Continuous Observation Planning

for Autonomous Exploration

by

Bradley R. Hasegawa

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 17, 2004 [September 2004]

Author_____
Department of Electrical Engineering and Computer Science
August 17, 2004

Certified by_____
John J. Leonard
Thesis Supervisor

Certified by_____
Brian C. Williams
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Continuous Observation Planning for Autonomous Exploration
by
Bradley R. Hasegawa

Submitted to the
Department of Electrical Engineering and Computer Science

August 17, 2004

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Many applications of autonomous robots depend on the robot being able to navigate in real world environments. In order to navigate or path plan, the robot often needs to consult a map of its surroundings. A truly autonomous robot must, therefore, be able to drive about its environment and use its sensors to build a map before performing any tasks that require this map. Algorithms that control a robot's motion for the purpose of building a map of an environment are called autonomous exploration algorithms. Because resources such as time and energy are highly constrained in many mobile robot missions, a key requirement of autonomous exploration algorithms is that they cause the robot to explore efficiently. Planning paths to candidate observation points that will lead to efficient exploration is challenging, however, because the set of candidates, and, therefore, the robot's plan, change frequently as the robot adds information to the map. The main claim of this thesis is that, in situations in which the robot discerns the large scale structure of the environment early on during its exploration, the robot can produce paths that cause it to explore efficiently by planning observations to make over a finite horizon. Planning over a finite horizon entails finding a path that visits candidates with the maximum possible total utility, subject to the constraint that the path cost is less than a given threshold value. Finding such a path corresponds to solving the Selective Traveling Salesman Problem (S-TSP) over the set of candidates. In this thesis, we evaluate our claim by implementing full horizon, finite horizon, and greedy approaches to planning observations, and comparing the efficiency of these approaches in both real and simulated environments. In addition, we develop a new approach for solving the S-TSP by framing it as an Optimal Constraint Satisfaction Problem (OCSP).

Thesis Supervisor: John J. Leonard
Title: Associate Professor of Ocean Engineering
Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my advisors, Professor Brian Williams and Professor John Leonard for believing in me and for giving me the opportunity to be a part of their groups. I am especially grateful for all of the time and effort both of my advisors have put into helping me and thinking through problems with me. In my mind, Professor Williams and Professor Leonard have gone above and beyond the call of duty in order to advise me, and I have learned more from my interactions with them than from anything else over this past year.

I am also very grateful to Paul Robertson, Seung Chung, Lars Blackmore, Paul Elliott, John Stedl, Ed Olson, and Jonathan Kennel for their extremely helpful feedback and thoughts on this thesis. In addition, I would like to thank Margaret Yoon, the former administrative assistant to Professor Williams' group, for all of her help in getting this research completed.

I have also received invaluable support and knowledge from I-hsiang Shu, Andrew Patrikalakis, Stanislav Funiak, Oliver Martin, Matt Walter, Mike Bosse, Sung Joon Kim, Raj Krishnan, Tazeen Mahtab, Martin Sachenbacher, Thomas Leaute, Bobby Effinger, Greg Sullivan, Aisha Walcott, Jillian Redfern, Andreas Hoffman, Steve Block, Tsoline Mikaelian, Judy Chen, Hui Li, Mike Benjamin, and all of the other members of Professor Williams and Professor Leonard's research groups. I feel very fortunate to have had not one but two amazing groups of people to work with.

I could not have made it through the year without my friends and family. In particular, I would like to thank T. D. Luckett and Michael Ogrydziak for putting up with me when most other people probably would not have.

Finally, I owe very special thanks to my parents for all of their sacrifices, for all of the opportunities that they have given me, for their support and advice, and for always being there for me. I am incredibly lucky to have gotten the parents that I have.

This research was supported by the NASA Ames Cross Enterprise Technology Development Program (CETDP), contract NAG2-1466ONR, and by the Office of Naval Research, awards N00014-03-1-0879 and N00014-02-C-0210.

