, we have plotted the most recent solution cost versus time over the course of planning. This graph shows the costs of the solutions generated by the Anytime RRT approach strictly decreasing over time, while the solutions produced by the regular RRT approach vary widely in cost and do not exhibit any general improvement. The environment from which these results came, along with some of the trees and solutions produced by the Anytime RRT approach, are illustrated in Figure 8.6.

A number of extensions to the Anytime RRT approach appear promising. Firstly, it may be possible to exploit even more information from previous solutions to aid in the generation of new ones, such as low cost branches of previous RRTs or extracted knowledge concerning the nature of the configuration space (e.g. as in [Burns and Brock, 2005] but with cost considerations). It is also worth further investigating how heuristics can be most effectively used to focus the growth of the trees. For example, over the course of our experiments we have found that when the heuristic is not very informed, inflating the heuristic values of nodes close to the root can prevent the RRTs from growing into 'dead ends', where no new nodes can be added because the early nodes were too expensive. Using information from previous searches and the current search to improve the heuristic estimates may be an even more effective approach.

8.3 Improving *and* Repairing RRTs

As shown in the previous section, the Anytime RRT algorithm drastically improves the quality of the solutions generated by RRTs and is well suited to planning with limited deliberation time in high-dimensional configuration spaces. However, this algorithm assumes we have perfect information regarding the environment. As discussed in Chapter 7, this is rarely the case in realistic applications. Instead, it is more likely that the initial information available will be imperfect and the agent or team of agents will need to update the solution as new information is received. In Section 7.2 we introduced the Dynamic RRT algorithm, which efficiently performs such updates when changes are made to the configuration space. However, this algorithm does not reason about path cost during either its initial search phase or its repair phase, so its solutions suffer the same limitations as those generated by the original RRT algorithm.

When deliberation time is limited, the configuration space is very high-dimensional and potentially non-uniform in cost, *and* the agent or team of agents is receiving updated environment information, it is important to be able to both generate low-cost solutions in a timely manner and repair these solutions when new information arrives that invalidates them.

In the following section, we present a sampling-based path planning algorithm that

combines the anytime behavior of Anytime RRTs with the replanning quality of Dynamic RRTs to provide exactly this capability. The resulting algorithm, *Anytime Dynamic RRTs*, interleaves planning, execution and observation exactly as in the discrete Anytime D* algorithm, and provides an approach tailored to very high-dimensional path planning problems in partially-known environments.

8.4 Anytime Dynamic Rapidly-exploring Random Trees

Since the Anytime RRT algorithm generates a series of low-cost RRTs and the Dynamic RRT algorithm efficiently repairs RRTs when changes are made to the configuration space, it seems natural to investigate whether the two algorithms can be combined into a single anytime replanning approach that is able to improve and repair its solution over time.

It is in fact quite straightforward to couple these approaches together. To begin with, the Anytime RRT algorithm is used to provide the best solution possible within the time available for planning. This solution is then executed by the team, and can be further improved upon by the Anytime RRT algorithm while the team is in motion. When changes are observed in the environment, these changes may invalidate the current solution path being executed by the team. However, by maintaining the search tree associated with this solution, the Dynamic RRT algorithm can be used to efficiently repair the tree and generate a new solution. Once this new solution has been produced, it can then be improved using the Anytime RRT algorithm, and the entire process can continue until the team reaches its goal.

This coupling has some significant advantages. First, we are able to take advantage of the low-cost solutions generated by the Anytime RRT algorithm, and can continually improve over time the quality of the solution executed by the team. Second, when changes are observed, rather than having to start the entire anytime process over from scratch, we are able to maintain the current low-cost solution and repair it. This results in a much better solution than we would otherwise have generated from scratch, and since we are only repairing the portions of the current solution that have been invalidated, this new solution is generated very quickly. Further, since we are continually improving the solution by generating new trees through the configuration space, the overall amount of trimming required when new information is received is minor. Thus, by combining both anytime and replanning capabilities, we are able to provide better solutions than either approach alone and we are able to provide these solutions with less computation.

We call the resulting approach *Anytime Dynamic Rapidly-exploring Random Trees* (Anytime Dynamic RRTs). An example application of the approach to a single agent planning problem is shown in Figure 8.9. In this example, the agent plans an initial path in an anytime fashion, then as it is about to traverse its least-costly path, it observes a new

obstacle that invalidates this path. It then updates its solution to take into account this obstacle. As with the Dynamic RRT algorithm, it can be beneficial for Anytime Dynamic RRTs to search backwards from the goal configuration so that the tree can be efficiently trimmed when new information is received in the vicinity of the agent or team of agents.

Pseudocode of the *Main* function of the Anytime Dynamic RRT algorithm is provided in Figure 8.10. In this pseudocode, two trees are used: the tree T is used to construct improved solutions over time and the tree R is used to store the best solution generated thus far. When changes to configuration space take place, the affected nodes in the tree R are trimmed and this tree is repaired to generate a new solution. The *GrowRRT* function used in the *RegrowRRT* function (line 21) and during the anytime planning phase (line 4) is the anytime function presented in Figure 8.3.

We have also included a backwards-searching version of the algorithm for interleaving planning, execution, and observation for a team of agents navigating through an environment. Pseudocode of this version is shown in Figure 8.11, with differences from the forwards, static version shown in red. The agents begin at configuration $q_{initial}$ and navigate to q_{goal}. The current solution path in the tree R is executed by the agents while the tree T is used to improve the quality of this solution. When changes are observed, the tree R is repaired (as above) to quickly generate a new solution for the team.

By varying the size of the execution steps taken by the team (line 2), it is possible to allow for more or less planning time for the team. For instance, if q_{goal} is updated to be a configuration several steps ahead of the current configuration in the path, rather than its direct parent, then more time is available for planning while the team transitions to q_{goal}. However, care must be taken to ensure that any obstacles that may reside along the segment of the path between the current configuration and q_{goal} will be observed by the agents before they start moving towards q_{goal}.

Anytime Dynamic RRT Results

We have applied the Anytime Dynamic RRT algorithm to the constrained exploration task in non-uniform cost, partially-known environments. To analyse its performance for this task, we planned traverses for a team of 3 agents navigating across a series of 100 random environments containing both communication obstacles and areas of non-uniform traversability cost. An example such environment is shown in Figure 8.12. An initial path was planned for the team through this environment, exactly as in the Anytime RRT experiments detailed earlier. However, as the agents began to execute this path, they observed navigation obstacles that were randomly placed in the environment. When these obstacles invalidated the current solution, the team was forced to generate a new solution.

We compared Anytime Dynamic RRTs to the original RRT algorithm, Dynamic RRTs,

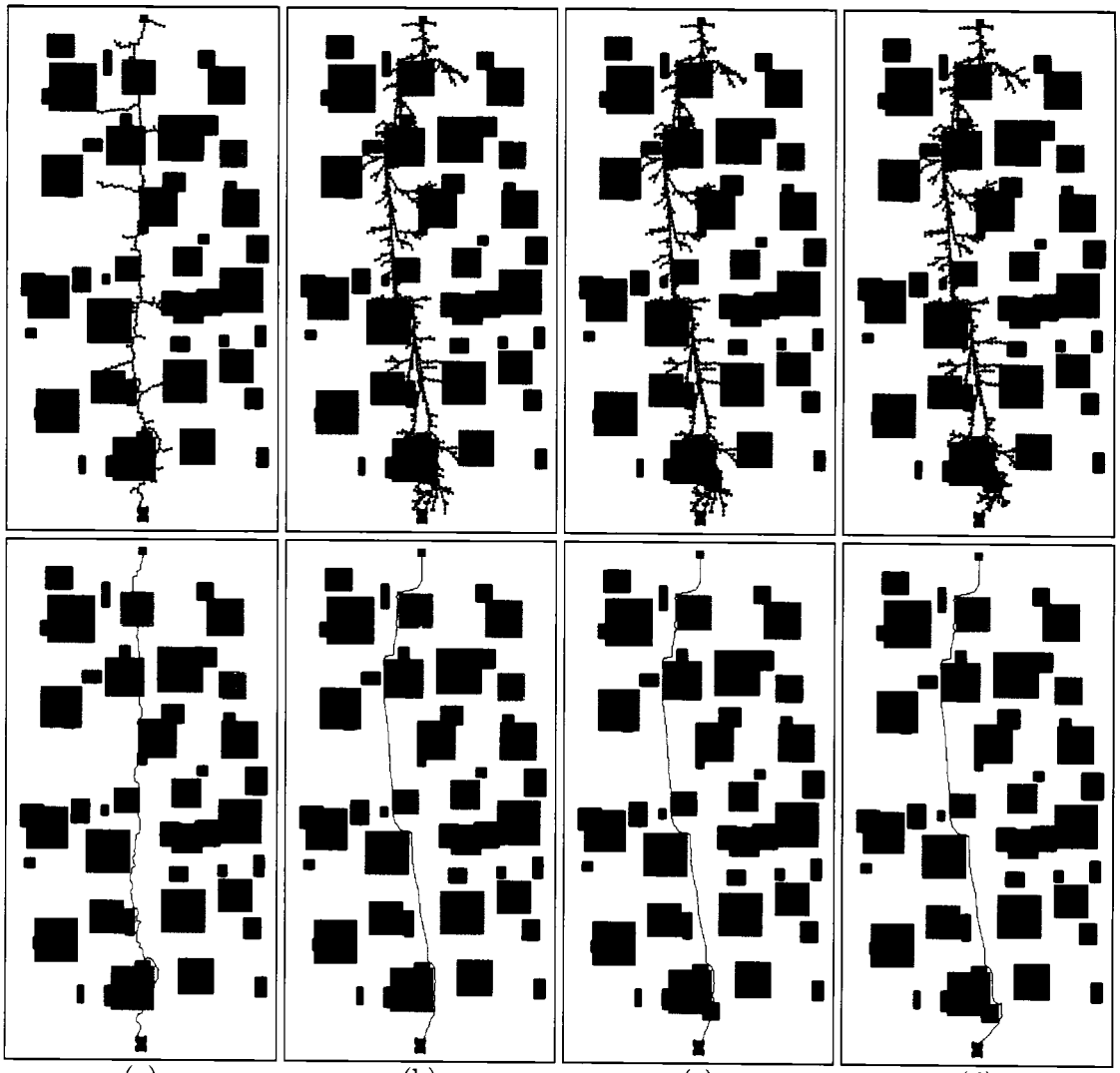


Figure 8.9: Anytime Dynamic RRTs used for single robot path planning. The initial configuration is shown by the vehicle at the bottom of the environment, the goal is the square at the top. Shaded regions represent higher-cost areas to traverse through; black regions represent obstacles. (a) Initial RRT generated without cost consideration. The top image shows the tree while the bottom image shows the solution path. (b) Final RRT generated in initial planning time, using costs of previous solutions and nodes of the current tree to bias the growth of the current tree. (c) A new obstacle is discovered in front of the agent, invalidating the final solution path and a portion of the search tree. (d) The tree is repaired to account for the new obstacle and a new solution is generated. The majority of the low-cost solution remains unaffected.

```
Main()
 1  T.d_b = 1; T.c_b = 0; T.C_s = ∞;
 2  forever
 3    ReinitializeRRT(T);
 4    T.C_n = GrowRRT(T);
 5    if (T.C_n ≠ null)
 6      R = T;
 7      PostCurrentSolution(R);
 8      R.C_s = T.C_s; R.d_b = T.d_b; R.c_b = T.c_b;
 9      T.C_s = (1 − ε_f) · T.C_n;
10      T.d_b = T.d_b − δ_d;
11      if (T.d_b < 0)
12        T.d_b = 0;
13      T.c_b = T.c_b + δ_c;
14      if (T.c_b > 1)
15        T.c_b = 1;
16    for each new obstacle o in configuration space
17      InvalidateNodes(R, o);
18    if significant obstacle changes were observed
19      increase R.C_s, R.d_b, and T.d_b; decrease R.c_b and T.c_b;
20    if solution path of R contains an invalid node
21      RegrowRRT(R);
22      PostCurrentSolution(R);
23      T.C_s = (1 − ε_f) · R.C_n;
```

Figure 8.10: The Anytime Dynamic RRT Algorithm: Main function

and Anytime RRTs. Each approach was allowed to run for a total initial planning time of 10 seconds (on a 1.5 GHz Powerbook G4), and each individual RRT was allowed to take up to 2 seconds. During execution, each approach planned from a configuration three steps ahead of it on the current path (i.e. q_{goal} was advanced three nodes in the path in Figure 8.11 line 2). The time taken to traverse each edge was set to 0.5 seconds, so that a total of 1.5 seconds was available for planning before the best solution was taken and the first three steps in this path were executed. Figure 8.13 plots the average cost of the team traverses following each of the four approaches. For these experiments, the Anytime Dynamic RRT and Anytime RRT approaches used an ϵ_f value of 0.1. When the solution was invalidated due to newly-observed obstacles, the Anytime RRT approach reset its C_s, d_b, and c_b values, while the Anytime Dynamic RRT approach increased these values by an amount that was dependent on how much of the search tree and solution was affected. If the newly-observed obstacles were minor, the Anytime Dynamic RRT approach only increased these values by a small amount, so that a low-cost repair could be made to the tree. If the newly-observed

```
MoveAgents()
 1 while (q_goal ≠ q_start)
 2   q_goal = R.parent(q_goal);
 3   while agents are not at q_goal;
 4     move agents towards q_goal;
 5     if any new obstacles are observed
 6       mark these obstacles as new;
Main()
 7 q_start = q_goal; q_goal = q_initial;
 8 fork(MoveAgents());
 9 while (q_goal ≠ q_start)
10   ReinitializeRRT(T);
11   T.C_n = GrowRRT(T);
12   if (T.C_n ≠ null)
13     R = T;
14     PostCurrentSolution(R);
15     R.C_s = T.C_s; R.d_b = T.d_b; R.c_b = T.c_b;
16     T.C_s = (1 − ε_f) · T.C_n;
17     T.d_b = T.d_b − δ_d;
18     if (T.d_b < 0)
19       T.d_b = 0;
20     T.c_b = T.c_b + δ_c;
21     if (T.c_b > 1)
22       T.c_b = 1;
23   for each new obstacle o in configuration space
24     InvalidateNodes(R, o);
25   if significant obstacle changes were observed
26     increase R.C_s, R.d_b, and T.d_b and decrease R.c_b and T.c_b;
27   if solution path of R contains an invalid node
28     RegrowRRT(R);
29     PostCurrentSolution(R);
30     T.C_s = (1 − ε_f) · R.C_n;
```

Figure 8.11: Using Anytime Dynamic RRTs to Interleave Planning and Execution. Differences between the forwards, static version of the *Main* function are shown highlighted in red.

obstacles affected a large portion of the tree, the Anytime Dynamic RRT approach increased these values a lot, possibly resetting them to their initial values. As with the experiments in known environments described in Section 8.2, for our heuristics we used Euclidean distance and our available extensions were straight-line segments for each agent.

By combining the anytime capability of Anytime RRTs and the replanning capability of Dynamic RRTs, Anytime Dynamic RRTs are able to provide better solutions for the

Figure 8.12: Sample initial map used for constrained exploration in partially-known environments. The initial positions of the agents are on the left, with the goals on the right.

team than either of these approaches, and much better solutions than the standard RRT algorithm. Both Anytime RRTs and Anytime Dynamic RRTs perform better than the non-anytime approaches because they are able to take the cost of the solution into account when performing their searches. Further, they use information from previous searches to help guide and restrict the next search towards cheaper solutions. Anytime Dynamic RRTs produce lower-cost traverses than Anytime RRTs because they are able to cope with new information much more intelligently. Rather than resetting the anytime parameters and generating a brand new solution from scratch, Anytime Dynamic RRTs are able to maintain the current, low-cost solution and repair just the portions of this solution that have been affected by the new information. This means both that Anytime Dynamic RRTs are able to produce a valid solution more efficiently than Anytime RRTs and that the solution produced is better.

We have also used the Anytime Dynamic RRT algorithm to plan for a team of 3 autonomous E-gator vehicles performing constrained exploration in a partially-known outdoor environment. As in our simulation experiments, the team began with a prior map of the environment that contained a collection of communication obstacles and areas of non-uniform traversal cost. The agents used this prior map and the Anytime Dynamic RRT algorithm to plan an initial path from their starting positions to their desired goal positions. As they traversed their paths, they observed navigation obstacles such as trash cans and bushes that existed in the environment but were not in their prior map. The E-gators used onboard laser range finders to detect these navigation obstacles. When these obstacles interfered with their current solution paths, these paths were repaired to account for the obstacles. Throughout the team's traverse, the solution path was continually being improved in an

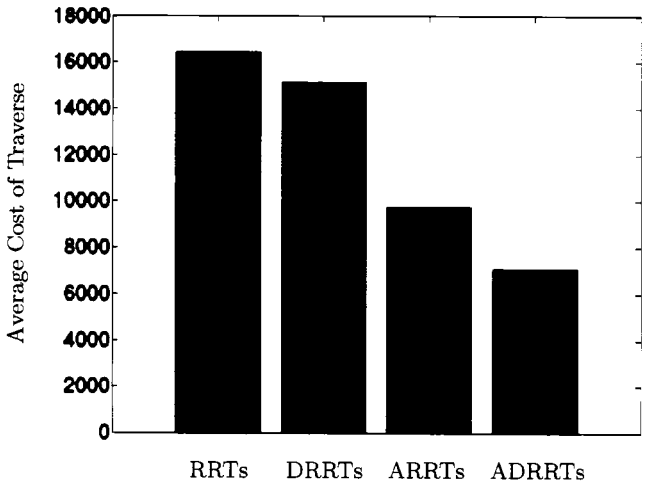


Figure 8.13: Anytime Dynamic RRT (ADRRT) results from our 3 agent constrained exploration experiments, along with results from regular RRTs, Dynamic RRTs (DRRTs), and Anytime RRTs (ARRTs).

anytime fashion.

Figure 8.14 shows a sequence of images from the team of E-gators navigating across an 80×100 meter environment. The top images show the initial positions of the gators and the prior map used by the team. The left image shows the center vehicle in the foreground and the right-most vehicle in the background. The right image shows the map, along with the agents (the red, green, and blue arrows) and their paths, the communication obstacles (in red), the high-cost regions (in dark grey), the goals for the agents (the red circles), and the line-of-sight links across the team (in yellow). The middle images show the team after the left-most vehicle has detected an obstacle in its path and the solution for the team has been repaired. The bottom images show the end of the run, with the right image of the pair showing the full paths traversed by the vehicles and the obstacles observed in the environment.

As we saw in Section 4.2 in regards to the Anytime D* algorithm, approaches that are able to improve and repair their solutions over time can be extremely useful for complex planning problems involving imperfect information. The results provided here show this holds true for the Anytime Dynamic RRT algorithm. By extending RRTs to exhibit both anytime and replanning behavior, Anytime Dynamic RRTs produce low-cost solutions to very high-dimensional planning problems in partially-known environments.

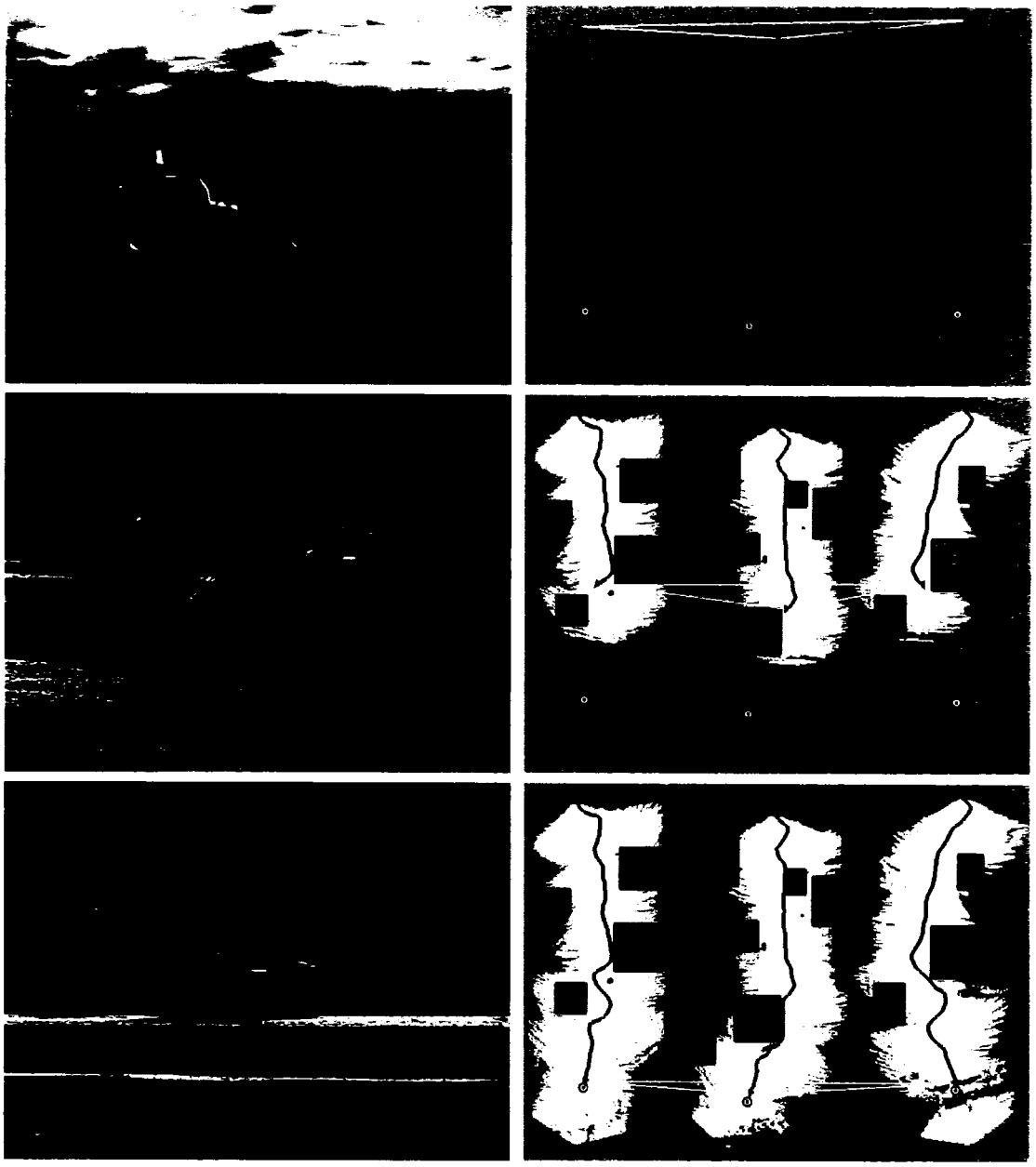


Figure 8.14: Anytime Dynamic RRTs used for constrained exploration by a team of 3 vehicles in an outdoor environment. On the left are images taken at various points during the traverse by the team. On the right are the corresponding positions of the vehicles (the color arrows), along with the paths executed thus far, the communication obstacles (in red) and observed navigation obstacles (in black), the high-cost regions (dark grey), and the areas observed to be obstacle-free by the onboard lasers (in white).