1 Introduction

There are a vast number of potential applications for autonomous robots that can move about and act on the physical world. Some of the most important uses of these robots are situations in which it is too dangerous, costly, or technologically difficult to send humans to perform a task. These situations are challenging because they require the robot to operate in the noisy and unpredictable real world, as opposed to carefully controlled factory floors. Recent advances, however, have put robots for many of these applications within reach. Specifically, researchers have worked on rovers for exploring Mars [25], autonomous underwater vehicles (AUVs) for detecting ocean mines [46] and conducting oceanographic surveys [3], unmanned air vehicles (UAVs) for surveillance [27], and urban robots for performing building search and rescue [11] or assisting the growing elderly population [39].

A fundamental problem for all autonomous mobile robots is being able to navigate and reason about the surrounding environment. In many cases, the only way for a robot to solve this problem is to consult an internal model (map) of its environment. For example, a very simple navigation task for an AUV could be to travel forward 50 meters, make a 90 degree turn, and then travel straight for 25 more meters. The AUV could try to traverse paths like this one using dead-reckoning, yet eventually it would find itself unacceptably far off course. The problem is that in all methods of dead-reckoning, including inertial guidance systems and odometry for wheeled robots, the navigation error accumulates over time. Furthermore, in space, underwater, and indoors, GPS is not available to help a robot to navigate. Therefore, the robot must use its sensors to recognize landmarks and localize itself within a map of the environment, in order to bound its navigation error. More sophisticated spatial reasoning, such as planning a path between two points that avoids obstacles, also requires a mobile robot to have a map of the surroundings.

The main problem with a robot relying on maps is that, for many environments, such maps do not exist. Fortunately, researchers have developed algorithms that allow a robot to use its sensors to build a map of its environment and at the same time localize

itself within this map. In other words, these algorithms allow a robot to perform simultaneous localization and mapping (SLAM) [12] [34] [32]. SLAM algorithms passively process sensor and dead-reckoning data in order to build the best map possible of the part of the environment that the robot has seen. However, these algorithms do not actively move the robot in order to add information to the map. Often, a robot cannot construct an adequate map as it performs its mission; therefore, it must drive about its environment and build a map before it performs actions requiring this map. Algorithms for actively controlling a robot's actions for the purpose of adding information to the robot's map are called autonomous exploration algorithms.

Unfortunately, resources such as time or energy are highly constrained in many autonomous robotics missions [4]. For a robot to spend a lot of time or energy driving around and building a map before it even begins performing its intended mission is highly undesirable. As a result, a key requirement for methods of autonomous exploration is that they cause the robot to build the best map possible, while using resources as efficiently as possible. The larger the environment is, the bigger the potential exists for a robot to waste significant resources mapping that environment, and hence the more important it is for the robot's exploration strategy to be efficient.

Although a number of methods of autonomous exploration have been developed, little work has been done on optimizing the efficiency of these methods. Therefore, this thesis develops and evaluates general methods of planning observations during exploration in order to improve efficiency. A major contribution of this thesis is the development of a family of methods for planning observations over a finite horizon called finite horizon methods. In order to facilitate evaluation, we implemented and tested a variety of observation planning methods on a real and simulated robot.

In the next section we explain in more detail how SLAM algorithms and autonomous exploration algorithms work. Then, building on this explanation, we explain what it means to plan observations for autonomous exploration. We then provide a precise problem statement for the thesis, and explain the technical challenges that this problem statement presents. We next outline the architecture of the autonomous exploration system that we implemented and tested. Finally, we enumerate the claims of this thesis.

1.1 SLAM and Autonomous Exploration

SLAM algorithms take data from a robot's sensors and dead-reckoning system, and produce a map of the environment and an estimate of the robot's position within this map. These algorithms usually use state estimation techniques to keep track of the robot's estimate of the position of objects in the environment, as well as its uncertainty in these positions. There are a number of different map representations that SLAM algorithms use, including occupancy grid maps [13], scan-matched maps [18] [49] [21], and feature maps [44]. The autonomous exploration implementation that we test in this thesis uses line feature maps. Line feature-based SLAM only adds objects to its map that can be reasonably represented as a line. Selectively adding objects to the map avoids the computational burden of estimating the position of every point that the robot's sensors have ever seen. Figure 1.1 shows a typical line feature map. The thin line winding through the map is the robot's estimated path through the environment.