8.5 Discussion

Although single-shot sampling-based planning algorithms such as RRTs are extremely effective at generating feasible solutions, one of their most significant limitations is their inability to incorporate cost into their searches in order to produce high quality solutions. In this chapter, we have presented various techniques for biasing RRTs in favor of better solutions. These techniques are extremely useful for producing low cost solutions in non-uniform cost search spaces. Further, we have shown how these techniques can be used to create an anytime sampling-based planning algorithm that not only improves its solution over time, but can enforce bounds on the quality of this improvement.

Our anytime planner, known as the *Anytime RRT* algorithm, generates a series of RRTs, each producing a new solution that is guaranteed to be less expensive than the previous solution by a user-defined improvement factor ϵ_f. Thus, a valid solution is returned as quickly as by the standard RRT algorithm, but the quality of this solution is then improved while deliberation time allows. The resulting algorithm provides similar benefits as discrete anytime algorithms, such as ARA* [Likhachev et al., 2003], but is able to plan over much larger, higher-dimensional search spaces. We have provided key properties of the algorithm including relative bounds on the quality of the solutions generated, and have demonstrated its effectiveness for both single agent navigation and multi-agent constrained exploration in known environments.

In order to provide high quality solutions when deliberation time is limited and the environment is imperfectly-known, we have combined the Anytime RRT algorithm with the Dynamic RRT algorithm presented in the previous chapter to create a sampling-based anytime, replanning algorithm which we call *Anytime Dynamic RRTs*. The Anytime Dynamic RRT algorithm is able to both improve and repair the solution over time, and is particularly well-suited to path planning in high-dimensional search spaces for teams of agents navigating partially-known environments. We have presented results highlighting its benefits in both simulation and for a team of 3 robotic vehicles performing constrained exploration in partially-known outdoor environments.

Chapter 9

Multi-agent Planning in Dynamic Environments

When a team of agents is navigating through an environment containing other moving agents or objects, it is important for the team to take into account these other dynamic elements in order to produce more robust, less-costly solutions.

As mentioned in Chapter 5, incorporating dynamic elements into the path planning problem involves reasoning about time. We have already discussed some common approaches for doing this, such as planning in full state-time space or using a simplified representation of either our state-space or time. For the multi-agent planning problem, since we are already dealing with such a high-dimensional search space to begin with, sampling-based approaches can be much more appropriate than discrete approaches for additionally coping with time.

In this chapter, we describe how the sampling-based algorithms developed in the second part of this thesis can be effectively applied to planning in dynamic environments. We begin by discussing how many of the multi-agent planning approaches described in Chapter 6 can be used for planning in dynamic environments, along with some of the limitations of these approaches. Next, we show how we can represent the constrained exploration problem in dynamic environments as a search through state-time space. Because information regarding the dynamic elements is rarely perfect, we consider two different scenarios involving imperfect initial information. First, we consider the case where the team initially has reasonably accurate information about the entire trajectory of these elements, and occasionally updates these trajectories based on new information during the team's traverse. This represents a situation where the dynamic elements may be under our control or are providing us with their intended movement. We also consider the case where the team initially has very inaccurate information concerning these dynamic elements, and is only provided their positions, headings, and velocities. This is a similar situation as the one addressed in Chapter 5 and requires the team to estimate the trajectories of the dynamic elements, then update these

estimates regularly as new information is received. We apply our RRT-based algorithms to different instances of these scenarios and present comparative results.

9.1 Incorporating Dynamic Elements in Multi-agent Planning

As described in detail in Chapter 5, incorporating dynamic obstacles into planning requires reasoning about time. A robust way to do this is to extend the search space to include a time dimension, so that every state in our extended search space consists of a configuration from our configuration space and a point in time. When dealing with single agent planning problems, adding this extra dimension to our search space often presents a significant relative increase in the complexity of the planning problem, and so various approaches have been developed that attempt to extend the efficient single agent planning algorithms to cope with state-time space (several such approaches were described in Chapter 5). However, for multi-agent planning problems we are already dealing with high-dimensional configuration spaces, so the relative increase in complexity resulting from adding an additional dimension for time is not nearly as significant.

As a result, several of the multi-agent planning algorithms from Chapter 6 can be used directly to plan over configuration-time space. In particular, sampling-based algorithms such as RRTs and PRMs can search through this extended search space quite efficiently. In the following section we describe how the multi-agent constrained exploration problem in dynamic environments can be represented as a search in configuration-time space, and we use the RRT algorithm to plan paths through this space.

In Chapters 7 and 8 we discussed in detail several limitations of the basic RRT algorithm, in particular its inability to cope with imperfect information regarding the configuration space and the poor quality of its solutions. These limitations motivated our development of a series of extensions to the RRT approach. In order to generate better solutions and efficiently repair the current solution when new information is received, we also apply these algorithms to our constrained exploration problem in dynamic environments.

Performing Constrained Exploration in Dynamic Environments

Incorporating dynamic elements into the constrained exploration problem can be done by adding the time dimension to our search space, as mentioned above. In the resulting configuration-time search space the complete trajectories of dynamic obstacles can be represented, and these obstacle trajectories can then be avoided during planning.

Figure 9.1 shows an example of the constrained exploration problem in a dynamic environment. In this image, we have three robots performing constrained exploration in an environment containing a number of dynamic obstacles, such as other agents (shown in red).

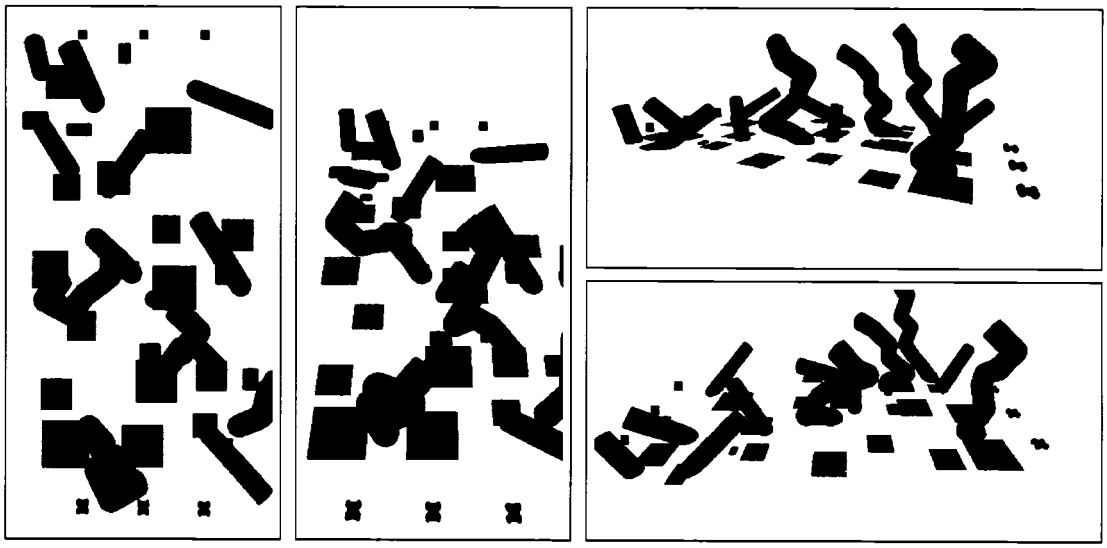


Figure 9.1: The Constrained Exploration Problem in Dynamic Environments. These images show a collection of different views of an environment containing a series of dynamic obstacles, communication obstacles, and high-cost regions. The trajectories of the dynamic obstacles are shown in red, with the time axis extending upwards. The ground plane has been shaded light gray in order to distinguish it from the background in the various views.

We have represented time by the vertical axis (as in Figure 5.8), so that the trajectories of these dynamic obstacles extend upwards. In order to provide a feasible solution to this task, a joint path needs to be planned for the team through configuration-time space that avoids both the static obstacles in the environment and the dynamic obstacles.

RRTs can be used to plan such paths. To do this, the same approach used for the constrained exploration task in static environments can be used with a couple minor modifications. First, some care must be taken when sampling the search space and selecting a node from the tree for extension. One method for doing this is to sample from the full configuration-time space, then select from the tree the node closest to this sample. However, if the sample point has a time value that is less than its closest node in the tree, then it is impossible for the tree to extend from this node to the sample point. Another approach, which we use in our implementation, is to generate samples from the configuration space and find the nodes in the tree that are closest to these samples in the configuration space. This method ignores the time dimension when sampling and selecting nodes for extension, but incorporates time when performing the extension operation. This has the effect of biasing the tree to explore the configuration space as much as possible.

The second minor modification required to the RRT approach used for static environments regards the extension operation. When dealing with dynamic obstacles, the extension operation needs to increment the time component of the node being extended and collision-

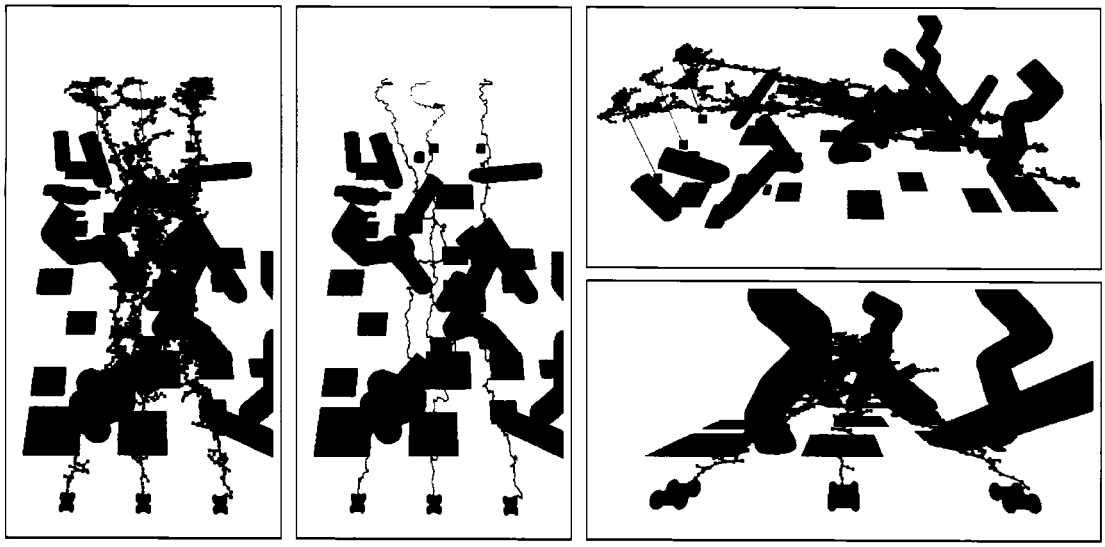


Figure 9.2: Using RRTs for performing constrained exploration in dynamic environments. These images show a collection of different views of an RRT grown through a configuration space containing a series of dynamic obstacles. As with earlier RRT images, the high-dimensional tree has been broken into 3 trees, one for each agent, for visualization. The trajectories of the dynamic obstacles are shown in red, with the time axis extending upwards. The second image from the left shows the solution path generated by the RRT.

check the extension against both the static obstacles and the dynamic obstacles. This collision-checking is not fundamentally different from the static case, but it does require the extension operation to be parameterized by time, whereas in the static case only the configurations along this extension needed to be collision-checked.

Given these modifications, RRTs can be successfully applied to this task. Figure 9.2 shows an RRT and corresponding solution path for the team of agents performing constrained exploration in the environment shown in Figure 9.1. Notice that the RRT and path extend upwards through time. In these images, we have also added vertical lines extending from the goals of the agents upwards through time to show that the goal of the search is a configuration and not dependent on time.

9.2 Using Anytime RRTs and Anytime Dynamic RRTs for Constrained Exploration in Dynamic Environments

Although RRTs can be used to solve the constrained exploration task in dynamic environments, we have already seen that much better solutions can be provided in static environments by using one of the extensions to the RRT algorithm introduced in Chapters 7 and 8. It thus makes sense to investigate whether these approaches also provide improved

performance in the dynamic setting. Further, if the initial information concerning either the static or dynamic elements of the environment is imperfect, the replanning ability of these algorithms may be particularly useful for efficiently repairing the solution. Since the dynamic elements vary their positions with time, the speed with which solutions are generated or repaired is even more critical in this domain, to ensure that solutions are not out of date when they are produced.

To this end, we applied the Anytime RRT and Anytime Dynamic RRT algorithms to this task and compared their performance to the standard RRT algorithm. To analyse the ability of each of the three approaches to cope with imperfect information, we had each approach generate an initial trajectory for the team, then as this trajectory was being executed, new information was received concerning the dynamic elements in the environment. We simulated two common types of dynamic elements and included both in the environment. The first were dynamic elements whose full trajectories were provided to the team initially and then these entire trajectories were updated as new information was received. These dynamic elements were intended to represent other agents or objects that we had fairly accurate information about, and which provided us with updates whenever their intended trajectories changed. The second type of dynamic elements we included in the environment had unknown trajectories. These elements were intended to represent other agents or objects that we had very little information about, such as their current positions and velocities. For these elements, we estimated their trajectories by extrapolating their current velocities, as in our single agent planning scenario described in Section 5.2, and then updated these trajectories as new observations of their positions and velocities were made.

We had both the Anytime RRT and Anytime Dynamic RRT algorithms search in a forwards direction, from the current configuration (and time) of the team to the goal configuration. This was done for two reasons. First, when searching through configuration-time space, it can be computationally expensive to search backwards from the goal because there is no specific destination configuration-time node to search from. Instead, the search needs to be initialized with a set of different configuration-time nodes, where the configuration value of each of these nodes is the goal configuration and the time value of each node varies over some estimated range of times. Searching backwards from this set can be cumbersome and connecting this search tree to the current configuration and time of the team can be difficult, since the time value must match. It is possible to perform such a search, and we saw in Chapter 5 that this worked well for single agent planning in dynamic environments, however it is usually more efficient to search in a forwards direction when possible. Typically, backwards searching is performed in order to facilitate efficient repair of solutions when new information is received and to allow for the situation where the agent or team of agents is moving. With the Anytime RRT algorithm, no repair is performed, so this

consideration was irrelevant. With the Anytime Dynamic RRT algorithm, it is certainly most efficient to repair solutions when new information is being received close to the leaves of the tree rather than the root. However, as we will discuss next, in the current scenario this is not as relevant as it was in our static planning domains. Further, the Anytime Dynamic RRT approach can still use and repair previous forwards-searching solutions when the agent or team of agents is moving. To do this, it simply trims all areas of the tree that are not descendants of the current node of the agent or team, as well as any portions of the tree affected by any newly-observed obstacles. The current node then becomes the root of the tree and the tree can continue to be grown. This is a useful advantage the Anytime Dynamic RRT algorithm has over discrete replanning algorithms, such as Anytime D*.

The second reason we perform the search in a forwards direction is due to the nature of the new information being received by the team during its traverse. In the static constrained exploration scenario, new information was being received in the vicinity of the team, and thus affecting portions of the current search tree that were near the current configuration of the team. In our dynamic scenario, however, this is no longer necessarily the case. Because the team is receiving updated information in the form of entire trajectories of dynamic elements, this information can affect areas of the configuration-time search space that are very distant from the current configuration-time state of the team. This means that repairs will not be localized to areas of the tree near the current node of the team, and so performing these repairs for a backwards-searching tree will not necessarily be more efficient than for a forwards-searching tree. As a result, the backwards-searching approach holds no real benefit in efficiency in regards to replanning to counter the greater efficiency of the forwards-searching approach in regards to generating initial solutions.

Figures 9.3 and 9.4 present pseudocode of the Anytime Dynamic RRT algorithm being used to interleave planning, execution, and observation in environments containing imperfectly-known dynamic obstacles. Differences between the backwards-searching version of the algorithm used for searching through static configuration spaces are highlighted in red. We have also added a new function *UpdateRoot* that updates the tree as the agent or team moves along the solution path. This function is used in conjunction with a modified version of the *TrimRRT* function to trim and repair the tree when new information is received. Apart from this new function, there are also a few new minor constructs: T.nodeonpath(i) returns the node at index i in the solution path of tree T and T.childonpath(q) returns the node after q in the solution path of tree T. Because this pseudocode is tailored to searching through configuration-time space, the identifiers q, q_{start}, q_{goal}, etc, refer to configuration-time states. The function *Configuration*(q) returns the configuration element of the configuration-time state q. The goal state q_{goal} may have an unspecified time value if we are not concerned with the time taken to reach the goal configuration.

```
UpdateRoot(rrt T)
 1  q = T.nodeonpath(1);
 2  while (q ≠ q_start)
 3    q.flag = INVALID;
 4    q = T.childonpath(q);

TrimRRT(rrt T)
 5  Q = ∅; i = 1;
 6  while (i < T.size())
 7    q_i = T.node(i); q_p = T.parent(q_i);
 8    if (q_p.flag = INVALID) AND (q_i ≠ q_start)
 9      q_i.flag = INVALID;
10    if (q_i.flag ≠ INVALID)
11      Q = Q ∪ {q_i};
12    i = i + 1;
13  T = CreateTreeFromNodes(Q);
```

Figure 9.3: Using Anytime Dynamic RRTs to Interleave Planning and Execution in Configuration-Time Space (forwards-searching version): UpdateRoot and (Modified) Trim-RRT functions

Constrained Exploration in Dynamic Environments Results

To compare the relative performance of Anytime RRTs and Anytime Dynamic RRTs against standard RRTs for the constrained exploration task in dynamic environments, we had each approach plan traverses for a team of 3 agents navigating across a series of random environments containing communication obstacles, areas of non-uniform traversability cost, and dynamic obstacles. These traverses were similar to those performed in Section 8.4 except that the environment now contained dynamic obstacles and new information was received concerning these dynamic obstacles as the team executed its traverse. We left Dynamic RRTs out of our comparison because they were shown to be less effective than Anytime Dynamic RRTs in non-uniform cost search spaces in the previous chapter. As discussed above, two types of dynamic obstacles were considered. In our experiments, we created 20 different environments containing randomly generated communication obstacles and high-cost areas, and for each of these environments we used 10 different collections of dynamic obstacles, each having a randomly-generated trajectory. Each of these collections contained 8 dynamic obstacles whose trajectories were unknown and needed to be estimated by the team, and 4 dynamic obstacles for which fairly accurate trajectories were initially provided to the team. An example environment used in our experiments is shown in Figure 9.1.

As the team traversed through the environment, updated information on the actual trajectories of both types of dynamic obstacles was received. Each of the dynamic obstacles

```
MoveAgents()
 1 while (Configuration(q_start) ≠ Configuration(q_goal))
 2   q_start = R.childonpath(q_start);
 3   while agents are not at q_start;
 4     move agents towards q_start;
 5     if any new obstacles or changes to dynamic obstacle trajectories are observed
 6       mark these obstacles or obstacle trajectories as new;
Main()
 7 q_start = q_initial;
 8 fork(MoveAgents());
 9 while (Configuration(q_start) ≠ Configuration(q_goal))
10   ReinitializeRRT(T);
11   T.C_n = GrowRRT(T);
12   if (T.C_n ≠ null)
13     R = T;
14     PostCurrentSolution(R);
15     R.C_s = T.C_s; R.d_b = T.d_b; R.c_b = T.c_b;
16     T.C_s = (1 − ε_f) · T.C_n;
17     T.d_b = T.d_b − δ_d;
18     if (T.d_b < 0)
19       T.d_b = 0;
20     T.c_b = T.c_b + δ_c;
21     if (T.c_b > 1)
22       T.c_b = 1;
23   for each new obstacle or dynamic obstacle trajectory o in configuration-time space
24     InvalidateNodes(R, o);
25   if significant obstacle changes were observed
26     increase R.C_s, R.d_b, and T.d_b and decrease R.c_b and T.c_b;
27   if solution path of R contains an invalid node
28     UpdateRoot(R);
29     RegrowRRT(R);
30     T.C_s = (1 − ε_f) · R.C_n;
```

Figure 9.4: Using Anytime Dynamic RRTs to Interleave Planning and Execution in Configuration-Time Space (forwards-searching version): (Modified) MoveAgents and Main functions.

with unknown trajectories changed their course 20 times, and each of the other dynamic obstacles updated their full trajectories 4 times. Whenever new information was received concerning any of the dynamic obstacles, each planning approach checked whether this information invalidated its current solution. If so, a new solution was generated, with each approach using its associated method for producing this solution.

Each approach was allowed to run for a total initial planning time of 20 seconds (on a

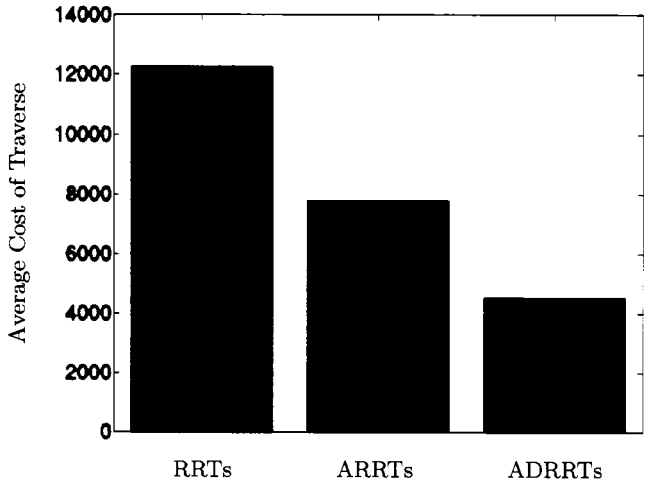


Figure 9.5: Comparative results from our 3 agent constrained exploration experiments in dynamic environments.

1.5 GHz Powerbook G4), and each individual RRT was allowed to take up to 10 seconds. During execution, each approach planned from a configuration three steps ahead of it on the current path (i.e. q_{start} was advanced three nodes in the path in Figure 9.4 line 2). The time taken to traverse each edge was set to 2 seconds, so that a total of 6 seconds was available for planning before the best solution was taken and the first three steps in this path were executed. Figure 9.5 plots the average cost of the team traverses following each of the three approaches.

As with our earlier constrained exploration experiments, the Anytime Dynamic RRT and Anytime RRT approaches used an ϵ_f value of 0.1. Again, when the solution was invalidated due to newly-observed trajectories of the dynamic obstacles, the Anytime RRT approach reset its C_s, d_b, and c_b values, while the Anytime Dynamic RRT approach increased these values by an amount that was dependent on how much of the search tree and solution was affected. For these experiments we tried out a more informed heuristic function than Euclidean distance: for each agent we initially pre-computed a simple 2D navigation function to its goal location then stored these values for each position in our discretized 2D map. The heuristic cost between two configurations was computed as the sum of the individual navigation function estimates for each agent.

As can be seen from the results of these experiments, Anytime Dynamic RRTs maintain their performance edge over the other approaches when dynamic obstacles are added to the environment. Although new information regarding the trajectories of these obstacles can affect states in the configuration-time space that are widely-separated in terms of distance, Anytime Dynamic RRTs are still able to repair the solution when this new information is

received more efficiently than regenerating a series of new solutions from scratch.

9.3 Discussion

In this chapter, we have described the general problem of multi-agent path planning in dynamic environments. We have discussed how existing multi-agent planning approaches can be applied to this problem and provided an example illustration of the RRT algorithm being used to solve the constrained exploration problem in environments containing dynamic obstacles.

To improve upon the results provided by the RRT algorithm, particularly in regards to solution quality, we have applied the Anytime RRT and Anytime Dynamic RRT algorithms introduced in Chapter 8 to this problem. Each of these algorithms, along with RRTs, can be made to search for solutions through configuration-time space and they are thus all able to incorporate the trajectories of dynamic obstacles. We have also provided a version of the Anytime Dynamic RRT algorithm that interleaves planning, execution, and observation while searching in a *forwards* direction through the search space. This version of the algorithm is particularly well-suited to time-varying problems as, in contrast to backwards searching approaches, it does not need to construct an initial set of configuration-time goal nodes and search backwards from all of these nodes.

We have compared the performance of the standard RRT approach, Anytime RRTs, and Anytime Dynamic RRTs on the problem of constrained exploration in dynamic environments, and found that Anytime Dynamic RRTs are able to produce much less expensive solutions than the other two approaches. In dynamic environments, as in static environments, the ability of Anytime Dynamic RRTs to efficiently repair the current low-cost solution as well as improve this solution over time provides it with a significant advantage over algorithms that are only endowed with one, or neither, of these capabilities.

Chapter 10

Conclusion

Path planning for single agents and multi-agent teams navigating through the real world involves overcoming a number of challenges not present in simpler domains. In this thesis, we have concentrated on three of these challenges: *imperfect information*, *limited deliberation time*, and *dynamic environments*. We have looked at how these challenges impact two motivating planning problems: outdoor mobile robot navigation and multirobot constrained exploration.

We began by describing the single agent path planning problem and presented a number of approaches that are currently applied to this task. In particular, we looked at two general classes of path planning algorithms: discrete and sampling-based. We stated that discrete algorithms are typically favored when dealing with low-dimensional problems such as outdoor mobile robot navigation, and sampling-based algorithms are favored when dealing with more complex, higher-dimensional problems. We then described some very common discrete search algorithms used for path planning in mobile robot navigation.

However, in real world scenarios the initial information available concerning the environment is imperfect, and so any plans generated based on this initial information may become invalid or suboptimal as new information is received. In Chapter 3 we described two different approaches for coping with imperfect information: assumptive planning and planning with uncertainty.

Assumptive planners use the initial information available to come up with an estimate of the environment, then plan using this estimate. When new information is later received, the estimate is updated along with the current plan. Generating brand new plans in such situations can be extremely expensive in terms of computation. As a result, discrete replanning algorithms have been developed that repair the current plan when new information is received. These algorithms are much more efficient than planning over from scratch, yet even these algorithms can be significantly improved upon. In Sections 3.2 and 3.3 we introduced the *Delayed D** and *Field D** algorithms, two replanning approaches that provide improved efficiency and solution quality, respectively, than current replanning algorithms

applied to mobile robot navigation. Delayed D* delays processing new negative information (such as newly-discovered obstacles) as long as possible, and is able to provide optimal solutions in half the time of current approaches in common mobile robot scenarios. Field D* uses interpolation to allow for more flexibility in planning paths through commonly-used grid representations of the environment and produces more direct, less-costly paths for our robots to execute.

The assumptive planning approach works well when errors in the initial estimate of the environment are not crucial, for instance when the environment is relatively empty. However, there are certain situations where this approach will produce grossly suboptimal paths. For instance, imagine there exists a narrow valley in the environment and there is some uncertainty as to whether the valley can be navigated. In such situations it is important to reason about the uncertainty associated with the initial information and plan paths that take into account this uncertainty. Unfortunately, this transforms the path planning problem into a POMDP and makes generating solutions very difficult in general and intractable for our mobile robot navigation scenario, where the size of the information space is monumental. To reduce the size of this space, we introduced an approach in Section 3.6 that determined the most important areas of uncertainty in the environment and then incorporated just these areas of uncertainty into the path planning task. We then introduced an efficient heuristic search algorithm called *Propagating AO** (PAO*), that is able to produce optimal paths given these areas of uncertainty, for an order of magnitude less computation time than is required by current approaches.

As well as having to deal with imperfect initial information, robots acting in the real world often have very limited time available for planning. In such cases, solutions must be generated quickly to ensure there is a feasible action for the robot to execute. In Chapter 4 we described a class of algorithms known as anytime approaches, which are able to rapidly produce an initial, suboptimal solution, and then improve the quality of this solution as deliberation time allows. These algorithms are very useful for planning under time constraints. However, they typically require perfect initial information as they rely on the results of previous processing to efficiently improve their solutions. They are thus unable to cope with both limited planning time *and* imperfect information. To remedy this, in Section 4.2 we introduced a discrete, anytime replanning algorithm called *Anytime Dynamic A** that is able to both improve its solution over time and repair the solution when new information is received concerning the environment. Anytime Dynamic A* is able to provide high quality solutions in high-dimensional search spaces, even when confronted with both limited deliberation time and imperfect information.

We then went on to show how Anytime Dynamic A* can be used to solve problems involving all three of our challenges, that is, where time for planning is limited, the agent has imperfect initial information, and the environment is dynamic. To do this, we combined

the discrete Anytime Dynamic A* algorithm with an efficient sampling-based representation of the configuration space. The resulting approach is able to provide low-cost solutions to very general path planning problems.

In the second part of the thesis we concentrated on multi-agent path planning. To begin with, we discussed how the sampling-based approaches described in Chapter 2 were well-suited to this task due to their ability to cope with very high-dimensional search spaces. In particular, we described how the Rapidly-exploring Random Tree (RRT) algorithm can be used to solve our motivating constrained exploration problem.

In Chapter 7 we looked at multi-agent planning with imperfect initial information, and discussed the limitations of current RRT-based approaches to this problem. Even the most efficient of these approaches generates a complete new search tree when new information is received that invalidates the previous solution. We then introduced a new sampling-based replanning algorithm called *Dynamic RRTs* that repairs the search tree and solution when new information is received. The Dynamic RRT algorithm trims the portions of the existing search tree that have been affected by new obstacle information, and re-grows the remaining tree to quickly produce a new feasible solution. We also showed how this algorithm can be used as a sampling-based analog of the widely-used D* algorithm.

Another significant limitation of RRTs is their inability to incorporate cost into their searches in order to produce high quality solutions. In Chapter 8 we introduced an anytime variant of RRTs called *Anytime RRTs* that is able to produce low cost solutions in both uniform and non-uniform cost search spaces and with limited deliberation time. This algorithm generates a series of RRTs, each producing a new solution that is guaranteed to be less expensive than the previous solution by a user-defined improvement factor. This approach provides similar benefits as discrete anytime algorithms but is able to plan over much larger, higher-dimensional search spaces.

To provide high quality solutions given limited deliberation time *and* imperfect initial information, we combined the Dynamic RRT and Anytime RRT algorithms to create a sampling-based anytime replanning algorithm called *Anytime Dynamic RRTs*. This algorithm can both improve the quality of its solution over time *and* repair the solution when new information is received. We showed how this approach can be used to interleave planning, execution, and observation for a team of agents navigating through an environment, as a sampling-based analog of our discrete Anytime Dynamic A* algorithm.

Finally, we applied the Anytime Dynamic RRT algorithm to a multi-agent path planning problem involving all three of our challenges: the constrained exploration problem in imperfectly-known, dynamic environments. We described how this algorithm can be used to search through configuration-time space and showed that its relative advantage over competing approaches extended to this domain also.

10.1 Usage of Algorithms

The algorithms presented in this thesis are applicable to a wide range of path planning problems involving single agents and multi-agent teams. However, in order to select an appropriate algorithm for a specific problem instance, it is useful to have some intuition regarding their characteristics.

The discrete replanning algorithms introduced here are well-suited to problems involving relatively low-dimensional search spaces in which changes to the search space are taking place. However, because these algorithms work by repairing their solutions when changes to the search space occur, if the search space is changing a huge amount between each planning episode it can be more efficient to simply discard the existing solution and generate a new one from scratch. This is true of all replanning algorithms, so it is important to think carefully about just how dynamic or imperfectly-known the search space is in a particular problem before applying a replanning algorithm. Typically, in mobile robot navigation the changes to the search space are localized around the robot and thus this problem can be solved effectively with replanning algorithms.

The anytime replanning algorithm we presented combines the benefits of both anytime and replanning algorithms in terms of its ability to both improve its solution over time and repair its solution when changes to the search space occur. However, it also suffers some of the drawbacks of both anytime and replanning algorithms. In particular, as just discussed in regards to replanning algorithms, if the search space is undergoing huge changes then it can be more efficient to ignore the existing solution and generate a brand new one. This is particularly true for our anytime replanning algorithm, as there is some extra overhead in combining the anytime and replanning capabilities. Thus, it is important when applying this algorithm that attention is paid to how much has changed in the search space between planning episodes; if the changes are large then a new search should be performed[1]. However, the anytime property of this algorithm can be very beneficial in such scenarios, as it can be used to very efficiently generate a suboptimal new solution and then improve this solution while deliberation time allows.

The sampling-based algorithms introduced in this thesis are suited to very high-dimensional search spaces, for which discrete algorithms would be unable to generate solutions. As with their discrete counterparts, the applicability of the sampling-based replanning algorithms to particular problems depends on the nature of changes taking place in the search space. However, since the process of removing newly-invalid portions of the search tree when changes occur is not as expensive in the sampling-based algorithms (we simply throw out the entire sub-tree when a branch is invalidated), even when huge changes are taking place

[1] In Chapter 4 we discuss just what constitutes a 'large' change to the search space, and how this quality can be determined for a particular application.

replanning with these algorithms is never too much more computationally expensive than planning from scratch. One additional important consideration in replanning in very high dimensional search spaces, though, is how changes to the search space are detected. If some environmental change is observed, it may not be trivial to determine which areas of the search space are affected by this change. In certain problem domains, this determination may be extremely computationally expensive and/or the size of the affected area of the search space may be very large, and so even if the environmental changes are small the consequences of these changes in terms of planning time may be very large.

Finally, the anytime sampling-based algorithms we presented are well-suited to high-dimensional search problems for which several solutions exist of varying quality. Because these algorithms use heuristic information in their generation of solutions, the quality of their results can be improved significantly by having accurate heuristics for a particular planning problem. However, since they also exploit information from previous solutions to aid in their generation of new, improved solutions, this heuristic information is not absolutely necessary; it will certainly help but its absence does not render these algorithms ineffective. Further, even very simple, easily-generated heuristics can be used by these algorithms to generate high quality solutions.

As a general note, one key concept that has been taken advantage of throughout this thesis is the reuse of information. Exploiting previous paths and path costs has been a central idea in our replanning algorithms. Both our discrete and sampling-based replanning approaches use this information to efficiently generate new solutions. Previously computed path cost information is also reused in our PAO* algorithm to update the path costs of several other states during our search, significantly improving the overall efficiency of planning. Our anytime algorithms exploit not just the path costs of states but the overall quality of the previous solution to ensure that the next solution generated is better. The major assumption of all these approaches, particularly the replanning algorithms, is that there is enough useful information still present in these previous solutions or path costs that it is worth extracting this information and repairing these solutions rather than generating new ones. Of course, this is not always true, and in the worst-case our replanning approaches can be less efficient than replanning from scratch. However, in general this does hold true, and in certain scenarios, such as mobile robot navigation, these approaches are extremely effective. By exploiting previous information and processing, we are able to efficiently improve or repair our current solution and provide better overall task performance.

10.2 Contributions

In conclusion, the contributions of this work are a series of planning algorithms that are effective for both single agent and multi-agent path planning problems involving imperfect

information, limited deliberation time, and dynamic environments. In particular, we have contributed the following algorithms:

- A graph-based replanning algorithm, *Delayed D**, able to repair solutions more efficiently than existing replanning algorithms
- An interpolation-based replanning algorithm, *Field D**, able to efficiently provide straighter, less-costly paths through both uniform and non-uniform resolution grids than existing grid-based planning algorithms
- An uncertainty-based path planning algorithm, *Propagating AO**, able to reason about the uncertainty in the information held by an agent to produce informed paths for an order of magnitude less computation than required by competing approaches
- An anytime replanning algorithm, *Anytime Dynamic A**, able to both improve its solution over time and repair the solution when new information is received; to our knowledge, Anytime Dynamic A* is the first such algorithm developed
- A sampling-based replanning algorithm, *Dynamic RRTs*, able to plan and replan in very high-dimensional search spaces far more efficiently than existing RRT-based approaches
- A sampling-based anytime algorithm, *Anytime RRTs*, able to produce much better solutions in high-dimensional, non-uniform cost search spaces than existing RRT-based approaches, and to produce these solutions in an anytime fashion and with enforceable relative solution bounds
- A sampling-based anytime, replanning algorithm, *Anytime Dynamic RRTs*, able to plan in high-dimensional search spaces and to both improve its solution over time and repair the solution when new information is received; this algorithm allows for true interleaving of planning, execution, and observation in high-dimensional search spaces

As a result of these algorithms, our agents are better equipped to deal with the complexities associated with planning and acting in the real world.

10.3 Future Extensions

There are a number of promising directions for building upon and improving the approaches presented in this thesis.

Firstly, planning with uncertainty remains a challenging area. In this thesis, we have provided an approach for extracting key areas of uncertainty and then incorporating these

areas into the planning process. However, in general this remains a very difficult problem and our approach is limited to fairly small sets of uncertain areas. Recently, Likhachev and Stentz have developed an approximation algorithm that is able to cope with many more areas of uncertainty [Likhachev and Stentz, 2006], and this work is currently being extended to more complex instances of the navigation problem involving teams of coordinating agents and potential adversaries. This is currently an active area and much more remains to be done. A second source of uncertainty we did not discuss in this thesis is *positional* uncertainty, where our agent does not have perfect information concerning its position in the environment. Dealing with this type of uncertainty becomes particularly important when planning for agents navigating in areas without global position information. In such scenarios, taking into account the positional uncertainty is very important for generating paths that can actually be executed by the agent. Roy and Thrun [Roy and Thrun, 1999] and more recently Gonzalez and Stentz [Gonzalez and Stentz, 2005] have developed path planners that perform this reasoning and plan paths that account for positional uncertainty and exploit environmental cues to improve the agent's position estimate. Again, there are several promising avenues for further research in this area, including incorporating an information-based uncertainty planner (such as the one we presented in Chapter 3) with a position-based uncertainty planner to deal with combinations of uncertainty.

Finally, as mentioned in Chapter 6, centralized planning approaches for multi-agent planning (such as our RRT-based algorithms) are only applicable to relatively small sized teams, since the dimensionality of the search space increases linearly with the size of the team. To coordinate very large teams of agents, distributed approaches are typically used, which can produce highly suboptimal results. Instead, Kalra et al. [Kalra et al., 2005] are currently investigating the applicability of distributed planning approaches that employ centralized planners for dynamically-formed sub-teams to provide efficient, high quality solutions for large teams performing complex tasks. Such an approach combines the strengths of distributed approaches, in terms of efficiency and robustness, with centralized approaches, in terms of solution quality and feasibility. This is a promising and active area of research that can benefit greatly from the planning approaches described in this thesis.

There are also ways in which the planning algorithms and techniques described in this thesis can better be applied to mobile robot navigation in real environments.

Firstly, in this thesis we have described a number of efficient graph-based search algorithms along with how these algorithms can be used for mobile robot planning and navigation. An important component of this application is extracting the graph to plan over. In particular, deriving appropriate traversal costs for the edges in this graph is of paramount importance to ensure that sensible, safe paths are produced. In general, much parameter tuning is required to take onboard local perception data and extract these costs to be used in planning. This process is made much more complex when additional sources of informa-

tion are being used, such as long range perception or prior map information. Constructing a mapping from such sources of information into costs is very challenging. One approach that has recently shown real promise is to *learn* this mapping. In particular, Sofman et al. [Sofman et al., 2006] present a technique that uses information obtained by a vehicle that has traversed through and observed some areas of an environment to update its cost estimate of areas that have not yet been been traversed (but for which there exists some prior map information). The idea is to use the onboard local perception system and the prior map information to learn the mapping from prior map features to traversal costs, so that traversal costs of regions for which only prior map information exists can be estimated. This idea of using local areas of accurate cost information to help update distant areas of less-accurate information was first proposed by Wellington and Stentz [Wellington and Stentz, 2003] and was used to great effect by the Stanford Racing Team's Grand Challenge vehicle to combine information from long-range vision sensors with short-range, high-fidelity laser range finders [Dahlkamp et al., 2006]. This general strategy shows real promise for tackling one of the more tedious, hand-tuned aspects of outdoor navigation.

Secondly, a very common approach to mobile robot navigation that we have discussed in this thesis is to combine an approximate global path planner with a high-fidelity local planner. In general, this approach works quite well and the coupling provides smooth, goal-directed trajectories for the vehicle. However, there are certainly situations where the mismatch between the global and local fidelities can cause problems. In Chapter 3 we described a couple existing approaches for improving upon this strategy, in particular using the global plan to seed a higher-fidelity planner that plans out all the way to the goal. However, often performing this high-fidelity planning over an extended distance is not computationally feasible. Instead, one possibility is to take the plan returned by the global planner and use the local planner to track this plan in simulation out to the goal. If this fails, then the area in which it fails can be updated in the global planner's search space and the global planner can be used to replan a new global solution. This new solution can then be tracked by the local planner and the entire process can iterate until a solution is provided by the global planner that can be tracked. This approach has the advantage of only performing global planning in the lower-fidelity search space while also providing globally-feasible plans in the higher-fidelity space, and appears quite promising.

Appendix A

Proofs of Theorems

A.1 The Delayed D* Algorithm

We first define terms used in our proofs and introduce some heuristic properties. All the proofs presented in this section originally appeared in [Ferguson and Stentz, 2004a].

In the following we assume Delayed D* operates on a finite size graph. The set of states is denoted by S. $Succ(s)$ denotes the set of successors of state $s \in S$ and $Pred(s)$ denotes the set of predecessors of state s.

For any pair of states $s, s' \in Succ(s)$ the cost between the two needs to be positive: $c(s, s') > 0$. $c^*(s, s')$ denotes the cost of a shortest path from s to s'. For $s = s'$ we define $c^*(s, s') = 0$. $g^*(s)$ denotes the cost of a shortest path from s to s_{goal}. $h(s, s')$ denotes the heuristic cost from state s to state s'. For simplicity, $h(s)$ denotes the heuristic cost from the start state, s_{start}, to a state s.