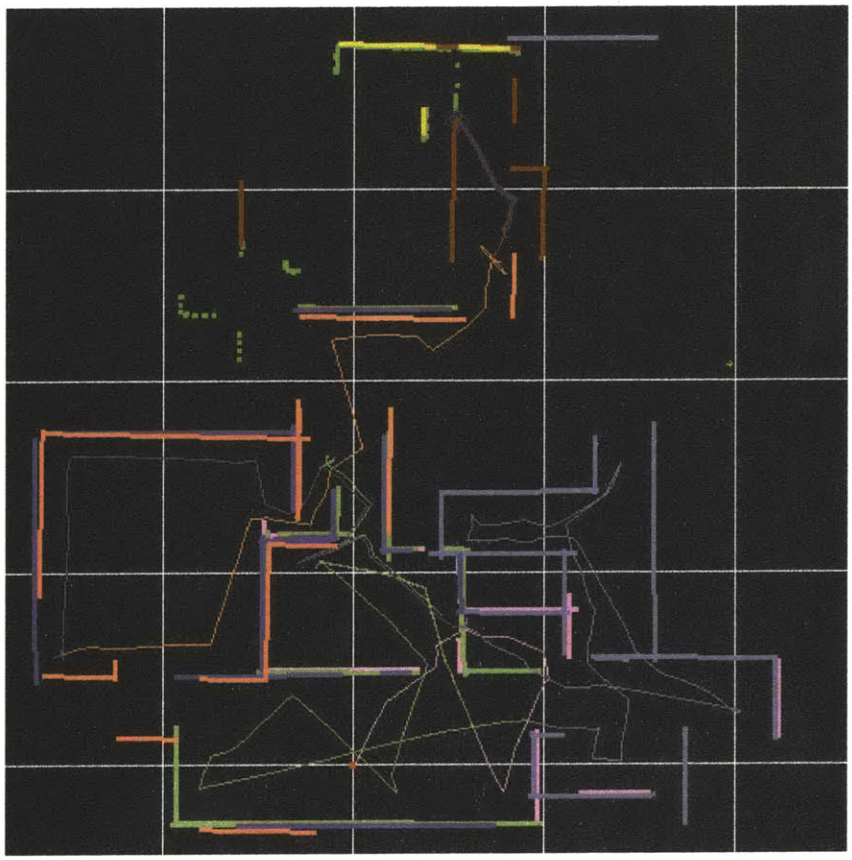


Figure 1.1 Example of a Line Feature Map

Because SLAM algorithms keep track of both the estimated positions of objects in the environment and the uncertainty in these position estimates, there are two categories of approaches to autonomous exploration. Exploration algorithms in the first category aim to decrease the map's uncertainty about the positions of objects, by having the robot re-observe these objects. Exploration algorithms in the second category aim to increase the map's coverage by having the robot observe areas it has never seen before. Less work has been done on improving the efficiency of exploration algorithms in the second category; therefore, this thesis focuses on exploration for increasing map coverage.

Most methods of exploration for increasing map coverage guide a robot's motion by placing candidate observation points on the border that separates regions of the environment that the robot has and has not sensed. These exploration methods also assign each candidate a utility that estimates how much new area the robot should see by visiting that candidate. We call approaches to placing candidates and assigning utilities to them candidate identification and scoring methods. In order to explore its environment, a robot plans a path to visit some subset of these candidate observation points. As the robot traverses this path, it continuously recalculates where to place the candidates and what utility to assign to them, in order to reflect updates to the map.

These methods of exploration for increasing map coverage, therefore, consist of two components: candidate identification and scoring, and planning a path to visit a subset of the candidates. We call planning a path to visit a subset of the candidates observation planning. If the robot constantly recalculates this path, in order to keep up to date with the changes to the continuously recalculated set of candidates, then we say that the robot is performing continuous observation planning. A number of different approaches to candidate identification and scoring exist, largely in order to handle different map representations [18] [53] [37]. However, given that most of these candidate identification and scoring methods try to place candidates on the border between explored and unexplored areas and assign a utility to each candidate estimating the amount of information the robot will gain by visiting the candidate, it is possible that one approach to observation planning will work well for all methods of candidate identification and scoring. This thesis looks for such an approach to observation planning.