We call a heuristic function h admissible if and only if it does not overestimate the shortest path cost between *any* two states, i.e., if and only if $h(s, s') \leq c^*(s, s')$ for all $s, s' \in S$. We call a heuristic function h backward consistent if and only if the following holds: $h(s_{start}) = 0$, and $h(s) \leq h(s') + c(s', s)$, for all states $s \in S$ and $s' \in Pred(s)$. Note that backward consistent heuristics are also admissible.

One property of backward consistent heuristics that will be useful to us later is that, for any states $s \in S$, $s' \in Pred(s)$, we have $h(s) \leq h(s') + c^*(s', s)$. We prove this below.

Lemma 1. *If h is a backward consistent heuristic, then for any states $s, s' \in S$, $h(s) \leq h(s') + c^*(s', s)$.*

Proof. By contradiction. Assume there exist some states $s, s' \in S$ such that $h(s) > h(s') + c^*(s', s)$.

Find the state $s^* \in S$ closest to s' along a shortest path from s' to s for which $h(s^*) > h(s') + c^*(s', s^*)$. Let $s_p \in S$ be the predecessor state to s^* along this shortest path. Then,

$h(s^*) \leq h(s_p) + c(s_p, s^*)$ $\quad$ h is backward consistent

$$
\begin{aligned}
&\leq (h(s') + c^*(s', s_p)) + c(s_p, s^*) && s_p \text{ has } h(s_p) \leq h(s') + c^*(s', s_p) \\
&= h(s') + (c^*(s', s_p) + c(s_p, s^*)) && \text{rearranging} \\
&= h(s') + c^*(s', s^*) && \text{since } s_p \text{ is predecessor of } s^* \text{ along path.}
\end{aligned}
$$

But we chose s^* such that $h(s^*) > h(s') + c^*(s', s^*)$. Contradiction. Thus, for all states $s, s' \in S$, $h(s) \leq h(s') + c^*(s', s)$. ■

We now prove a number of properties of the Delayed D* algorithm, beginning with its termination. We assume the heuristic function used is nonnegative and backward consistent, and that our state space is finite. Pseudocode of the algorithm has been reproduced in Figures A.1 and A.2.

Lemma 2. *The* ComputePathDelayed *(CPD) function of the Delayed D* algorithm always terminates.*

Proof: By contradiction. Assume that CPD never terminates. We show this results in a contradiction.

At any point in time, we can partition the state space into two sets:

S_1: those states that will be expanded again in a finite amount of time

S_2: those states that will never be expanded again.

Further, because of our assumption, we know that set S_1 will always be nonempty. Let's choose our point in time, t, to be sufficiently large so that set S_2 is maximized, i.e., after time t *all* states that are only expanded a finite number of times will not be expanded again.

We know from our construction of set S_1 that all members of this set will be expanded again in a finite amount of time after time t. Let t' be the first point in time after t at which all states in S_1 have been expanded at least once since time t. Finally, select t^* to be the first point in time after t' at which there is an overconsistent state at the top of the queue.

Such a t^* must exist. To see this, select a time $t'' > t'$ at which all states in S_1 have been expanded at least once since time t'. Now, each state can only be expanded once as an underconsistent state before it must be expanded as an overconsistent state (since underconsistent states have their v-values set to ∞, so the next time they are made inconsistent they must have their g-values less than their v-values). Thus, either some state in S_1 was expanded as an overconsistent state between t' and t'', or the next state expanded must be an overconsistent state. In either case, we have a $t^* \leq t''$ at which there is an overconsistent state at the top of the queue.

Denote the overconsistent state at the top of the queue s. Since s is overconsistent, we know that $v(s) > g(s)$. Now, $g(s) = c(s, s') + v(s')$, where s' is some successor of s. We examine this state s'. Either s' is inconsistent at time t^*, or it is not.

UpdateSetMembership(s)

1 if ($v(s) \neq g(s)$)
2 insert/update s in $OPEN$ with key(s);
3 else
4 if ($s \in OPEN$) remove s from $OPEN$;

UpdateSetMembershipLower(s)

5 if ($v(s) > g(s)$)
6 insert/update s in $OPEN$ with key(s);
7 else if ($v(s) = g(s)$)
8 if ($s \in OPEN$) remove s from $OPEN$;

ComputePathDelayed()

9 while(key(s_{start}) > $\min_{s \in OPEN}$(key(s)) OR $v(s_{start}) < g(s_{start})$)
10 remove s with the smallest key(s) from $OPEN$;
11 if ($v(s) > g(s)$)
12 $v(s) = g(s)$;
13 for each predecessor s' of s
14 if s' was never visited before
15 $v(s') = g(s') = \infty$; $bp(s') =$ **null**;
16 if ($g(s') > c(s', s) + v(s)$)
17 $bp(s') = s$;
18 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembershipLower(s');
19 else
20 $v(s) = \infty$; UpdateSetMembership(s);
21 for each predecessor s' of s
22 if s' was never visited before
23 $v(s') = g(s') = \infty$; $bp(s') =$ **null**;
24 if ($bp(s') = s$)
25 $bp(s') = \text{argmin}_{s'' \in Succ(s')} c(s', s'') + v(s'')$;
26 $g(s') = c(s', bp(s')) + v(bp(s'))$; UpdateSetMembership($s'$);

Figure A.1: (Reproduction) Delayed D*: ComputePathDelayed function

Case 1: s' is inconsistent

If s' is inconsistent, it is either overconsistent or underconsistent. If it is overconsistent, i.e., $v(s') > g(s')$, then it must be on the queue, since any time a state is made overconsistent it is immediately added (Figure A.1 lines 1 - 2; 5 - 6). But if it is overconsistent, then its key is:

$$
\begin{aligned}
key(s') &\doteq [min(v(s'), g(s')) + h(s'), min(v(s'), g(s'))] \\
&\doteq [g(s') + h(s'), g(s')] && v(s') > g(s') \\
&\dot{\leq} [g(s') + h(s) + c^*(s, s'), g(s')] && \text{heuristic backward consistent} \\
&\dot{<} [v(s') + h(s) + c^*(s, s'), v(s')] && v(s') > g(s')
\end{aligned}
$$

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + h(s_start, s); g(s)];
 3 else
 4   return [v(s) + h(s_start, s); v(s)];
FindRaiseStatesOnPath()
 5 s = s_start; raise = FALSE; CLOSED = ∅;
 6 while (s ≠ s_goal)
 7   add s to CLOSED;
 8   bp(s) = argmin_{s'∈Succ(s)}(c(s, s') + v(s'));
 9   g(s) = c(s, bp(s)) + v(bp(s));
10   if (v(s) ≠ g(s))
11     UpdateSetMembership(s);
12     raise = TRUE;
13   if (bp(s) ∈ CLOSED)
14     return raise;
15   else
16     s = bp(s);
17 return raise;
Main()
18 g(s_start) = v(s_start) = ∞; v(s_goal) = ∞; bp(s_goal) = bp(s_start) = null;
19 g(s_goal) = 0; OPEN = ∅;
20 insert s_goal into OPEN with key(s_goal);
21 forever
22   ComputePathDelayed();
23   raise = FindRaiseStatesOnPath();
24   while (raise = TRUE)
25     ComputePathDelayed();
26     raise = FindRaiseStatesOnPath();
27   wait for changes in edge costs;
28   for all directed edges (u, v) with changed edge costs
29     update the edge cost c(u, v);
30     if (u ≠ s_goal)
31       if u was never visited before
32         v(u) = g(u) = ∞; bp(u) = null;
33       bp(u) = argmin_{s'∈Succ(u)} c(u, s') + v(s');
34       g(u) = c(u, bp(u)) + v(bp(u)); UpdateSetMembershipLower(u);
```

Figure A.2: (Reproduction) Delayed D*: Main function

$$
\begin{aligned}
&\doteq [(c^*(s,s') + v(s')) + h(s), v(s')] && \text{rearranging}\\
&\dot{<} [g(s) + h(s), g(s)] && g(s) \text{ computed from } v(s')\\
&\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] && v(s) > g(s)
\end{aligned}
$$

$\doteq key(s)$. definition of key value.

If this is the case, then we have $key(s') \dot{<} key(s)$ when both s and s' are on the queue, so s would not have been at the top of the queue. Contradiction.

If s' is underconsistent, i.e., $v(s') < g(s')$, then it may or may not be on the queue. If it *is* on the queue, then its key is:

$$
\begin{aligned}
key(s') &\doteq [min(v(s'), g(s')) + h(s'), min(v(s'), g(s'))] \\
&\doteq [v(s') + h(s'), v(s')] && v(s') < g(s') \\
&\dot{\leq} [v(s') + h(s) + c^*(s, s'), v(s')] && \text{heuristic backward consistent} \\
&\doteq [(c^*(s, s') + v(s')) + h(s), v(s')] && \text{rearranging} \\
&\dot{<} [g(s) + h(s), g(s)] && g(s) \text{ computed from } v(s') \\
&\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] && v(s) > g(s) \\
&\doteq key(s). && \text{definition of key value.}
\end{aligned}
$$

As above, $key(s') \dot{<} key(s)$ when both s and s' are on the queue, so s would not have been at the top of the queue. Contradiction.

If s' is *not* on the queue, then the only way we could have $v(s') < g(s')$ is if s' was underconsistent with the values $\{v(s'), g(s')\}$ when CPD was called. This is because, during the course of the algorithm, any time a state is made underconsistent it is immediately added to the queue (Figure A.1 lines 1 - 2). But we know that at time t^*, all states in S_1 have been expanded at least once. Thus, state s' could not possibly still have these initial values without being on the queue. Contradiction.

Case 2: s' is consistent

If s' is consistent at time t^*, then let us consider how it could ever become inconsistent again. There are two possibilities. The g-value of state s' could increase above its v-value, or it could decrease below this value.

For the g-value of state s' to increase, its current best successor state must have its v-value increase. In other words, state s'' for which $g(s') = c(s', s'') + v(s'')$ must have its v-value increase. Now, either state s'' is underconsistent at time t^* or it is not. If it is not, then by the same argument as in the previous lines, *its* current best successor must have its v-value increase. We can repeat this line of reasoning to conclude that there must be *some* underconsistent state s^* at time t^* that is a *direct descendant* of state s'[1]. This state s^* must be a direct descendent or else its underconsistency would have no effect on state s'. We pick the first such underconsistent direct descendent state, s^*, i.e., the one with highest v-value.

Because any underconsistent states at time t^* must be on the queue (see argument at end of Case 1 above), this state s^* must be on the queue at time t^*. It's key value is:

[1]In other words, there exists a state s^* that can be reached from state s' by always moving from the current vertex x, starting at s', to some successor y that minimizes $c(x, y) + v(y)$.

$$
\begin{aligned}
key(s^*) &\doteq [min(v(s^*), g(s^*)) + h(s^*), min(v(s^*), g(s^*))] & \\
&\doteq [v(s^*) + h(s^*), v(s^*)] & v(s^*) < g(s^*) \\
&\dot{\leq} [v(s^*) + h(s') + c^*(s', s^*), v(s^*)] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s', s^*) + v(s^*)) + h(s'), v(s^*)] & \text{rearranging} \\
&\dot{<} [v(s') + h(s'), v(s')] & s^* \text{ is direct descendant of } s' \\
&\dot{\leq} [v(s') + h(s) + c^*(s, s'), v(s')] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s, s') + v(s')) + h(s), v(s')] & \text{rearranging} \\
&\dot{<} [g(s) + h(s), g(s)] & g(s) \text{ computed from } v(s') \\
&\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] & v(s) > g(s) \\
&\doteq key(s). & \text{definition of key value.}
\end{aligned}
$$

Thus, $key(s^*) \dot{<} key(s)$ when both s and s^* are on the queue, so s would not have been at the top of the queue. Contradiction.

For the g-value of state s' to decrease, some successor of s' would need to lower *its* cost value (i.e., state s'' such that $v(s') > v_{new}(s') = c(s', s'') + v_{new}(s'')$). We can repeat this reasoning to create a sequence of states, s', s'', s''', etc. Eventually, one state in this sequence has to be on the queue in order to bring about the changes to those preceding it.

Further, during the course of the algorithm, states can only be made overconsistent when an overconsistent state is expanded (Figure A.1 lines 11 - 18)[2]. Thus, some state in this sequence must be on the queue at time t^*, with its g-value holding its future v-value (which we have denoted v_{new}).

So this overconsistent state, call it s^*, must exist on the queue when s is expanded. Its key value is:

$$
\begin{aligned}
key(s^*) &\doteq [min(v(s^*), g(s^*)) + h(s^*), min(v(s^*), g(s^*))] & \\
&\doteq [g(s^*) + h(s^*), g(s^*)] & v(s^*) > g(s^*) \\
&\doteq [v_{new}(s^*) + h(s^*), v_{new}(s^*)] & g(s^*) = v_{new}(s^*) \\
&\dot{\leq} [v_{new}(s^*) + h(s') + c^*(s', s^*), v_{new}(s^*)] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s', s^*) + v_{new}(s^*)) + h(s'), v_{new}(s^*)] & \text{rearranging} \\
&\dot{<} [v_{new}(s') + h(s'), v_{new}(s')] & v_{new}(s') \text{ caused by } v_{new}(s^*) \\
&\dot{\leq} [v_{new}(s') + h(s) + c^*(s, s'), v_{new}(s')] & \text{heuristic backward consistent} \\
&\dot{\leq} [(c^*(s, s') + v_{new}(s')) + h(s), v_{new}(s')] & \text{rearranging} \\
&\dot{<} [g(s) + h(s), g(s)] & g(s) \text{ lowered by } v_{new}(s') \\
&\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] & v(s) > g(s)
\end{aligned}
$$

[2]when an underconsistent state s is expanded, the predecessor states which use s for their g-values are updated, but there is no way this could result in a decrease of their g-values: if the cost of using some other state is less than the cost of using s given its previous $v(s)$ value, then that other state has had its v-value decrease since the last time the g-value for the current state was computed. But if its v-value has decreased, then when it decreased the g-value for the current state would have been updated. Thus, the g-value of the current state can only increase.

$\doteq key(s)$. definition of key value.

Thus, $key(s^*) \dot{<} key(s)$ when both s and s^* are on the queue, so s would not have been at the top of the queue. Contradiction.

Conclusion:

Our assumption that set S_1 is nonempty has led to a contradiction. Therefore, set S_1 must be empty and so CPD must terminate in finite time. ■

Lemma 3. *After the* ComputePathDelayed *function terminates, for all* $s \in S$ *with* $key(s) \dot{<} key(s_{start})$ *we have* $v(s) \leq g(s)$.

Proof. Suppose s is a state with $v(s) > g(s)$ after *ComputePathDelayed* has terminated. We wish to show that $key(s) \dot{\geq} key(s_{start})$.

There are only three possible ways in Delayed D* that a state can ever find itself with its v-value greater than its g-value. Firstly, it could have had its g-value lowered to become less than its v-value by a changed edge cost, at Figure A.2 line 34. If this occurred, the state would have been immediately added to the queue.

Secondly, it could have had its g-value lowered to become less than its v-value by the expansion of an overconsistent successor state, at Figure A.1 lines 11 - 18. Again, if this occurred, the state would have been added to the queue at line 6.

Finally, it could have been expanded as an underconsistent state and had its v-value set to ∞, at Figure A.1 line 20. If this occurred, the state would have had its g-value previously updated at lines 19 - 26 when it became underconsistent. If its new g-value was less than ∞, the state would have been put back on the queue at line 6.

If s has $v(s) > g(s)$ after CPD terminates, then one of these three events had to have occurred most recently. In other words, the last time state s had its g or g-value change, it had to have been put back on the queue. But this means that it was on the queue when the function terminated. Since the function doesn't terminate until the minimum key value on the queue is at least as large as the key of the start state, we must conclude that $key(s) \dot{\geq} key(s_{start})$.

Thus, for all $s \in S$ with $key(s) \dot{<} key(s_{start})$, $v(s) \leq g(s)$ when fnComputePathDelayed terminates. ■

Lemma 4. *After the* ComputePathDelayed *function terminates, for all* $s \in S$ *with* $key(s) \dot{<} key(s_{start})$ *we have* $v(s) \leq g^*(s)$.

Proof. By contradiction. Assume there is some nonempty set of states R containing all $s \in S$ such that $key(s) \dot{<} key(s_{start})$ and $v(s) > g^*(s)$ after CPD terminates. We know that we can also define a nonempty set of states O containing all $s \in S$ such that $key(s) \dot{<} key(s_{start})$ and $v(s) \leq g^*(s)$. This set O is guaranteed to be nonempty because, at the very least, the goal will be a member.

Suppose s is an element in R for which an *optimal successor* is in set O[3]. Such an s must exist, as the optimal successors of each state in R can be followed to the goal, which is in set O, so there must be some state in R for which an optimal successor is in set O[4].

Now, the optimal successor of this state s, call this s', has $v(s') \leq g^*(s')$ as it is in set O. But when s' had its v-value set to this value, it would have updated the g-values of all possible predecessor states (Figure A.1 lines 11 - 18). Each of these states would use $v(s')$ to update its g-value if this would provide a lower value than its current g-value. Thus,

$$\begin{aligned} g(s) &\leq c(s,s') + v(s') && g(s) \text{ updated to be at least as low as } c(s,s') + v(s') \\ &\leq c(s,s') + g^*(s') && v(s') \leq g^*(s') \\ &= g^*(s) && s' \text{ is optimal successor of } s. \end{aligned}$$

But this means that s is *not* in R. Contradiction. Thus, our assumption that set R is nonempty is incorrect. ■

Lemma 5. *The* FindRaiseStatesOnPath *(FRSOP) function always terminates, and when it does, if no underconsistent states have been added to the queue, then an optimal solution path can be followed from s_{start} to s_{goal} by moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s,s') + v(s')$.*

Proof. FRSOP always terminates, since the state space is finite and FRSOP terminates as soon as any state is visited twice (Figure A.2 lines 13 - 14). If FRSOP terminates under this condition, then at least one of the states visited must have been added to the queue, since it is impossible to have a loop in our backpointers without some state in the loop having an inconsistent value.

If FRSOP does not terminate under the above condition and no states have been added to the queue, then we have arrived at the goal by starting at s_{start} and repeatedly moving from the current state s to any successor s' that minimizes $c(s,s') + v(s')$ (Figure A.2 line 8). Further, we have not encountered any states s along this path with $v(s) < g(s)$ (line 10). Now, all states along this path must also have key values less than $key(s_{start})$, since they each contribute to the current v-value of state s_{start}. Thus, according to Lemma 4, each of these states s has $v(s) \leq g^*(s)$. But if we were able to traverse the entire path without encountering *any* state s with $v(s) < g(s)$, then every state on this path is consistent, and the v-value of each state must in fact equal its actual cost when following this path. Since the actual cost from any state s cannot possibly be less than $g^*(s)$, we must have $v(s) = g^*(s)$ for each state along this path, including the state s_{start}. Since these costs were derived from following the traversed path, this path must be an optimal path. ■

[3]In other words, if all states were consistent and had their optimal v-values, then $g^*(s) = c(s,s') + g^*(s')$, for some successor s'. We call this state s' an *optimal successor* of s.

[4]The only exception is if *no* element in set R has a path to the goal, in which case all elements in O have g^*-values of ∞, so clearly cannot have their v-values greater than their g^*-values - a contradiction.

Theorem 1. *The Delayed D* algorithm always terminates and an optimal solution path can then be followed from s_{start} to s_{goal} by always moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + v(s')$.*

Proof. The only remaining portion of the Delayed D* algorithm that we need to show terminates is the loop at Figure A.2 lines 24 - 26. We have already shown, in Lemma 2, that CPD will always terminate. Further, Lemma 4 proved that when it does terminate, we have $v(s) \leq g^*(s)$, for all $s \in S$ such that $key(s) < key(s_{start})$. Now, FRSOP must also terminate, and when it does, either some underconsistent states have been found along the current path from s_{start} to s_{goal}, or an optimal solution path exists (by Lemma 5). In the latter case, the *raise* flag is set to $false$ and the Delayed D* update phase (the loop at lines 24 - 26) terminates. In the former case, the underconsistent states are added to the queue and CPD is called. We can show that this loop will only be performed a finite number of times as follows.

Consider the first underconsistent state encountered along the path traversed in FRSOP the first time the loop is entered. Call this state s. Since this state was reached from the start state, s_{start}, by always moving from the current vertex x to some successor y that minimizes $c(x, y) + v(y)$, (Figure A.2 line 8) it is a direct descendant of s_{start}. Since this state is the *first* underconsistent direct descendant of s_{start} along this path, we must have $v(s_{start}) \geq c^*(s_{start}, s) + v(s)$[5]. Then,

$$
\begin{aligned}
key(s) &\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] & \\
&\doteq [v(s) + h(s), v(s)] & v(s) < g(s) \\
&\dot{\leq} [v(s) + h(s_{start}) + c^*(s_{start}, s), v(s)] & h \text{ is backward consistent} \\
&\doteq [(v(s) + c^*(s_{start}, s)) + h(s_{start}), v(s)] & \text{rearranging} \\
&\dot{<} [v(s_{start}) + h(s_{start}), v(s_{start})] & v(s_{start}) \geq c^*(s_{start}, s) + v(s) \\
&\doteq key(s_{start}) & \text{definition of key value}
\end{aligned}
$$

where $key(s_{start})$ is the key of the start state before CPD is called. Since the only difference in CPD between the upcoming call and its last call is that there have been some underconsistent states added to the queue, there is no way that the key value of the start state can *decrease* during this call to CPD. Thus, since $key(s) \dot{<} key(s_{start})$ when CPD is called, and $key(s)$ cannot decrease until $v(s)$ decreases (which will not happen until s is expanded), we know that s will be expanded during this call of CPD. This means that s is made overconsistent at some point during this call of CPD.

Now, s may be made underconsistent again, during this call of CPD or some subsequent call. But, during this update phase (i.e., while the loop at Figure A.2 lines 24 - 26 is still

[5]Otherwise, if $v(s_{start}) < c^*(s_{start}, s) + v(s)$, then there is some other state s' between s_{start} and s along this path with $v(s') < g(s')$. Contradiction.

being performed), it will never again be the first underconsistent state encountered along the path traversed in FRSOP. This is because, if s is made underconsistent again, then it is automatically put back on the queue, at Figure A.1 lines 26; 1 - 2. If s is then later encountered as the first underconsistent state encountered along the path traversed in FRSOP, then from before we know that its key value is:

$$
\begin{aligned}
key(s) &\doteq [min(v(s), g(s)) + h(s), min(v(s), g(s))] \\
&\doteq [v(s) + h(s), v(s)] && v(s) < g(s) \\
&\dot{\leq} [v(s) + h(s_{start}) + c^*(s_{start}, s), v(s)] && h \text{ is backward consistent} \\
&\doteq [(v(s) + c^*(s_{start}, s)) + h(s_{start}), v(s)] && \text{rearranging} \\
&\dot{<} [v(s_{start}) + h(s_{start}), v(s_{start})] && v(s_{start}) \geq c^*(s_{start}, s) + v(s) \\
&\doteq key(s_{start}) && \text{definition of key value.}
\end{aligned}
$$

But we know that s is on the queue, so if its key value is less than the key value of the start state, then it would have been expanded the previous call to CPD. Contradiction. Thus, once a state has been the first underconsistent state encountered along the path traversed in FRSOP, it will never again be in this position. Since our state space is finite, this means that FRSOP can only return *true* a finite number of times. Thus, eventually FRSOP will return $false$ and the entire update phase will terminate, leaving an optimal solution path from the start to the goal (by Lemma 5). ■

A.2 The Field D* Algorithm

Theorem 2. *Let* $\{x^*, y^*\} = argmin_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + yv(s_2) + (1-y)v(s_1)]$, $x \in [0,1]$, $y \in [0,1]$, *where* s_1, s_2, b, *and* c *are as defined in the text, and* $v(s_1)$ *and* $v(s_2)$ *are optimal path costs for* s_1 *and* s_2 *given our linear interpolation assumption. Then* $x^* \in \{0,1\}$ *or* $y^* \in \{0,1\}$.

Proof. To prove this, it is useful to transform our original path planning problem (see Figure A.3). If we pretend that the cost of traversing the edge $\overrightarrow{s_1 s_2}$ is the difference in cost between the two states s_1 and s_2, then our original problem can be solved by finding the cheapest path from s through s_2. To see this, let $f = v(s_1) - v(s_2)$. Then our path cost minimization (equation 3.3) can be written as:

$$\min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + (1-y)f + v(s_2)]. \tag{A.1}$$

This is equivalent to computing the minimum-cost path from s through s_2, where portions of the path traveled along the edge $\overrightarrow{s_1 s_2}$ (e.g. between s_y and s_2) use a traversal cost

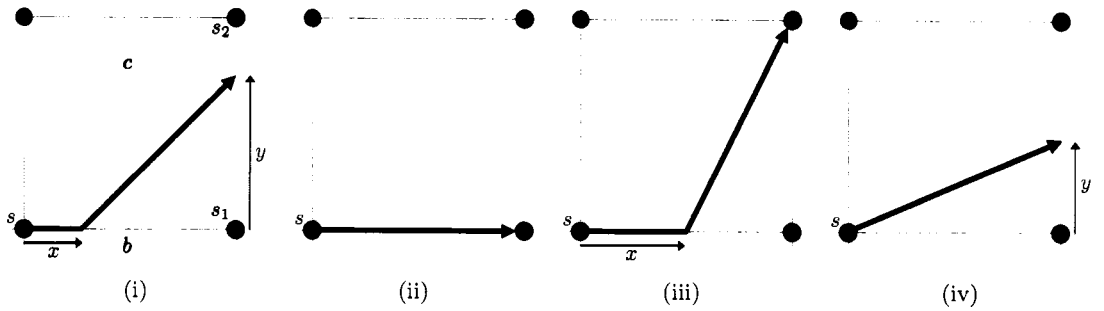


Figure A.3: Computing the path cost of state s using the path cost of two of its neighbors, s_1 and s_2, and the traversal costs c of the center cell and b of the bottom cell. Illustrations (ii) through (iv) show the possible optimal paths from s to edge $\overrightarrow{s_1 s_2}$. This is a reproduction of Figure 3.12.

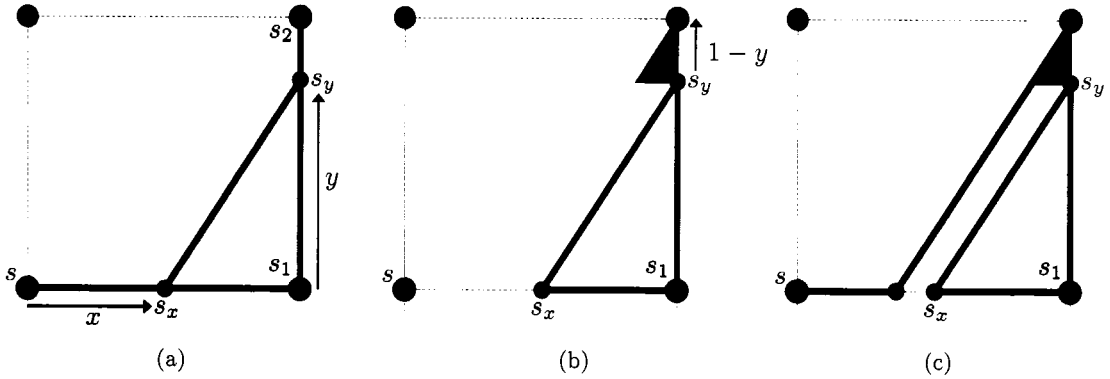


Figure A.4: (a) Imagine the blue path is the optimal path, traveling along the bottom edge to s_x then across the center cell to s_y, then up the right edge to s_2. Notice the triangle formed between vertices s_x, s_1, and s_y. (b) We can create a scaled version of this triangle with a vertical edge length of $1 - y$. (c) Combining the hypotenuses of the two triangles shown in (b) produces a lower-cost path than the one shown in (a), forcing a contradiction. This is a reproduction of Figure 3.13.

of f[6]. If $f < 0$ then it is always cheapest to take a direct route to s_1, then travel along the entirety of the edge $\overrightarrow{s_1 s_2}$.

Now, assume the optimal path involves traveling along parts of both the bottom and right edges *and* some cutting across part of the center cell (i.e., $x^* \in (0,1)$ and $y^* \in (0,1)$). Thus, the path travels from s along the bottom edge some distance $x \in (0,1)$ to point s_x, then cuts across the center cell to arrive at a point s_y on the right edge some distance $y \in (0,1)$ from s_1, then travels along the right edge to s_2 (see Figure A.4(a)). Since this path is optimal, the cost of taking the straight-line path from s_x through the center cell to

[6]If $f > r$, where r is the traversal cost of the right cell, then the path cost $v(s_1)$ is clearly suboptimal. This is a contradiction.

s_y must be cheaper than going from s_x along the bottom edge to s_1 then up the right edge to s_y. Thus, we have the following relationship:

$$c\sqrt{(1-x)^2+y^2} \leq (1-x)b + yf \tag{A.2}$$

The straight-line path between s_x and s_y, along with the line between s_x and s_1 and the line between s_1 and s_y, define a right angled triangle. We know that since the cost of the weighted hypotenuse $\overrightarrow{s_x s_y}$ is cheaper than the combined costs of the weighted sides $\overrightarrow{s_x s_1}$ and $\overrightarrow{s_1 s_y}$, this will be the case if we were to scale the size of the triangle by any amount (maintaining the same ratios of side lengths).

Assume without loss of generality that $(1-y) < x$, so that s_y is closer to s_2 than s_x is to s. Consider a scaled version of this triangle with a vertical edge of length $(1-y)$ (see Figure A.4(b)). The horizontal edge of this triangle will have length $(1-y)\frac{(1-x)}{y}$. Since this is a scaled version of our original triangle, the weighted cost of the hypotenuse of this new triangle is cheaper than the combined weighted costs of the horizontal and vertical edges[7]. But this means we could combine the hypotenuse of this new triangle with our previous hypotenuse and construct a path that went from s along the bottom edge a distance of $x-(1-y)\frac{(1-x)}{y}$ then straight to s_2, and the cost of this path would be *less* than the cost of our original (optimal) path (see Figure A.4(c)). This is a contradiction. Thus, it is not possible that *both* x^* and y^* will be in the range $(0,1)$. ■

A.3 The Anytime D* Algorithm

The proofs included in this section originally appeared in [Likhachev et al., 2005b] and were derived by Max Likhachev. We have included them here for completeness. They have been split into several subsections. Section A.3 proves that the *ComputePath* function in Figure A.5 correctly maintains its variables. Sections A.3 and A.3 prove the key properties of the algorithm. Finally, Section A.3 proves several properties regarding the efficiency of the algorithm.

Notation

In the following we assume Anytime D* operates on a finite size graph. The set of states is denoted by S. $Succ(s)$ denotes the set of successors of state $s \in S$ and $Pred(s)$ denotes the set of predecessors of state s.

For any pair of states $s, s' \in Succ(s)$ the cost between the two needs to be positive: $c(s,s') > 0$. $c^*(s,s')$ denotes the cost of a shortest path from s to s'. For $s = s'$ we

[7]Note that we have drawn this new triangle above our previous triangle only to show how they could be combined to form a single triangle. The costs of the vertical and horizontal edges of the new triangle are derived from the values f and b, respectively.

UpdateSetMembership(s)

```
1  if (v(s) ≠ g(s))
2    if (s ∉ CLOSED) insert/update s in OPEN with key(s);
3    else if (s ∉ INCONS) insert s into INCONS;
4  else
5    if (s ∈ OPEN) remove s from OPEN;
6    else if (s ∈ INCONS) remove s from INCONS;
```

ComputePath()

```
7  while(key(s_goal) > min_{s∈OPEN}(key(s)) or v(s_goal) < g(s_goal))
8    remove s with the smallest key(s) from OPEN;
9    if (v(s) > g(s))
10     v(s) = g(s); CLOSED←CLOSED ∪ {s};
11     for each successor s' of s
12       if s' was never visited by Anytime D* before
13         v(s') = g(s') = ∞; bp(s') = null;
14       if (g(s') > v(s) + c(s, s'))
15         bp(s') = s;
16         g(s') = v(bp(s')) + c(bp(s'), s'); UpdateSetMembership(s');
17   else
18     v(s) = ∞; UpdateSetMembership(s);
19     for each successor s' of s
20       if s' was never visited by Anytime D* before
21         v(s') = g(s') = ∞; bp(s') = null;
22       if (bp(s') = s)
23         bp(s') = argmin_{s''∈Pred(s')} v(s'') + c(s'', s');
24         g(s') = v(bp(s')) + c(bp(s'), s'); UpdateSetMembership(s');
```

Figure A.5: (Reproduction) Anytime D*: ComputePath function

define $c^*(s, s') = 0$. $g^*(s)$ denotes the cost of a shortest path from s_{start} to s. The task of Anytime D* is to maintain a path from state s_{start} to state s_{goal} that has a cost of at most $\epsilon * c^*(s_{start}, s_{goal})$. AD* may dynamically change ϵ to trade off the quality of solution with the computational expense of computing it. We restrict the range of ϵ values to $1 \leq \epsilon < \infty$.

In our pseudocode AD* searches forward, from s_{start} towards s_{goal}. Consequently, the provided heuristic values need to be forward consistent: $h(s) \leq c(s, s') + h(s')$ for any $s, s' \in Succ(s)$ and $h(s) = 0$ for $s = s_{goal}$.

AD* maintains two estimates of start distances (the cost of a shortest path from s_{start} to the state) for a state $s \in S$: $g(s)$ and $v(s)$. We call a state s *inconsistent* iff $v(s) \neq g(s)$, *overconsistent* iff $v(s) > g(s)$, *underconsistent* iff $v(s) < g(s)$ and *consistent* iff $v(s) = g(s)$. AD* also maintains back-pointers, $bp(s)$, that point to the predecessor state of s via which a currently found path from s_{start} to s goes.