In order to evaluate how various approaches to observation planning improve the efficiency of exploration, we implemented and tested these approaches for one particular candidate identification and scoring method. Specifically, we used the Newman, Bosse, and Leonard candidate identification and scoring method, because it is the only method that can handle line feature maps. This candidate identification and scoring method places candidates at either end of a line feature, in order to encourage the robot to discover the full extent of the line. The method then estimates how much new area a robot will see from each candidate, by measuring the density of features around the candidate and seeing how closely the robot has passed by the candidate in the past. If there are many line features in the map around the candidate, then the robot must have seen the area around that candidate before. In addition, if the robot's path ever passed within sensor range of the area around the candidate, then it is also likely that the robot has seen that area before. The Newman, Bosse, and Leonard method summarize these measures of how much unexplored area a robot will see from a candidate into a utility and assigns this utility to the candidate. Figure 1.2 illustrates how the method places and scores candidates for a partially completed map. The triangle in the figure represents the robot's estimated position and heading. The line coming out of the back of the robot is the robot's estimated path through the environment. The circles represent candidates, and each candidate is labeled with its utility. The straight lines in the figure are the line features of the map.

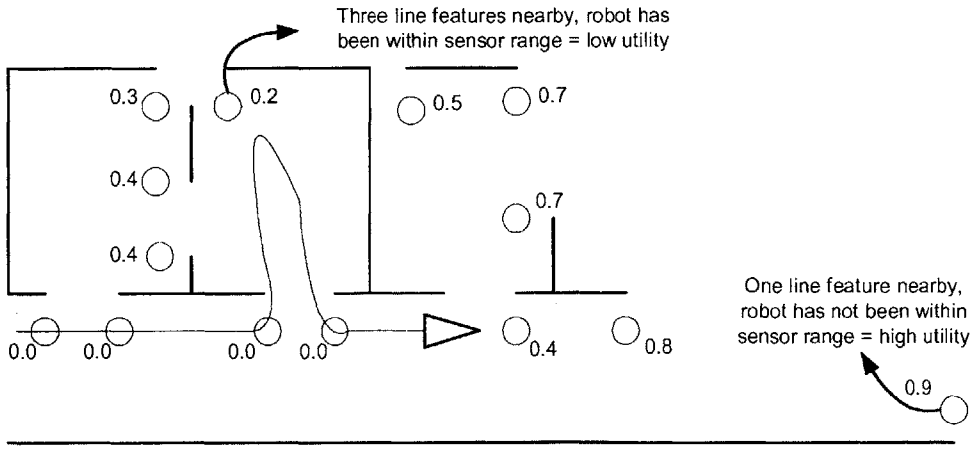


Figure 1.2 Candidate Identification and Scoring with Newman, Bosse, and Leonard Method

1.2 Approaches to Observation Planning

In order to evaluate how efficiently approaches to observation planning cause a robot to explore its environment, we must have some measure of the efficiency of the path that the robot executes during exploration. For now, we can measure the efficiency of a robot's exploration by taking the sum of the utilities of the candidates that the robot visits before the robot's path exceeds a given maximum cost. The cost of a path may be the length of a path, the amount of time that it takes the robot to traverse the path, the amount of energy that the robot expends along the path, and so on. Because the utility of a candidate estimates how much new area the robot will see from that candidate, this measure quantifies the tradeoff between the desire to maximize the amount of new area that the robot maps, with the desire to minimize the cost of the robot's path.

Currently, all methods of exploring in order to increase map coverage take the greedy approach to observation planning. The greedy approach selects one candidate for the robot to visit, and outputs the least cost path to this candidate. The candidate that the greedy approach selects is the candidate that minimizes some function $f(c_i)$, where c_i is a candidate[1]. One possible function is to return the cost of the least cost path to c_i, in which case the greedy approach selects the candidate that the robot has the lowest least cost path to. Another possible function is to return the negative of the utility of c_i, in which case the greedy approach selects the candidate that has the highest utility. Some implementations combine the previous two functions by using a function that increases as the least cost path to c_i increases and decreases as the utility of c_i increases.

The problem with the greedy approach is that it only plans paths to be locally efficient. A series of locally efficient paths, however, is not guaranteed to be globally efficient. An obvious alternative to the greedy method, then, is to plan a globally optimal path. A globally optimal path is a path that takes the robot to every candidate in the map

[1] In the way it is described here, the greedy approach would more appropriately be named the myopic approach. Myopic decision making methods produce a one-step plan in which the agent takes the action that would be optimal if the agent's life were to end immediately afterwards. Greedy methods use the same criteria to choose actions to take as myopic methods, but greedy methods produce multiple step plans. Note, however, that a robot performing continuous observation planning with a myopic method would execute the exact same path as a robot performing continuous observation planning with a greedy method. In this thesis, therefore, we do not distinguish between these two methods. Instead, we use the term "greedy method" to refer both to methods that produce paths to only one candidate and to methods that produce paths to every candidate in the map greedily.

such that no other path taking the robot to every candidate has a lower total path cost. We call the observation planning method that outputs a globally optimal path for the set of candidates the full horizon method. We refer to paths planned by the full horizon approach as full horizon paths. Finding a globally optimal path for a given set of candidates maps to solving the Traveling Salesman Problem (TSP) over the set of candidates.[2]

If the robot is able to execute a full horizon path to completion without the set of candidates changing at all, then the full horizon approach to observation planning is guaranteed to cause the robot to explore its environment at least as efficiently as any other observation planning method (as long as the maximum path cost, over which we measure efficiency, is not less than the cost of the full horizon path). Unfortunately, the set of candidates almost always changes as the robot explores its environment. Candidates appear, disappear, move, and change utility as the robot finds out more about the environment, largely because most exploration methods place candidates on the border between the explored and unexplored parts of the map. As the robot increases the coverage of its map, this border moves outward, along with all of the candidates on the border. A robot performing continuous observation planning with the full horizon approach will recalculate the full horizon path to adjust for the set of candidates changing. However, if the robot does not get to execute its full horizon paths to completion, then there is no guarantee that the parts of the full horizon paths that it does execute will be efficient at all.

In order to develop methods of planning efficient exploration paths when the set of candidates changes, we need to characterize the way in which the set of candidates changes. Unfortunately, there are currently no methods of predicting precisely how or when the set of candidates will change during exploration. However, we do know that the farther the robot travels, the more likely that the robot is to map previously unseen areas, and the more likely it is that the set of candidates will change. We, therefore, can model the times that the candidates change as an arrival-type stochastic process. Recall

[2] Burgard et al [10] first pointed out that planning a globally optimal path to explore an environment maps to solving the Traveling Salesman Problem. As far as we know, however, no one has ever implemented or evaluated a method of exploration for increasing map coverage that plans its paths by solving the TSP. The implementation and evaluation of the TSP approach for a particular candidate identification and scoring method is therefore one of the contributions of this thesis.

that we can think of a stochastic process X[t] as a sequence of random variables, such that for any value of $t = t_i$, $X[t_i]$ is a random variable. In an arrival-type stochastic process, each random variable $X[t_i]$ has two possible values: $X[t_i] = 1$ (an arrival) or $X[t_i] = 0$ (no arrival). For our purposes, the variable t corresponds to the distance that the robot has traveled, and an arrival corresponds to an instant when the set of candidates changes. We do not attempt to model *how* the set of candidates changes when it changes, since how the set of candidates changes depends strongly on the particular candidate identification and scoring method.

We can, therefore, think of an exploration mission as a series of intervals over which the candidates do not change. Each interval is separated from the interval before and after it by instants when the candidates change in an unpredictable way. With this model of how the candidates change, the best that an observation planning method can do is to plan paths that are optimally efficient over some distance in which the candidates are not likely to change. This distance could be any distance shorter than the longest interval over which the candidates do not change. A logical choice is the expected value of the distance between arrivals. To be precise, then, we would like the robot to plan a path that is not longer than a given distance (the horizon length) and that visits a subset of the candidates with the maximum possible total utility. We refer to this method of observation planning as the finite horizon method. Finding such a path corresponds to solving the Selective Traveling Salesman Problem (S-TSP) over the set of candidates. The development and characterization of the finite horizon approach to observation planning is one of the main contributions of this thesis.