```
key(s)
 1 if (v(s) ≥ g(s))
 2   return [g(s) + ε · h(s); g(s)];
 3 else
 4   return [v(s) + h(s); v(s)];
Main()
 5 g(s_goal) = v(s_goal) = ∞; v(s_start) = ∞; bp(s_goal) = bp(s_start) = null;
 6 g(s_start) = 0; OPEN = CLOSED = INCONS = ∅; ε = ε_0;
 7 insert s_start into OPEN with key(s_start);
 8 forever
 9   ComputePath();
10   publish ε-suboptimal solution;
11   if ε = 1
12     wait for changes in edge costs;
13   for all directed edges (u, v) with changed edge costs
14     update the edge cost c(u, v);
15     if v ≠ s_start
16       if v was never visited by Anytime D* before
17         v(v) = g(v) = ∞; bp(v) = null;
18       bp(v) = arg min_{s'' ∈ Pred(v)} v(s'') + c(s'', v);
19       g(v) = v(bp(v)) + c(bp(v), v); UpdateSetMembership(v);
20   if significant edge cost changes were observed
21     increase ε or re-plan from scratch (i.e., re-execute Main function);
22   else if (ε > 1)
23     decrease ε;
24   Move states from INCONS into OPEN;
25   Update the priorities for all s ∈ OPEN according to key(s);
26   CLOSED = ∅;
```

Figure A.6: (Reproduction) Anytime D*: Main function

To make the following proofs easier to read we assume that the min operation on an empty set returns ∞, argmin operation on the set consisting of only infinite values returns **null** and any state s with undefined values (in other words, a state that has not been visited yet) has $v(s) = g(s) = \infty$ and $bp(s) =$ **null**. Additionally, in the computation $g(s) = v(bp(s)) + c(bp(s), s)$ it is assumed that if $bp(s) =$ **null** then $g(s)$ is set to ∞.

Low-level Properties

Most of the results in this section are merely stating the correctness of the program state variables such as v-, g-values and bp-values, and *OPEN*, *CLOSED* and *INCONS* sets.

Lemma 6. *For any pair of states s and s', $\epsilon * h(s) \leq \epsilon * c^*(s, s') + \epsilon * h(s')$.*

Proof. The consistency property required of heuristics is equivalent to the restriction that $h(s) \leq c^*(s, s') + h(s')$ for *any* pair of states s, s' and $h(s_{goal}) = 0$ ([Pearl, 1984]). The lemma then follows by multiplying the inequality with ϵ. ■

Lemma 7. *At line 7 in the* ComputePath *function, all v- and g-values are non-negative,* $bp(s_{start}) =$ ***null****,* $g(s_{start}) = 0$ *and for* $\forall s \neq s_{start}$, $bp(s) = \arg\min_{s' \in Pred(s)}(v(s') + c(s', s))$, $g(s) = v(bp(s)) + c(bp(s), s)$.

Proof. The lemma holds the first time line 7 in the *ComputePath* function is executed due to the initialization performed by the *Main* function: the g- and v-values of all states except for s_{start} are infinite, and $v(s_{start}) = \infty$ and $g(s_{start}) = 0$. Also, the bp-values of all states are equal to **null**. In other words, for every state $s \neq s_{start}$, $bp(s) =$ **null** and $v(s) = g(s) = \infty$, and for $s = s_{start}$, $bp(s) =$ **null**, $g(s) = 0$ and $v(s) = \infty$. Thus, the lemma holds when line 7 is executed for the first time.

We will now prove that the lemma continues to hold during the next execution of line 7 and thus holds by induction. Before line 7 is executed next two mutually exclusive possibilities exist. One possibility is that the body of the while loop in *ComputePath* was executed. Another possibility is that the code in between two calls to *ComputePath* in the *Main* function was executed (lines 10-26).

Let us first consider the former possibility, namely another execution of the body of the while loop in the *ComputePath* function. The only places where g-, v- and bp-values are changed and therefore the lemma might be affected are lines 10, 15, 16, 18, 23, and 24. The initialization of newly visited states on lines 13 and 21 does not change the values of states according to our convention of assuming that all states that have not been visited so far have infinite v- and g-values and their bp-values are equal to **null**. Let us now examine each of the lines that do change the values of states.

If $v(s)$ is set to $g(s)$ on line 10, then it is decreased since s is being expanded as overconsistent (i.e., $v(s) > g(s)$ before the expansion according to the test on line 9). Thus, it may only decrease the g-values of its successors. The test on line 14 checks this for each successor of s and updates the bp- and g-values if necessary. Since all costs are positive $bp(s_{start})$ and $g(s_{start})$ can never be changed: it will never pass the test on line 14, and thus is always 0. Since $v(s)$ is set to $g(s)$ it still remains non-negative. Consequently, when the g-values of the successors of s are re-calculated their g-values also remain non-negative.

If $v(s)$ is set to ∞ on line 18, then it either stays the same or increases since it is the "else" case of the test on line 9 (i.e., $v(s) \leq g(s)$ before the expansion of s). (As we will show in later lemmas the inequality will always be strict as no consistent state is ever expanded.) Thus, it may only affect the g- and $bp-$ values of the successors of s if their bp-value is equal to s. The test on line 22 checks this for each successor of s and re-computes the bp- and g-values if necessary. Since $bp(s_{start}) =$ **null** the test will never pass for s_{start} and therefore

$bp(s_{start})$ and $g(s_{start})$ can never be changed. Since $v(s)$ is set to ∞ it remains non-negative. Consequently, when the g-values of the successors of s are re-calculated their g-values also remain non-negative. The lemma thus holds during the next execution of line 7 if the body of the *ComputePath* function while loop was executed in between.

Suppose now that before the next execution of line 7 the code that resides in between lines 10 and 26 in the *Main* function is executed. During this execution costs may change on line 14. The *bp*- and g-values, however, are updated correctly on the following lines 18 and 19. No other code during this execution changes any of the state values that appear in the lemma (the initialization code on line 17 does not change state values according to our assumption that all states that we haven't seen before are assumed to have infinite v- and g-values and *bp*-values equal to **null**.) Hence, independently of the execution path the lemma holds the next time line 7 is executed, and thus it holds always by induction. ■

Lemma 8. *At line 7 in the* ComputePath *function, OPEN and CLOSED are disjoint, OPEN contains only inconsistent states, the union of OPEN and CLOSED contains all inconsistent states (and possibly others) and INCONS contains all and only inconsistent states that are also in CLOSED.*

Proof. We will prove the lemma by induction. Consider the first execution of line 7 in the *ComputePath* function. Due to the initialization performed by the *Main* function, at this point $CLOSED = INCONS = \emptyset$ and *OPEN* contains only s_{start}. Because after the initialization for every state $s \neq s_{start}$, $v(s) = g(s) = \infty$, and for $s = s_{start}$, $v(s) = \infty \neq 0 = g(s)$, *OPEN* contains all and only inconsistent states and the lemma holds.

We now need to show that the lemma continues to hold the next time line 7 in the *ComputePath* function is executed given that the lemma held during all the previous executions of this line. Before line 7 is executed next the executed code could either be the body of the while loop in *ComputePath* or the code in between two calls to *ComputePath* in the *Main* function (lines 10-26).

Let us first consider another execution of the body of the while loop in the *ComputePath* function. In particular, let us examine all the lines of the body of the while loop in *ComputePath* where we change v- or g-values of states or their set membership during the following execution of *ComputePath*. On line 8 we remove a state s from *OPEN*. This state is expanded as either overconsistent or underconsistent.

If state s is overconsistent, then on line 10 we insert it into *CLOSED* but since we set $v(s) = g(s)$ on the same line, the state is consistent. The state is thus correctly not a member of *OPEN* and can not be a member of *INCONS* since at the last execution of line 7 it was a member of *OPEN*, and *OPEN* and *INCONS* were disjoint according to the statement of the lemma. The state s thus does not violate the lemma after line 10 is executed.

If state s is underconsistent, then on line 18 we set $v(s) = \infty$ but also call the *UpdateSetMembership* function on the same line. On all the other lines of *ComputePath* where we modify either v- or g-values of states except for state initialization (lines 13 and 21) we also call *UpdateSetMembership*. The state initialization code does not change the values of states according to our convention of assuming that all states that have not been visited so far have infinite v- and g-values. We therefore only need to show that *UpdateSetMembership* operates consistently with the lemma.

In *UpdateSetMembership* if a state s is inconsistent and is not in *CLOSED* it is inserted into *OPEN*, otherwise it is inserted into *INCONS* (unless it is already there). Since *OPEN* and *CLOSED* were disjoint before we called UpdateSetMemberhip, this procedure ensures that an inconsistent state s does appear in either *OPEN* or *CLOSED* but not both and does appear in *INCONS* if it also belongs to *CLOSED*. If a state s is consistent and belongs to *OPEN* then it is correctly removed from it. It does not belong to *CLOSED* since these sets were disjoint before *UpdateSetMembership* was called. Consequently, s also does not belong to *INCONS* because it is a subset of *CLOSED*. If a state s is consistent and does not belong to *OPEN*, then it may potentially belong to *INCONS*. We check this and remove s from *INCONS* if it is there on line 6.

Suppose now that before the next execution of line 7 in *ComputePath* the code that resides in between lines 10 and 26 in the *Main* function is executed. During this execution state values may change on line 19 (the initialization code on line 17 does not change state values according to our assumption that all states that we haven't seen before are assumed to have infinite v- and g-values). On the same line, however, we again correctly update the set membership by calling *UpdateSetMembership* on the same line. On line 24 we move *INCONS* into *OPEN*. Hence, *INCONS* becomes empty and *OPEN* contains all and only inconsistent states. We then set *CLOSED* to empty set on line 26. This makes the sets to be consistent with the statement of the lemma. ■

Lemma 9. *Suppose an overconsistent state s is selected for expansion on line 8 in ComputePath. Then the next time line 7 in ComputePath is executed $v(s) = g(s)$, where $g(s)$ before and after the expansion of s is the same.*

Proof. Suppose an overconsistent state s (i.e., $v(s) > g(s)$) is selected for expansion. Then on line 10 in *ComputePath* $v(s) = g(s)$, and it is the only place where a v-value changes while expanding an overconsistent state. We, thus, only need to show that $g(s)$ does not change. It could only change if $s \in Succ(s)$ and $g(s) > g(s) + c(s, s)$ (test on line 14 in *ComputePath*) which is impossible since all costs are positive. ■

Lemma 10. *The following properties hold for any two states $s, s' \in S$ and the definition of the procedure key as given in figure A.5:*

(a) if $c^*(s', s_{goal}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ *and* $g(s') > g(s) + \epsilon * c^*(s, s')$, *then* $key(s') > key(s)$;

(b) if $c^*(s', s_{goal}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ *and* $g(s') \geq v(s) + c^*(s, s')$, *then* $key(s') > key(s)$;

Proof. Consider first property (a): we are given two arbitrary states s' and s such that $c^*(s', s_{goal}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, and we need to show that these conditions imply key(s') > key(s). Given the definition of the *key* function in figure A.5 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [g(s) + \epsilon * h(s); g(s)]$. We examine the inequality $g(s') > g(s) + \epsilon * c^*(s, s')$ and add $\epsilon * h(s')$, which is finite since $c^*(s', s_{goal})$ is finite and heuristics are consistent. We thus have $g(s') + \epsilon * h(s') > g(s) + \epsilon * c^*(s, s') + \epsilon * h(s')$ and from Lemma 6 we obtain $g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s)$ that guarantees that the desired inequality holds.

We now prove property (b): we are given two arbitrary states s' and s such that $c^*(s', s_{goal}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$, and we need to show that these conditions imply key(s') > key(s). Given the definition of the *key* function in figure A.5 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. Since $v(s) < g(s)$, $v(s)$ is finite. Consider now the inequality $g(s') \geq v(s) + c^*(s, s')$. Because $v(s) < \infty$ and costs are positive we can conclude that $g(s') > v(s)$. We now add $\epsilon * h(s')$ to both sides of the inequality and use the consistency of heuristics as follows $g(s') + \epsilon * h(s') \geq v(s) + c^*(s, s') + \epsilon * h(s') \geq v(s) + c^*(s, s') + h(s') \geq v(s) + h(s)$. Hence, we have $g(s') + \epsilon * h(s') \geq v(s) + h(s)$ and $g(s') > v(s)$. These inequalities guarantee that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. ■

Main Properties

The following properties guarantee ϵ sub-optimality of each search iteration: every time *ComputePath* returns it has identified a set of states s for which a path from s_{start} to s as defined by backpointers is guaranteed to be of cost no larger than $g(s)$ which in turn is no more than a factor of ϵ greater than the optimal cost $g^*(s)$.

Optimality guarantees for single-shot optimal search algorithms such as Dijkstra's and A* search can often be proven by relying on the property that every time a state s with the smallest priority among all candidates for expansion is selected for expansion, all the states that lie on an optimal path from s_{start} to s have already been expanded and have correct values while all the states that have not been yet expanded have values that are bounded below by their optimal values. As a result, when expanding a state s the predecessor of s that lies on an optimal path from s_{start} to s can easily be identified as the state $s' \in Pred(s)$ that minimizes the sum of the value of s' and the cost $c(s', s)$. Consequently, the value of s during its expansion can be set to the sum of the value of s' and the cost $c(s', s)$.

With Anytime D*, things are more complicated because the *ComputePath* function needs only to compute an ϵ sub-optimal solution and may also encounter states whose values are actually smaller than their respective optimal values since costs may decrease in between search iterations. To deal with this we introduce a set Q:

$$Q = \{u \mid v(u) < g(u) \vee (v(u) > g(u) \wedge v(u) > \epsilon * g^*(u))\} \tag{A.3}$$

The set Q contains all underconsistent states and all those overconsistent states whose v-values are larger than a factor of ϵ of their optimal values, g^*-values. Given such a set, we can formulate a property that is equivalent to the property that we have described above for single-shot optimal search algorithms: if we select any overconsistent or consistent state s whose priority is smaller than or equal to the smallest priority of states in Q, then s has a g-value that is at most a factor of ϵ larger than $g^*(s)$ and the path defined by backpointers from s_{start} to s has a cost no larger than $g(s)$. This is put formally in Lemma 11.

Lemma 12 then builds on this result by showing that *OPEN* is always a superset of Q, and therefore any overconsistent or consistent state s whose priority is smaller than or equal to the smallest priority of states in *OPEN* has a g-value that is at most a factor of ϵ larger than $g^*(s)$ and the path defined by backpointers from s_{start} to s has a cost no larger than $g(s)$. This property can be used to explain the operation of AD* quite simply. When selecting for expansion a state s as a state with the smallest priority among all the states in *OPEN*, AD* guarantees to handle s correctly: if s is overconsistent, then setting $v(s) = g(s)$ makes $v(s)$ at most ϵ sub-optimal and thus removes s from set Q, while if s is underconsistent, then setting $v(s) = \infty$ forces s to become overconsistent and the next time it is selected for expansion it will be overconsistent.

Lemma 11. *At line 7 in the ComputePath function, let Q be defined according to the definition A.3. Then for any state s with $(c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)$ $\forall u \in Q)$, it holds that (i) $g(s) \leq \epsilon * g^*(s)$, (ii) the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$.*

Proof. (i) We first prove statement (i). We prove by contradiction. Suppose there exists an s such that $(c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)\ \forall u \in Q)$, but $g(s) > \epsilon * g^*(s)$. The latter implies that $g^*(s) < \infty$. We also assume that $s \neq s_{start}$ since otherwise $g(s) = 0 = \epsilon * g^*(s)$ from Lemma 7.

Consider a least-cost path from s_{start} to s, $\pi(s_0 = s_{start}, ..., s_k = s)$. The cost of this path is $g^*(s)$. Such path must exist since $g^*(s) < \infty$. We will now show that such path must contain a state s' that is overconsistent and whose v-value overestimates $g^*(s')$ by more than ϵ. As a result, such state is a member of Q. We will then show that its key, on the other hand, must be strictly smaller than the key of s. This, therefore, becomes a contradiction since s, according to the assumptions of the lemma, has a key smaller than

or equal to the key of any state in Q.

Our assumption that $g(s) > \epsilon * g^*(s)$ means that there exists at least one $s_i \in \pi(s_0, ..., s_{k-1})$, namely s_{k-1}, whose $v(s_i) > \epsilon * g^*(s_i)$. Otherwise,

$$\begin{aligned} g(s) = g(s_k) = \min_{s' \in Pred(s)} (v(s') + c(s', s_k)) &\leq \\ v(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * g^*(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * (g^*(s_{k-1}) + c(s_{k-1}, s_k)) = \epsilon * g^*(s_k) &= \epsilon * g^*(s) \end{aligned}$$

Let us now consider $s_i \in \pi(s_0, ..., s_{k-1})$ with the smallest index $i \geq 0$ (that is, the closest to s_{start}) such that $v(s_i) > \epsilon * g^*(s_i)$. We will first show that $\epsilon * g^*(s_i) \geq g(s_i)$. It is clearly so when $i = 0$ according to Theorem 7 which says that $g(s_i) = g(s_{start}) = 0$. For $i > 0$ we use the fact that $v(s_{i-1}) \leq \epsilon * g^*(s_{i-1})$ from the way s_i was chosen,

$$\begin{aligned} g(s_i) = \min_{s' \in Pred(s_i)} (v(s') + c(s', s_i)) &\leq \\ v(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_i) & \end{aligned}$$

We thus have $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, which also implies that $s_i \in Q$.

We now show that key(s) > key(s_i), and therefore arrive at a contradiction. According to our assumption

$$\begin{aligned} g(s) > \epsilon * g^*(s) &= \\ \epsilon * (c^*(s_0, s_i) + c^*(s_i, s_k)) &= \\ \epsilon * g^*(s_i) + \epsilon * c^*(s_i, s_k) &\geq \\ g(s_i) + \epsilon * c^*(s_i, s) & \end{aligned}$$

Hence, we have $g(s) > g(s_i) + \epsilon * c^*(s_i, s)$, $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, $v(s) \geq g(s)$ and $c^*(s, s_{goal}) < \infty$ from theorem assumptions. Thus, from Theorem 10, property (a), it follows that key(s) > key(s_i). This inequality, however, implies that $s_i \notin Q$ since key$(s) \leq$ key(u) $\forall u \in Q$. But this contradicts what we have proven earlier.

(ii) Let us now prove statement (ii). We assume that $g(s) < \infty$ for otherwise the statement holds trivially. Suppose we start following the backpointers starting at s. We need to show that we will reach s_{start} at the cumulative cost of the transitions less than or equal to $g(s)$ (we assume that if we encounter a state with *bp*-value equal to **null** before s_{start} is reached then the cumulative cost is infinite).

We first show that we are guaranteed not to encounter an underconsistent state or a

state with *bp*-value equal to **null** before s_{start} is reached. Once we have this property proven, we will be able to show that the cost of the path is bounded above by $g(s)$ simply from the fact that at each backtracking step in the path the g-value can only be larger than or equal to the sum of the g-value of the state the backpointer points to and the cost of the transition. Consequently, the g-value can never underestimate the cost of the remaining part of the path. The property that we are guaranteed not to encounter an underconsistent state or a state with *bp*-value equal to **null** before s_{start} is reached is based on the fact that any such state will have a key strictly smaller than the key of s or have an infinite g-value. The first case is impossible because key(s) is smaller than or equal to the key of any state in Q and this set already contains all underconsistent states. The second case can also be shown to be impossible quite trivially.

We thus first prove by contradiction the property that we are guaranteed not to encounter an underconsistent state or a state with *bp*-value equal to **null** before s_{start} is reached while following backpointers from s to s_{start}. Suppose the sequence of backpointer transitions leads us through the states $\{s_0 = s, s_1, \ldots, s_i\}$ where s_i is the first state that is either underconsistent or has $bp(s_i) =$ **null** (or both). It could not have been state s since $v(s) \geq g(s)$ from the assumptions of the theorem and $g(s) < \infty$ implies $bp(s) \neq$ **null** according to Lemma 7 (except when $s = s_{start}$ in which case the theorem holds trivially). We now show that s_i can not be underconsistent. Since all the states before s_i are *not* underconsistent and have defined backpointer values we have $g(s) = v(s_1) + c(s_1, s) \geq g(s_1) + c(s_1, s) = v(s_2) + c(s_2, s_1) + c(s_1, s) \geq \ldots \geq v(s_i) + \sum_{k=1..i} c(s_k, s_{k-1}) \geq v(s_i) + c^*(s_i, s)$. If s_i was underconsistent, then we would have had $c^*(s, s_{goal}) < \infty$, $v(s) \geq g(s)$, $v(s_i) < g(s_i)$ and $g(s) \geq v(s_i) + c^*(s_i, s)$, and this, according to Lemma 10 property (b), would imply that $key(s) > key(s_i)$ which means that $s_i \notin Q$ and therefore can not be underconsistent according to the definition of Q. We will now show that $bp(s_i)$ can not be equal to **null** either. Since s_i is not underconsistent $v(s_i) \geq g(s_i)$. From our assumption that $g(s) < \infty$ and the fact that $g(s) \geq v(s_i) + c^*(s_i, s)$ it then follows that $g(s_i)$ is finite. As a result, from Lemma 7 $bp(s_i) \neq$ **null** unless $s_i = s_{start}$. Hence, as we backtrack from s to s_{start} the path defined by backpointers we are guaranteed to have states that are not underconsistent and whose *bp*-values are not equal to **null** except for s_{start}.

We are now ready to show that the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$. Let us denote such path as: $s_0 = s_{start}, s_1, ..., s_k = s$. Since all states on this path are either consistent or overconsistent and their *bp*-values are defined (except for s_{start}), for any i, $k \geq i > 0$, we have $g(s_i) = v(s_{i-1}) + c(s_{i-1}, s_i) \geq g(s_{i-1}) + c(s_{i-1}, s_i)$ from Lemma 7. For $i = 0$, $g(s_i) = g(s_{start}) = 0$ from the same theorem. Thus, $g(s) = g(s_k) \geq g(s_{k-1}) + c(s_{k-1}, s_k) \geq g(s_{k-2}) + c(s_{k-2}, s_{k-1}) + c(s_{k-1}, s_k) \geq ... \geq \sum_{j=1..k} c(s_{j-1}, s_j)$. That is, $g(s)$ is at least as large as the cost of the path from s_{start} to s as defined by backpointers. ■

Lemma 12. *At line 7 in ComputePath, for any state s with $(c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)\ \forall u \in OPEN)$, it holds that (i) $g(s) \leq \epsilon * g^*(s)$, (ii) the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$.*

Proof. Let Q be defined according to the definition A.3. To prove the lemma we will show that Q is a subset of *OPEN* and then appeal to Lemma 11. We will show that Q is a subset of *OPEN* by induction. We will first show that it holds initially because *OPEN* contains all inconsistent states initially and set Q is a subset of those. Afterwards, we will show that any state $s \in CLOSED$ always remains either consistent or overconsistent but with $v(s) \leq \epsilon * g^*(s)$. Given that the union of *OPEN* and *CLOSED* contains all inconsistent states, it is then clear that *OPEN* contains at least all those (and possibly other) inconsistent states that are in Q.

We now prove the lemma. From the definition of set Q it is clear that for any state $u \in Q$ it holds that u is inconsistent (that is, $v(u) \neq g(u)$). According to Lemma 8 and the fact that right before *ComputePath* is called *CLOSED* is always empty (lines 6 and 26 in the *Main* function) when the *ComputePath* function is called *OPEN* contains all inconsistent states. Therefore $Q \subseteq OPEN$, because as we have just said any state $u \in Q$ is also inconsistent. Thus, if any state s has $\text{key}(s) \leq \text{key}(u)\ \forall u \in OPEN$, it is also true that $\text{key}(s) \leq \text{key}(u)\ \forall u \in Q$. From the direct application of Theorem 11 it then follows that the first time line 7 in *ComputePath* is executed the lemma holds.

Also, because during the first execution of line 7 $CLOSED = \emptyset$ (lines 6 and 26 in the *Main* function), the following statement, denoted by (*), trivially holds when line 7 is executed for the first time within any particular call to the *ComputePath* function: for any state $v \in CLOSED$ it holds that $g(v) \leq v(v) \leq \epsilon * g^*(v)$ and $g(v) < v(s') + c^*(s', v)$ $\forall s' \in \{s'' \mid v(s'') < g(s'')\}$. We will later prove that this statement always holds and thus all states $v \in CLOSED$ are either consistent or overconsistent but ϵ sub-optimal (i.e., $v(v) \leq \epsilon * g^*(v)$).

We will now show by induction that the lemma continues to hold for the subsequent executions of line 7 within the current call to *ComputePath*. Suppose the lemma and the statement (*) held during all the previous executions of line 7, and they still hold when a state s is selected for expansion on line 8. We need to show that the lemma holds the next time line 7 in *ComputePath* is executed.

We first prove that the statement (*) still holds during the next execution of line 7. Suppose first we select an overconsistent state s to be expanded. Because it is added to *CLOSED* immediately afterwards, we need to show that it does not violate statement (*). Since when s is selected for expansion on line 8 $\text{key}(s) = \min_{u \in OPEN}(\text{key}(u))$, we have $\text{key}(s) \leq \text{key}(u)\ \forall u \in OPEN$. According to the assumptions of our induction then $g(s) \leq \epsilon * g^*(s)$. From Lemma 9 it then also follows that the next time line 7 is executed $g(s) = v(s) \leq \epsilon * g^*(s)$. To show that $g(s) < v(s') + c^*(s', s)\, \forall s' \in \{s'' \mid v(s'') < g(s'')\}$ after s is

expanded we show that this is true right before s is expanded and therefore since v-values of all states except for s do not change during the expansion of s and $g(s)$ does not change either (Lemma 9) it still holds afterwards. To show that the inequality held before the expansion of s we note that according to our assumptions *CLOSED* contained no underconsistent states and they were all therefore in *OPEN* (Lemma 8); from the way s was selected from *OPEN* it then followed that $\text{key}(s) \leq \text{key}(s') \forall s' \in \{s'' \mid v(s'') < g(s'')\}$; finally, the fact that s was overconsistent ($v(s) > g(s)$) implies that $g(s) < v(s') + c^*(s', s) \forall s' \in \{s'' \mid v(s'') < g(s'')\}$ because otherwise $c^*(s, s_{goal}) < \infty, v(s) > g(s)$, $v(s') < g(s')$ and $g(s) \geq v(s') + c^*(s', s)$ would imply $\text{key}(s) > \text{key}(s')$ according to Lemma 10, property(b). As for the rest of the states the statement (*) follows from the following observations: only v-value of s was changed and s is not underconsistent after its expansion (it is in fact consistent according to Lemma 9); since $g(s)$ decreased during the expansion of s the g-values of its successors could only decrease implying that they could not have violated the statement (*); and finally no other changes to either v- or g-values were done and no operations except for insertion of s were done on *CLOSED*.

Suppose now an underconsistent state s is selected for expansion. Because it is not added to *CLOSED*, we only need to show that statement (*) remains to hold true for all the states that were in *CLOSED* prior to the expansion of s. Since only v-value of s has been changed, none of the v-values of states in *CLOSED* are changed. We will now show that none of their g-values could have changed either. Since prior to the expansion of s, s was underconsistent and statement (*) held by our induction assumptions, it was true that for any state $v \in$ *CLOSED*, $g(v) < v(s) + c^*(s, v)$. This means that $bp(v) \neq s$ (Lemma 7) and therefore the test on line 22 will not pass and $g(v)$ will not change during the expansion of s. Finally, we will now show that the newly introduced underconsistent states could not have violated the statement (*) either. The v-values of states that were underconsistent before s was expanded were not changed (v-value of only s was changed and s could not remain underconsistent as its v-value was set to ∞). Suppose some state s' became underconsistent as a result of expanding s. We need to show that after the expansion of s, for any state $v \in$ *CLOSED* it holds that $g(v) < v(s') + c^*(s', v)$. Since s' became underconsistent as a result of expanding s it must be the case that before the expansion of s $v(s') \geq g(s')$ and $bp(s') = s$ (in order for $g(s')$ to change). Consequently before the expansion of s, $v(s') \geq g(s') = v(s) + c(s, s')$. Since before the expansion of s statement (*) held, for any state $v \in$ *CLOSED* $g(v) < v(s) + c^*(s, v)$. We thus had $g(v) < v(s) + c^*(s, v) \leq v(s) + c(s, s') + c^*(s', v) \leq v(s') + c^*(s', v)$. This continues to hold after the expansion of s since neither $v(s')$ nor $g(v)$ changes during the expansion of s as we have just shown. Hence the statement (*) continues to hold the next time line 7 in *ComputePath* is executed.

We now prove that after s is expanded the lemma itself also holds. We prove it by

showing that Q continues to be a subset of *OPEN* the next time line 7 is executed. According to Lemma 8 *OPEN* set contains all inconsistent states that are not in *CLOSED*. Since, as we have just proved, the statement (*) holds the next time line 7 is executed, all states s in *CLOSED* set have $g(s) \leq v(s) \leq \epsilon * g^*(s)$. Thus, any state s that is inconsistent and has either $g(s) > v(s)$ or $v(s) > \epsilon * g^*(s)$ (or both) is guaranteed to be in *OPEN*. Now consider any state $u \in Q$. As we have shown earlier such state u is inconsistent, and either $g(u) > v(u)$ or $v(u) > \epsilon * g^*(u)$ (or both) according to the definition of Q. Thus, $u \in$ *OPEN*. This shows that $Q \subseteq$ *OPEN*. Consequently, if any state s has $c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in$ *OPEN*, it is also true that $c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in Q$, and the statement of the lemma holds from Lemma 11. This proves that the lemma holds during the next execution of line 7 in *ComputePath*, and proves the whole lemma by induction. ■

Correctness

The following lemma and theorem show how the lemmas in the previous section lead quite trivially to the correctness of AD*.

Lemma 13. *When the* ComputePath *function exits, the following holds for any state s with $(c^*(s, s_{goal}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq \min_{s' \in OPEN}(key(s')))$: the cost of the path from s_{start} to s defined by backpointers is no larger than $\epsilon * g^*(s)$.*

Proof. This result follows directly from Lemma 12 after we combine the statements (i) and (ii) of the lemma together. ■

Theorem 3. *When the ComputePath function exits, the cost of the path from s_{start} to s_{goal} defined by backpointers is no larger than $\epsilon * g^*(s_{goal})$.*

Proof. According to the termination condition of the *ComputePath* function, upon its exit $(v(s_{goal}) \geq g(s_{goal}) \wedge \text{key}(s_{goal}) \leq \min_{s' \in OPEN}(\text{key}(s')))$. The proof then follows from Corollary 13 noting that $c^*(s_{goal}, s_{goal}) = 0$. ■

Efficiency

Several lemmas and theorems in this section provide theoretical guarantees about the efficiency of AD*.

Lemma 14. *Within any particular execution of the ComputePath function once a state is expanded as overconsistent it can never be expanded again (independently of it being overconsistent or underconsistent).*

Proof. Suppose a state s is selected for expansion as overconsistent for the first time during the execution of the *ComputePath* function. Then, it is removed from *OPEN* set on

line 8 and inserted into *CLOSED* set on line 10. It can then never be inserted into *OPEN* set again unless the *ComputePath* function exits since any state that is about to be inserted into *OPEN* set is checked against membership in *CLOSED* on line 2. Because only the states from *OPEN* set are selected for expansion, s can therefore never be expanded second time. ■

Lemma 15. *No state is expanded more than once as underconsistent within any particular execution of the ComputePath function.*

Proof. Once a state is expanded as underconsistent its v-value is set to ∞. As a result, unless the state is expanded as overconsistent this state can never become underconsistent again. This is so because for a state to be underconsistent it needs to have its v-value strictly less than its g-value, which implies that the v-value needs to be finite. The only way for a v-value to change its value onto a finite value, on the other hand, is during the expansion of the state as an overconsistent state. However, if the state is expanded as overconsistent then according to Theorem 14 the state is never expanded again. Thus, a state can be expanded at most once as underconsistent. ■

Theorem 4. *No state is expanded more than twice during any particular execution of the ComputePath function. A state can be expanded at most once as underconsistent and at most once as overconsistent.*

Proof. According to Lemmas 14 and 15 each state can be expanded at most once as underconsistent and at most once as overconsistent. Since there are no other ways to expand states, this leads to the desired result: each state is expanded at most twice. ■

Theorem 5. *A state s is expanded by the ComputePath function only if either it was inconsistent before ComputePath is called or its v-value was altered by ComputePath at some point during its current execution.*

Proof. Let us pick a state s such that right before a call to the *ComputePath* function it was consistent and during the execution of *ComputePath* its v-value has never been altered. Then it means that $v_{afterComputePath}(s) = v_{beforeComputePath}(s) = g_{beforeComputePath}(s)$. Since only states from *OPEN* are selected for expansion and *OPEN* contains only inconsistent states, then in order for s to have been selected for expansion, it should have had $v(s) \neq g(s)$. Because the v-value of s remains the same throughout the *ComputePath* function execution, it has to be the case that the g-value of s has changed since the beginning of *ComputePath*. If s is expanded as overconsistent then $v(s)$ is changed by setting it to $g(s)$, whereas if s is expanded as underconsistent then $v(s)$ is increased by setting it to ∞ (it could not have been equal to ∞ before since it was underconsistent, i.e., $v(s) < g(s) \leq \infty$). Both cases contradict to our assumption that $v(s)$ remained the same throughout the execution of *ComputePath*. ■

```
ReinitializeRRT(rrt T)
 1 T.cleartree();
 2 T.add(q_start);
GrowRRT(rrt T)
 3 q_new = q_start; time = 0;
 4 while (Distance(q_new, q_goal) > distance-threshold)
 5   q_target = ChooseTarget(T);
 6   if (q_target ≠ null)
 7     q_new = ExtendToTarget(q_target, T);
 8     if (q_new ≠ null)
 9       T.add(q_new);
10   UpdateTime(time);
11   if (time > max-time-per-rrt)
12     return null;
13 return T.c(q_start, q_new);
ChooseTarget(rrt T)
14 p = RandomReal([0.0, 1.0]);
15 if (p < goal-sampling-prob)
16   return q_goal;
17 else
18   q_new = RandomConfiguration();
19   attempts = 0;
20   while (h(q_start, q_new) + h(q_new, q_goal) > T.C_s)
21     q_new = RandomConfiguration();
22     attempts = attempts + 1;
23     if (attempts > max-sample-attempts)
24       return null;
25   return q_new;
```

Figure A.7: (Reproduction) The Anytime RRT Algorithm: GrowRRT and ChooseTarget functions

A.4 The Anytime RRT Algorithm

Because the Anytime RRT algorithm uses a solution bound to influence and restrict the growth of each successive RRT, we can prove a couple central properties regarding the quality of the successive solutions produced by the algorithm. These proofs originally appeared in [Ferguson and Stentz, 2006b].

Theorem 6. *When the solution bound is C_s, the cost of any solution generated by the Anytime RRT algorithm will be less than or equal to C_s.*

Proof. The solution bound C_s is used to influence both the sampling process in the function *ChooseTarget* and the extension process in the function *ExtendToTarget*. However,

NodeSelectionCost(rrt T, configuration q, configuration q_{target})

1 return $T.d_b \cdot$ **Distance**$(q, q_{target}) + T.c_b \cdot T.c(q_{start}, q)$;

ExtendToTarget(rrt T, configuration q_{target})

2 $Q_{near} =$ **kNearestNeighbors**(q_{target}, k, T);
3 while Q_{near} is not empty
4 remove configuration q_{tree} with minimum **NodeSelectionCost**$(T, q_{tree}, q_{target})$ from Q_{near};
5 $Q_{ext} =$ **GenerateExtensions**(q_{tree}, q_{target});
6 $q_{new} = \text{argmin}_{q \in Q_{ext}} c(q_{tree}, q)$;
7 $T.c(q_{start}, q_{new}) = T.c(q_{start}, q_{tree}) + c(q_{tree}, q_{new})$;
8 if $(T.c(q_{start}, q_{new}) + h(q_{new}, q_{goal}) \leq T.\mathcal{C}_s)$
9 return q_{new};
10 return **null**;

Main()

11 $T.d_b = 1$; $T.c_b = 0$; $T.\mathcal{C}_s = \infty$;
12 forever
13 **ReinitializeRRT**(T);
14 $T.\mathcal{C}_n =$ **GrowRRT**(T);
15 if $(T.\mathcal{C}_n \neq$ **null**$)$
16 **PostCurrentSolution**(T);
17 $T.\mathcal{C}_s = (1 - \epsilon_f) \cdot T.\mathcal{C}_n$;
18 $T.d_b = T.d_b - \delta_d$;
19 if $(T.d_b < 0)$
20 $T.d_b = 0$;
21 $T.c_b = T.c_b + \delta_c$;
22 if $(T.c_b > 1)$
23 $T.c_b = 1$;

Figure A.8: (Reproduction) The Anytime RRT Algorithm: ExtendToTarget and Main functions

it is its use in *ExtendToTarget* that ensures that the solution generated will have a cost of at most $\mathcal{C}_s$.

A solution is generated by the Anytime RRT algorithm when the termination condition in function *GrowRRT* is satisfied (Figure A.7 line 4)[8], that is, when a node has been added to the tree that is within some specified distance of the goal configuration. The cost of the resulting solution is the cost of the associated node, which is the sum of all the edges in the tree along the path from the initial configuration to this node. This cost is set for each node in the *ExtendToTarget* function, before the node is originally added to the tree (Figure A.8 line 7). After this cost is set, it is then checked to ensure it is not greater than

[8]This function can also fail and return no solution (line 11), but in this case no solution is generated and so the current theorem does not apply.

the current solution bound C_s (line 8). If the cost is not greater, then the node is added to the tree (line 9). Otherwise, the node is abandoned and a new one is considered. As this is the only place in the algorithm where nodes are added to the tree, this check ensures that any nodes in the tree have costs that are at least as low as the solution bound C_s. Thus, when a node within the specified distance of the goal configuration is added to the tree and *GrowRRT* terminates successfully, the cost of this node, and hence the resulting solution, is guaranteed to be at least as low as the solution bound C_s. ■

The above property can be used to force the algorithm to improve the quality of each successive solution by an improvement factor of ϵ_f.

Theorem 7. *Each time a solution is posted by the Anytime RRT algorithm, the cost of this solution will be at most* $(1 - \epsilon_f)$ *times the cost of the previous solution posted.*

Proof. The first time a solution is generated this property trivially holds, since there is no valid previous solution so the cost of the previous solution is unbounded. After this first solution is generated, and each subsequent time a new solution is generated by the *GrowRRT* function, the solution bound C_s is updated to be $(1 - \epsilon_f)$ times the cost of the solution just generated (Figure A.8 line 17). Thus, by the previous theorem, the next time a solution is generated it will have a cost that is at most the value of this new solution bound, i.e. $(1 - \epsilon_f)$ times the previous solution cost. ■

Together, these properties ensure that solutions produced by the Anytime RRT algorithm improve over time and that the rate of improvement is at least as good as the solution improvement factor ϵ_f.

Corollary 2. *If* $\epsilon_f > 0$ *then the solutions posted by the Anytime RRT algorithm will have associated costs that are strictly decreasing. Moreover, these costs will decrease by at least a factor of* ϵ_f *between each successive solution.*

Proof. This follows directly from the previous theorem: each solution posted is guaranteed to be at most $(1 - \epsilon_f)$ times as expensive as the previous solution, so by setting $\epsilon_f > 0$ we ensure that the costs of the solutions are strictly decreasing, and by at least a factor of ϵ_f between each successive solution. ■

References

[Amato et al., 1998] Amato, N., Bayazit, O., Dale, L., Jones, C., and Vallejo, D. (1998). OBPRM: An obstacle-based PRM for 3D workspaces. In *Proceedings of the Workshop on the Algorithmic Foundations of Robotics.*

[Amato and Wu, 1996] Amato, N. and Wu, Y. (1996). A randomized roadmap method for path and manipulation planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Ambite and Knoblock, 1997] Ambite, J. and Knoblock, C. (1997). Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

[Arkin, 1989] Arkin, R. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112.

[Astrom, 1965] Astrom, K. (1965). Optimal control of partially observable Markovian systems. *Journal of the Franklin Institute*, 280(5):367–386.

[Azarm and Schmidt, 1996] Azarm, K. and Schmidt, G. (1996). A decentralized approach for the conflict free motion of multiple mobile robots. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Balch and Hybinette, 2000] Balch, T. and Hybinette, M. (2000). Social potentials for scalable multirobot formations. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Bar-Noy and Schieber, 1991] Bar-Noy, A. and Schieber, B. (1991). The Canadian Traveller Problem. In *Proceedings of the second annual ACM-SIAM Symposium on Discrete Algorithms*, pages 261 – 270.

[Barbehenn and Hutchinson, 1995] Barbehenn, M. and Hutchinson, S. (1995). Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest path trees. *IEEE Transactions on Robotics and Automation*, 11(2):198–214.

[Barraquand et al., 1992] Barraquand, J., Langlois, B., and Latombe, J. (1992). Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(2):224–241.

[Bekris et al., 2003] Bekris, K., Chen, B., Ladd, A., Plakue, E., and Kavraki, L. (2003). Multiple query probabilistic roadmap planning using single query primitives. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming.* Princeton University Press.

[Bennewitz et al., 2001] Bennewitz, M., Burgard, W., and Thrun, S. (2001). Optimizing schedules for prioritized path planning of multi-robot systems. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Bererton and Gordon, 2004] Bererton, C. and Gordon, G. (2004). Multi-robot coordination in the presence of an adversary. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS).*

[Bererton et al., 2003] Bererton, C., Gordon, G., Thrun, S., and Khosla, P. (2003). Auction mechanism design for multi-robot coordination. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS).*

[Bertsekas, 1987] Bertsekas, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ.

[Blum and Furst, 1995] Blum, A. and Furst, M. (1995). Fast planning through graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636 – 1642, Montreal.

[Bohlin and Kavraki, 2000] Bohlin, R. and Kavraki, L. (2000). Path planning using lazy PRM. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33.

[Boor et al., 1999] Boor, V., Overmars, M., and van der Stappen, A. (1999). The Gaussian sampling strategy for probabilistic roadmap planners. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Borenstein and Koren, 1991] Borenstein, J. and Koren, Y. (1991). The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288.

[Boutilier et al., 1998] Boutilier, C., Brafman, R., and Geib, C. (1998). Structured reachability analysis for MDPs. In *Uncertainty in Artificial Intelligence.*

[Bowling et al., 2004] Bowling, M., Browning, B., and Veloso, M. (2004). Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).*

[Branicky and Curtiss, 2002] Branicky, M. and Curtiss, M. (2002). Nonlinear and hybrid control via RRTs. In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems.*

[Brock and Khatib, 1999] Brock, O. and Khatib, O. (1999). High-speed navigation using the global dynamic window approach. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Brown and Jennings, 1995] Brown, R. and Jennings, J. (1995). A pusher/steerer model for strongly cooperative mobile robot manipulation. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Bruce and Veloso, 2002] Bruce, J. and Veloso, M. (2002). Real-time randomized path planning for robot navigation. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Brumitt and Stentz, 1996] Brumitt, B. and Stentz, A. (1996). Dynamic mission planning for multiple mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Burgard et al., 2000] Burgard, W., Moors, M., Fox, D., Simmons, R., and Thrun, S. (2000). Collaborative multi-robot exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Burns and Brock, 2005] Burns, B. and Brock, O. (2005). Toward optimal configuration space sampling. In *Proceedings of Robotics: Science and Systems (RSS).*

[Carsten et al., 2006] Carsten, J., Ferguson, D., and Stentz, A. (2006). 3D Field D*: Improved Path Planning and Replanning in Three Dimensions. Submitted to IROS.

[Chaimowicz et al., 2004] Chaimowicz, L., Kumar, V., and Campos, M. (2004). A paradigm for dynamic coordination of multiple robots. *Autonomous Robots*, 17(1):7–21.

[Chaimowicz et al., 2001] Chaimowicz, L., Sugar, T., Kumar, V., and Campos, M. (2001). An architecture for tightly coupled multi-robot cooperation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Chakrabarti et al., 1988] Chakrabarti, P., Ghosh, S., and DeSarkar, S. (1988). Admissibility of AO* when heuristics overestimate. *Artificial Intelligence*, 34:97–113.

[Chang and Slagle, 1971] Chang, C. and Slagle, J. (1971). An admissible and optimal algorithm for searching AND-OR graphs. *Artificial Intelligence*, 2:117 – 128.

[Chen et al., 1995] Chen, D., Szczerba, R., and Uhran, J. (1995). Planning conditional shortest paths through an unknown environment: A framed-quadtree approach. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*.

[Cheng and LaValle, 2002] Cheng, P. and LaValle, S. (2002). Resolution complete Rapidly-exploring Random Trees. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Chien et al., 2000] Chien, S., Knight, R., Stechert, A., Sherwood, R., and Rabideau, G. (2000). Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the International Conference on Artificial Intelligence, Planning and Scheduling (AIPS)*.

[Choset and Burdick, 1995] Choset, H. and Burdick, J. (1995). Sensor based planning, part I: The generalized voronoi graph. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Choudhury and Lynch, 2002] Choudhury, P. and Lynch, K. (2002). Trajectory planning for second-order underactuated mechanical systems in presence of obstacles. In *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*.

[Clark et al., 2003] Clark, C., Rock, S., and Latombe, J. (2003). Motion planning for multirobot systems using dynamic robot networks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Dahlkamp et al., 2006] Dahlkamp, H., Kaehler, A., Stavens, D., Thrun, S., and Bradski, G. (2006). Self-supervised monocular road detection in desert terrain. In *Proceedings of Robotics: Science and Systems (RSS)*.

[Dasgupta et al., 1994] Dasgupta, P., Chakrabarti, P., and DeSarkar, S. (1994). Agent searching in a tree and the optimality of iterative deepening. *Artificial Intelligence*, 71:195–208.

[de Berg et al., 2000] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000). Visibility graphs: Finding the shortest route. In *Computational Geometry: Algorithms and Applications, Second Edition*, pages 307–317. Springer-Verlag.

[Dean and Boddy, 1988] Dean, T. and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

[Dias et al., 2004a] Dias, M., Zinck, M., Zlot, R., , and Stentz, A. (2004a). Robust multi-robot coordination in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Dias et al., 2004b] Dias, M., Zlot, R., Zinck, M., Gonzalez, J., and Stentz, A. (2004b). A versatile implementation of the traderbots approach for multirobot coordination. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS).*

[Dijkstra, 1959] Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

[Edelkamp, 2001] Edelkamp, S. (2001). Planning with pattern databases. In *Proceedings of the European Conference on Planning.*

[Ersson and Hu, 2001] Ersson, T. and Hu, X. (2001). Path planning and navigation of mobile robots in unknown environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Ferguson et al., 2006] Ferguson, D., Kalra, N., and Stentz, A. (2006). Replanning with RRTs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Ferguson et al., 2005] Ferguson, D., Likhachev, M., and Stentz, A. (2005). A guide to heuristic-based path planning. In *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling.*

[Ferguson and Stentz, 2004a] Ferguson, D. and Stentz, A. (2004a). Delayed D*: The Proofs. Technical Report CMU-RI-TR-04-51, Carnegie Mellon Robotics Institute.

[Ferguson and Stentz, 2004b] Ferguson, D. and Stentz, A. (2004b). Planning with Imperfect Information. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Ferguson and Stentz, 2004c] Ferguson, D. and Stentz, A. (2004c). Robust Path Planning with Imperfect Maps. In *Proceedings of the Army Science Conference.*

[Ferguson and Stentz, 2005a] Ferguson, D. and Stentz, A. (2005a). Field D*: An Interpolation-based Path Planner and Replanner. In *Proceedings of the International Symposium on Robotics Research (ISRR).*

[Ferguson and Stentz, 2005b] Ferguson, D. and Stentz, A. (2005b). The Delayed D* Algorithm for Efficient Path Replanning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Ferguson and Stentz, 2006a] Ferguson, D. and Stentz, A. (2006a). Anytime RRTs. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*.

[Ferguson and Stentz, 2006b] Ferguson, D. and Stentz, A. (2006b). Anytime RRTs: The Proofs. Technical Report CMU-RI-TR-06-07, Carnegie Mellon School of Computer Science.

[Ferguson and Stentz, 2006c] Ferguson, D. and Stentz, A. (2006c). Multi-resolution Field D*. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*.

[Ferguson and Stentz, 2006d] Ferguson, D. and Stentz, A. (2006d). Using Interpolation to Improve Path Planning: The Field D* Algorithm. *Journal of Field Robotics*, 23(2):79–101.

[Ferguson et al., 2004] Ferguson, D., Stentz, A., and Thrun, S. (2004). PAO* for Planning with Hidden State. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, New Orleans, LA.

[Ferrer, 2002] Ferrer, G. (2002). *Anytime Replanning Using Local Subplan Replacement.* PhD thesis, University of Virginia.

[Fiorini and Shiller, 1996] Fiorini, P. and Shiller, Z. (1996). Time optimal trajectory planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Fraichard, 1999] Fraichard, T. (1999). Trajectory planning in a dynamic workspace: a 'state-time' approach. *Advanced Robotics*, 13(1):75–94.

[Frazzoli et al., 2002] Frazzoli, E., Dahleh, M., and Feron, E. (2002). Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance and Control*, 25(1):116–129.

[Fredslund and Matarić, 2001] Fredslund, J. and Matarić, M. (2001). Robot formations using only local sensing and control. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*.

[Fujimura, 1991] Fujimura, K. (1991). *Motion Planning in Dynamic Environments.* Springer-Verlag, Tokyo.

[Fujimura, 1995] Fujimura, K. (1995). Time-minimum routes in time-dependent networks. *IEEE Transactions on Robotics and Automation*, 11(3):343–351.

[Gelperin, 1977] Gelperin, D. (1977). On the optimality of A*. *Artificial Intelligence*, 8(1):69–76.

[Gerkey and Matarić, 2002a] Gerkey, B. and Matarić, M. (2002a). Pusher-watcher: An approach to fault-tolerant tightly-coupled robot coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Gerkey and Matarić, 2002b] Gerkey, B. and Matarić, M. (2002b). Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768.

[Gerkey et al., 2005] Gerkey, B., Thrun, S., , and Gordon, G. (2005). Parallel stochastic hill-climbing with small teams. In *Multi-Robot Systems*. Kluwer.

[Gonzalez and Stentz, 2005] Gonzalez, J. and Stentz, A. (2005). Planning with uncertainty in position: An optimal and efficient planner. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*.

[Hara et al., 1996] Hara, M., Fukuda, M., Nishibayashi, H., Aiyama, Y., Ota, J., and Arai, T. (1996). Motion control of cooperative transportation system by quadruped robots based on vibration model in walking. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*.

[Hart et al., 1968] Hart, P., Nilsson, N., and Rafael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107.

[Hawes, 2002] Hawes, N. (2002). An anytime planning agent for computer game worlds. In *Proceedings of the Workshop on Agents in Computer Games, 3rd International Conference on Computers and Games (CG)*.

[Hebert et al., 1999] Hebert, M., McLachlan, R., and Chang, P. (1999). Experiments with driving modes for urban robots. In *Proceedings of SPIE Mobile Robots*.

[Holleman and Kavraki, 2000] Holleman, C. and Kavraki, L. (2000). A framework for using the workspace medial axis in PRM planners. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Horvitz, 1987] Horvitz, E. (1987). Problem-solving design: Reasoning about computational value, trade-offs, and resources. In *Proceedings of the Second Annual NASA Research Forum*.

[Howard and Kelly, 2005] Howard, T. and Kelly, A. (2005). Trajectory generation on rough terrain considering actuator dynamics. In *Proceedings of the International Conference on Advanced Robotics (FSR).*

[Howard and Kelly, 2006] Howard, T. and Kelly, A. (2006). Constrained optimization path following over rough terrain. In *Proceedings of the International Symposium on Experimental Robotics (ISER).*

[Hsu et al., 2003] Hsu, D., Jiang, T., Reif, J., and Sun, Z. (2003). The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Hsu et al., 2002] Hsu, D., Kindel, R., Latombe, J., and Rock, S. (2002). Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255.

[Hsu et al., 1999] Hsu, D., Latombe, J., and Motwani, R. (1999). Path planning in expansive configuration spaces. *International Journal of Computational Geometry and Applicaitons*, 4:495–512.

[Huiming et al., 2001] Huiming, Y., Chia-Jung, C., Tong, S., and Qiang, B. (2001). Hybrid evolutionary motion planning using follow boundary repair for mobile robots. *Journal of Systems Architecture*, 47:635–647.

[Huntsberger et al., 2003] Huntsberger, T., Pirjanian, P., Trebi-Ollennu, A., Nayar, H., Aghazarian, H., Ganino, A., and Garrett, M. (2003). Campout: a control architecture for tightly coupled coordination of multirobot systems for planetary surface exploration. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 33(5).

[Jaillet and Simeon, 2004] Jaillet, L. and Simeon, T. (2004). A PRM-based motion planner for dynamically changing environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Jaillet et al., 2005] Jaillet, L., Yershova, A., LaValle, S., and Simeon, T. (2005). Adaptive tuning of the sampling domain for dynamic-domain RRTs. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Jennings et al., 1997] Jennings, J., Whelan, G., and Evans, W. (1997). Cooperative search and rescue with a team of mobile robots. In *Proceedings of the International Conference on Advanced Robotics (ICAR).*

[Kaelbling et al., 1998] Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence.*

[Kagami et al., 2003] Kagami, S., Kuffner, J., Nishiwaki, K., Okada, K., and Inaba, M. (2003). Humanoid arm motion planning using stereo vision and RRT search. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Kallmann et al., 2003] Kallmann, M., Aubel, A., Abaci, T., and Thalmann, D. (2003). Planning collision-free reaching motions for interactive object manipulation and grasping. *Eurographics*, 22(3).

[Kallmann and Matarić, 2004] Kallmann, M. and Matarić, M. (2004). Motion Planning using Dynamic Roadmaps. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kalra et al., 2005] Kalra, N., Ferguson, D., and Stentz, A. (2005). Hoplites: A market-based framework for planned tight coordination in multirobot teams. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kalra et al., 2006] Kalra, N., Ferguson, D., and Stentz, A. (2006). Constrained Exploration for Studies in Multirobot Coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kambhampati and Davis, 1986] Kambhampati, S. and Davis, L. (1986). Multi-resolution path planning for mobile robots. *IEEE Journal of Robotics and Automation*, RA-2(3).

[Kamon et al., 1996] Kamon, I., Rivlin, E., and Rimon, E. (1996). A new range-sensor based globally convergent navigation algorithm for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kant and Zucker, 1986] Kant, K. and Zucker, S. (1986). Toward efficient planning: the path-velocity decomposition. *International Journal of Robotics Research*, 5(3):72–89.

[Kavraki et al., 1996] Kavraki, L., Svestka, P., Latombe, J., and Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580.

[Kelly, 1995] Kelly, A. (1995). *An Intelligent Predictive Control Approach to the High Speed Cross Country Autonomous Navigation Problem.* PhD thesis, Carnegie Mellon University.

[Khatib, 1986] Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98.

[Khatib et al., 1996] Khatib, O., Yokoi, K., Chang, K., Ruspini, D., Holmberg, R., Casal, A., and Baader, A. (1996). Force strategies for cooperative tasks in multiple mobile manipulation systems. In *Robotics Research: The Seventh International Symposium*, pages 333–342. Springer.

[Kim and Ostrowski, 2003] Kim, J. and Ostrowski, J. (2003). Motion planning of aerial robots using Rapidly-exploring Random Trees with dynamic constraints. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kim et al., 2001] Kim, J., Vidal, R., Shim, D., Shakernia, O., and Sastry, S. (2001). A hierarchical approach to probabilistic pursuit-evasion games with unmanned ground and aerial vehicles. In *Proceedings of the IEEE Conference on Decision and Control.*

[Kindel et al., 2000] Kindel, R., Hsu, D., Latombe, J., and Rock, S. (2000). Kinodynamic motion planning amidst moving obstacles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kobilarov and Sukhatme, 2004] Kobilarov, M. and Sukhatme, G. (2004). Time Optimal Path Planning on Outdoor Terrain for Mobile Robots under Dynamic Constraints. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Koenig, 1997] Koenig, S. (1997). *Goal-Directed Acting with Incomplete Information.* PhD thesis, Carnegie Mellon University.

[Koenig and Likhachev, 2002] Koenig, S. and Likhachev, M. (2002). Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Koenig et al., 2004] Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong Planning A*. *Artificial Intelligence*, 155(1-2):93–146.

[Koes et al., 2005] Koes, M., Nourbakhsh, I., and Sycara, K. (2005). Heterogeneous multi-robot coordination with spatial and temporal constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

[Koes et al., 2006] Koes, M., Nourbakhsh, I., and Sycara, K. (2006). Constraint optimization architecture for search and rescue robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Konolige, 2000] Konolige, K. (2000). A gradient method for realtime robot control. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Korf, 1993] Korf, R. (1993). Linear-space best-first search. *Artificial Intelligence*, 62:41–78.

[Kosuge et al., 1999] Kosuge, K., Hirata, Y., Asama, H., Kaetsu, H., and Kawabata, K. (1999). Motion control of multiple autonomous mobile robots handling a large object

in coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kuffner and LaValle, 2000] Kuffner, J. and LaValle, S. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Kuffner et al., 2003] Kuffner, J., Nishiwaki, K., Kagami, S., Inaba, M., and Inoue, H. (2003). Motion planning for humanoid robots. In *Proceedings of the International Symposium on Robotics Research (ISRR).*

[Larson, 1967] Larson, R. (1967). A survey of dynamic programming computational procedures. *IEEE Transactions on Automatic Control*, pages 767–774.

[Larson and Casti, 1982] Larson, R. and Casti, J. (1982). *Principles of Dynamic Programming, Part 2.* Marcel Dekker, New York.

[Latombe, 1991] Latombe, J. (1991). *Robot Motion Planning.* Kluwer Academic Publishers, Boston, MA.

[LaValle, 1998] LaValle, S. (1998). Rapidly-exploring Random Trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa state University.

[LaValle, 2006] LaValle, S. (2006). *Planning Algorithms.* Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/).

[LaValle and Kuffner, 1999] LaValle, S. and Kuffner, J. (1999). Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[LaValle and Kuffner, 2001] LaValle, S. and Kuffner, J. (2001). Rapidly-exploring Random Trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions*, pages 293–308.

[Leven and Hutchinson, 2002] Leven, P. and Hutchinson, S. (2002). Real-time motion planning in changing environments. In *Proceedings of the International Symposium on Robotics Research (ISRR).*

[Li and Shie, 2002] Li, T. and Shie, Y. (2002). An Incremental Learning Approach to Motion Planning with Roadmap Management. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Likhachev, 2003] Likhachev, M. (2003). Search techniques for planning in large dynamic deterministic and stochastic environments. Thesis proposal. School of Computer Science, Carnegie Mellon University.

[Likhachev, 2005] Likhachev, M. (2005). *Search-based Planning for Large Dynamic Environments.* PhD thesis, Carnegie Mellon University.

[Likhachev et al., 2005a] Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S. (2005a). Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).*

[Likhachev et al., 2005b] Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S. (2005b). Anytime Dynamic A*: The Proofs. Technical Report CMU-RI-TR-05-12, Carnegie Mellon School of Computer Science.

[Likhachev et al., 2003] Likhachev, M., Gordon, G., and Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems.* MIT Press.

[Likhachev and Koenig, 2002] Likhachev, M. and Koenig, S. (2002). Speeding up the Parti-Game Algorithm. In *Advances in Neural Information Processing Systems.* MIT Press.

[Likhachev and Stentz, 2006] Likhachev, M. and Stentz, A. (2006). PPCP: Efficient probabilistic planning with clear preferences in partially-known environments. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

[Lin and Hsu, 1995] Lin, F. and Hsu, J. (1995). Cooperation and deadlock-handling for an object-sorting task in a multi-agent robotic system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Lumelsky and Stepanov, 1987] Lumelsky, V. and Stepanov, A. (1987). Path planning strategies for point mobile automation moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430.

[Matthies et al., 2000] Matthies, L., Xiong, Y., Hogg, R., Zhu, D., Rankin, A., Kennedy, B., Hebert, M., Maclachlan, R., Won, C., Frost, T., Sukhatme, G., McHenry, M., and Goldberg, S. (2000). A portable, autonomous, urban reconnaissance robot. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS).*

[Mazer et al., 1992] Mazer, E., Ahuactzin, J., and bessiere, P. (1992). The Ariadne's Clew algorithm. In *Proceedings of the International Conference of The Society of Adaptive Behavior.*

[Mazer et al., 1998] Mazer, E., Ahuactzin, J., and Bessiere, P. (1998). The Ariadne's Clew algorithm. *Journal of Artificial Intelligence Research*, 9:295–316.

[Mitchell, 2000] Mitchell, J. (2000). *Handbook of Computational Geometry*, chapter Geometric Shortest Paths and Network Optimization, pages 633–701. Elsevier Science.

[Mitchell and Papadimitriou, 1991] Mitchell, J. and Papadimitriou, C. (1991). The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38:18–73.

[Monahan, 1982] Monahan, G. (1982). A survey of partially observable Markov decision processes: theory, models and algorithms. *Management Science*, 28(1).

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

[Nabbe and Hebert, 2004] Nabbe, B. and Hebert, M. (2004). Path planning with hallucinated worlds. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*.

[Naffin and Sukhatme, 2004] Naffin, D. and Sukhatme, G. (2004). Negotiated formations. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*.

[Nagy and Kelly, 2001] Nagy, B. and Kelly, A. (2001). Trajectory generation for car-like robots using cubic curvature polynomials. In *Proceedings of the International Conference on Advanced Robotics (FSR)*.

[Nguyen et al., 2003] Nguyen, H., Pezeshkian, N., Raymond, M., Gupta, A., and Spector, J. (2003). Autonomous communication relays for tactical robots. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*.

[Nieuwenhuisen and Overmars, 2004] Nieuwenhuisen, D. and Overmars, M. (2004). Useful cycles in probabilistic roadmap graphs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Nilsson, 1980] Nilsson, N. (1980). *Principles of Artificial Intelligence*. Tioga Publishing Company.

[Nourbakhsh and Genesereth, 1996] Nourbakhsh, I. and Genesereth, M. (1996). Assumptive Planning and Execution: a Simple, Working Robot Architecture. *Autonomous Robots Journal*, 3(1):49–67.

[O'Donnell and Lozano-Pérez, 1989] O'Donnell, P. and Lozano-Pérez, T. (1989). Deadlock-free and collision-free coordination of two robot manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Oriolo et al., 2004] Oriolo, G., Vendittelli, M., Freda, L., and Troso, G. (2004). The SRT Method: Randomized strategies for exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Østergaard et al., 2001] Østergaard, E., Matarić, M., and Sukhatme, G. (2001). Distributed multi-robot task allocation for emergency handling. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Osumi et al., 1998] Osumi, H., Terasawa, M., and Nojiri, H. (1998). Cooperative control of multiple mobile manipulators on uneven ground. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Ota et al., 1995] Ota, J., Miyata, N., Arai, T., Yoshida, E., Kurabayashi, D., and Sasaki, J. (1995). Transferring and regrasping a large object by cooperation of multiple mobile robots. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Pai and Reissell, 1998] Pai, D. and Reissell, L. (1998). Multiresolution rough terrain motion planning. *IEEE Transactions on Robotics and Automation*, 14(1):19–33.

[Pan and Luo, 1991] Pan, T. and Luo, R. (1991). Motion planning for mobile robots in a dynamic environment with moving obstacles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Papadimitriou and Tsitsiklis, 1987] Papadimitriou, C. and Tsitsiklis, J. (1987). The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450.

[Parker, 1993] Parker, L. (1993). Designing control laws for cooperative agent teams. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

[Petty and Fraichard, 2005] Petty, S. and Fraichard, T. (2005). Safe motion planning in dynamic environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Philippsen, 2004] Philippsen, R. (2004). *Motion Planning and Obstacle Avoidance for Mobile Robots in Highly Cluttered Dynamic Environments.* PhD thesis, EPFL, Lausanne, Switzerland.

[Philippsen and Siegwart, 2005] Philippsen, R. and Siegwart, R. (2005). An Interpolated Dynamic Navigation Function. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Pineau et al., 2003a] Pineau, J., Gordon, G., and Thrun, S. (2003a). Point-Based Value Iteration: An anytime algorithm for POMDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).*

[Pineau et al., 2003b] Pineau, J., Gordon, G., and Thrun, S. (2003b). Policy-contingent abstraction for robust robot control. In *Proceedings of Uncertainty in Artificial Intelligence (UAI).*

[Pirjanian et al., 2002] Pirjanian, P., Leger, C., Mum, E., Kennedy, B., Garrett, M., Aghazarian, H., Farritor, S., and Schenker, P. (2002). Distributed control for a modular, reconfigurable cliff robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Pisula et al., 2000] Pisula, C., Hoff, K., Lin, M., and Manoch, D. (2000). Randomized path planning for a rigid body based on hardware acceleraated Voronoi sampling. In *Proceedings of the Workshop on the Algorithmic Foundations of Robotics.*

[Podsedkowski et al., 2001] Podsedkowski, L., Nowakowski, J., Idzikowski, M., and Vizvary, I. (2001). A new solution for path planning in partially known or unknown environments for nonholonomic mobile robots. *Robotics and Autonomous Systems*, 34:145–152.

[Prendinger and Ishizuka, 1998] Prendinger, H. and Ishizuka, M. (1998). APS, a prolog-based anytime planning system. In *Proceedings of the 11th International Conference on Applications of Prolog (INAP.*

[Rabin, 2000] Rabin, S. (2000). A* speed optimizations. In DeLoura, M., editor, *Game Programming Gems*, pages 272–287, Rockland, MA. Charles River Media.

[Ramalingam and Reps, 1996] Ramalingam, G. and Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305.

[Reif and Sharir, 1985] Reif, J. and Sharir, M. (1985). Motion planning in the presence of moving obstacles. In *Proceedings of the IEEE Conference on the Foundations of Computer Science.*

[Rich and Knight, 1992] Rich, E. and Knight, K. (1992). *Artificial Intelligence.* McGraw-Hill.

[Rowe and Richbourg, 1990] Rowe, N. and Richbourg, R. (1990). An efficient Snell's-law method for optimal-path planning across two-dimensional irregular homogeneous-cost regions. *International Journal of Robotics Research*, 9(6):48–66.

[Roy et al., 2005] Roy, N., Gordon, G., and Thrun, S. (2005). Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research*, 23:1–40.

[Roy and Thrun, 1999] Roy, N. and Thrun, S. (1999). Coastal navigation with mobile robots. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*.

[Samet, 1982] Samet, H. (1982). Neighbor Finding Techniques for Images Represented by Quadtrees. *Computer Graphics and Image Processing*, 18:37–57.

[Sanchez and Latombe, 2001] Sanchez, G. and Latombe, J. (2001). A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proceedings of the International Symposium on Robotics Research (ISRR)*.

[Schouwenaars et al., 2006] Schouwenaars, T., Stubbs, A., Paduano, J., and Feron, E. (2006). Multivehicle path planning for nonline-of-sight communication. *Journal of Field Robotics*, 23(3-4):269–290.

[Sethian, 1996] Sethian, J. (1996). A fast marching level set method for monotonically advancing fronts. *Applied Mathematics, Proceedings of the National Academy of Science*, 93:1591–1595.

[Shih et al., 1990] Shih, C., Lee, T., and Gruver, W. (1990). A unified approach for robot motion planning among moving polyhedral obstacles. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(4):903–915.

[Simeon et al., 2000] Simeon, T., Laumond, J., and Nissoux, C. (2000). Visibility based probabilistic roadmaps for motion planning. *Advanced Robotics Journal*, 14(6).

[Simmons et al., 2000] Simmons, R., Singh, S., Hershberger, D., Ramos, J., and Smith, T. (2000). First results in the coordination of heterogeneous robots for large-scale assembly. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*.

[Singh et al., 2000] Singh, S., Simmons, R., Smith, T., Stentz, A., Verma, V., Yahja, A., and Schwehr, K. (2000). Recent progress in local and global traversability for planetary rovers. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

[Smith, 1980] Smith, R. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12).

[Smith and Simmons, 2004] Smith, T. and Simmons, R. (2004). Heuristic search value iteration for POMDPs. In *Proceedings of Uncertainty in Artificial Intelligence (UAI).*

[Sofman et al., 2006] Sofman, B., Lin, E., Bagnell, J., Vandapel, N., and Stentz, A. (2006). Improving robot navigation through self-supervised online learning. In *Proceedings of Robotics: Science and Systems (RSS).*

[Sondik, 1971] Sondik, E. (1971). *The optimal control of partially observable Markov decision processes.* PhD thesis, Stanford University.

[Spaan and Vlassis, 2005] Spaan, M. and Vlassis, N. (2005). Perseus: Randomized Point-based Value Iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220.

[Stachniss and Burgard, 2002] Stachniss, C. and Burgard, W. (2002). An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Stentz, 1994] Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Stentz, 1995] Stentz, A. (1995). The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).*

[Stentz and Dias, 1999] Stentz, A. and Dias, M. (1999). A free market architecture for coordinating multiple robots. Technical Report CMU-RI-TR-99-42, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

[Stentz and Hebert, 1995] Stentz, A. and Hebert, M. (1995). A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145.

[Strandberg, 2004a] Strandberg, M. (2004a). Augmenting RRT-planners with local trees. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Strandberg, 2004b] Strandberg, M. (2004b). *Robot Path Planning: An Object-Oriented Approach.* PhD thesis, Royal Institute of Tehcnology (KTH).

[Stroupe, 2003] Stroupe, A. (2003). *Collaborative Execution of Exploration and Tracking Using Move Value Estimation for Robot Teams (MVERT).* PhD thesis, Carnegie Mellon University.

[Thayer et al., 2000] Thayer, S., Digney, B., Diaz, M., Stentz, A., Nabbe, B., and Hebert, M. (2000). Distributed robotic mapping of extreme environments. In *Proceedings of SPIE Mobile Robots.*

[Tompkins et al., 2004] Tompkins, P., Stentz, A., and Whittaker, W. (2004). Mission-Level Path Planning for Rover Exploration. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS).*

[Trovato, 1990] Trovato, K. (1990). Differential A*: an adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment. *Journal of Pattern Recognition and Artificial Intelligence*, 4(2).

[Tsitsiklis, 1995] Tsitsiklis, J. (1995). Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9).

[Urmson and Simmons, 2003] Urmson, C. and Simmons, R. (2003). Approaches for heuristically biasing RRT growth. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Vail and Veloso, 2003] Vail, D. and Veloso, M. (2003). Dynamic multi-robot coordination. In *Multi-Robot Systems.* Kluwer.

[van den Berg et al., 2006] van den Berg, J., Ferguson, D., and Kuffner, J. (2006). Anytime planning and replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[van den Berg and Overmars, 2004] van den Berg, J. and Overmars, M. (2004). Roadmap-based motion planning in dynamic environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Vasquez et al., 2004] Vasquez, D., Large, F., Fraichard, T., and Laugier, C. (2004). High-speed autonomous navigation with motion prediction for unknown moving obstacles. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Wagner and Arkin, 2004] Wagner, A. and Arkin, R. (2004). Multi-robot communication-sensitive reconnaissance. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Wang et al., 1996] Wang, Z., Nakano, E., and Matsukawa, T. (1996). Realizing cooperative object manipulation using multiple behaviour-based robots. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

[Wellington and Stentz, 2003] Wellington, C. and Stentz, A. (2003). Learning predictions of the load-bearing surface for autonomous rough-terrain navigation in vegetation. In *Proceedings of the International Conference on Advanced Robotics (FSR).*

[Wilmarth et al., 1999] Wilmarth, S., Amato, N., and Stiller, P. (1999). MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Yahja et al., 2000] Yahja, A., Singh, S., and Stentz, A. (2000). An efficient on-line path planner for outdoor mobile robots operating in vast environments. *Robotics and Autonomous Systems*, 33:129–143.

[Yahja et al., 1998] Yahja, A., Stentz, A., Singh, S., and Brumitt, B. (1998). Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Yakey et al., 2001] Yakey, J., LaValle, S., and Kavraki, L. (2001). Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6):951–958.

[Yamaguchi, 1997] Yamaguchi, H. (1997). Adaptive formation control for distributed autonomous mobile robot groups. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Yershova et al., 2005] Yershova, A., Jaillet, L., Simeon, T., and LaValle, S. (2005). Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*

[Zhou and Hansen, 2002] Zhou, R. and Hansen, E. (2002). Multiple sequence alignment using A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).* Student abstract.

[Zilberstein and Russell, 1993] Zilberstein, S. and Russell, S. (1993). Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).*

[Zilberstein and Russell, 1995] Zilberstein, S. and Russell, S. (1995). Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation.* Kluwer Academic Publishers.

[Zlot and Stentz, 2003] Zlot, R. and Stentz, A. (2003). Market-based multirobot coordination using task abstraction. In *Proceedings of the International Conference on Advanced Robotics (FSR).*

[Zlot et al., 2002] Zlot, R., Stentz, A., Dias, M., and Thayer, S. (2002). Multi-robot exploration controlled by a market economy. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).*