There are two reasonable choices for how to perform continuous observation planning using the finite horizon approach. First, every time we recalculate the finite horizon path, we can plan over the same horizon length, L. We call this method the receding horizon method. Alternatively, each time the robot recalculates the finite horizon path, the robot can subtract from the horizon length, L, the distance it has traveled since the initial path computation and recalculate the path over this adjusted horizon. When L minus the distance traveled falls to zero, the robot resets its distance traveled to zero and starts planning a path over a distance of L again. We refer to this method of re-computation as the fixed horizon method.

Neither finite horizon continuous observation planning method is clearly better than the other. A robot using the fixed horizon method is more likely to execute the paths it plans to completion without the set of candidates changing than a robot using the receding horizon method, for the length of the robot's plan constantly grows longer in the receding horizon method. Only when the robot executes its plan to completion without the set of candidates changing, can we guarantee that the robot will execute an efficient path. In other words, it is possible that the robot will constantly put off doing something efficient when using the receding horizon method, and eventually the set of candidates will change so that the robot will never get to do the efficient thing it was planning on. Even though this situation is possible, however, it may not be likely. In addition, the fixed horizon method has the weakness that the planning horizon constantly gets shorter and shorter, thereby making the method more and more like the greedy approach. The receding horizon method, therefore, has the potential to plan much more efficient paths than the fixed horizon method.

Note that neither finite horizon continuous observation planning method plans globally optimal paths. Therefore, it is possible for a robot using the greedy or full horizon approach to execute a path that is more efficient than the path that a robot would execute using the finite horizon approach in the same situation. However, because the finite horizon approach plans a path that is optimally efficient over the expected distance that the robot will travel before the candidates change, we expect that the finite horizon approach will cause the robot to execute the most efficient paths *on average*. When the greedy or full horizon approaches cause the robot to explore more efficiently than the finite horizon approach, they must do so by getting lucky.

One legitimate concern about the finite horizon approach is that the robot might never be able to visit more than one candidate in its path before the set of candidates changes. In this case, there should not be any advantage on average to planning a path to multiple candidates, and the best the robot can do is use the greedy approach. This concern is legitimate because candidate identification and scoring methods intentionally place candidates in locations from which the robot is likely to map a lot of new area. And when the robot maps a new area, the set of candidates usually changes. Therefore, if

the candidate identification and scoring method is doing its job well, the set of candidates should change every time the robot makes an observation at a candidate.

This concern may mean that the finite horizon approach is not always the best approach to observation planning, yet there are situations in which the finite horizon approach should still perform better than the greedy (or full horizon) approach. For example, the first candidate in the robot's finite horizon path may usually not be much worse in terms of the greedy function than the greedily best candidate, depending on the environment. In these cases, if the robot is able to execute even just one of its finite horizon paths to completion, the robot still might explore its environment more efficiently using the finite horizon approach than the greedy approach. In addition, we can shorten the length of the robot's horizon in order to make it more likely that the robot will be able to execute its finite horizon paths completely.

More importantly, there are situations in which the changes to the set of candidates do not usually affect the overall efficiency of a finite horizon path. These situations occur when the robot starts off knowing the large scale structure of the environment and is mapping in order to fill in the details. If the robot knows where most of the large groups of obstacles and interesting areas to map are, then the robot can plan an efficient large scale path between these interesting areas. When the robot arrives at these areas, the robot will map new objects and the set of candidates will change, but these changes will usually only be local. The changes to the set of candidates will, therefore, not significantly affect the large scale shape and efficiency of the robot's planned path.

Figure 1.3 shows a simple example of such a situation. Figure 1.3a shows what the robot's environment actually looks like. The environment is a fictional Mars terrain, and the squares represent rocks. The rocks are grouped into clusters, and the arrangement of these clusters is what we call the large scale structure of the environment. Figure 1.3b shows the robot's initial map. This map captures the overall structure of the environment, in that we can discern the approximate location of each of the rock clusters. Figure 1.3c shows a path that the robot might plan using the finite horizon approach with a long horizon. Note that even though the robot will discover new rocks and new candidates will appear that will change the robot's path at each cluster, the overall

structure of the robot's path will remain the same. Figure 1.3d shows the approximate path a robot might take to explore this environment using the greedy approach. Note that the finite horizon path fills in the map in a much shorter distance than the greedy path.

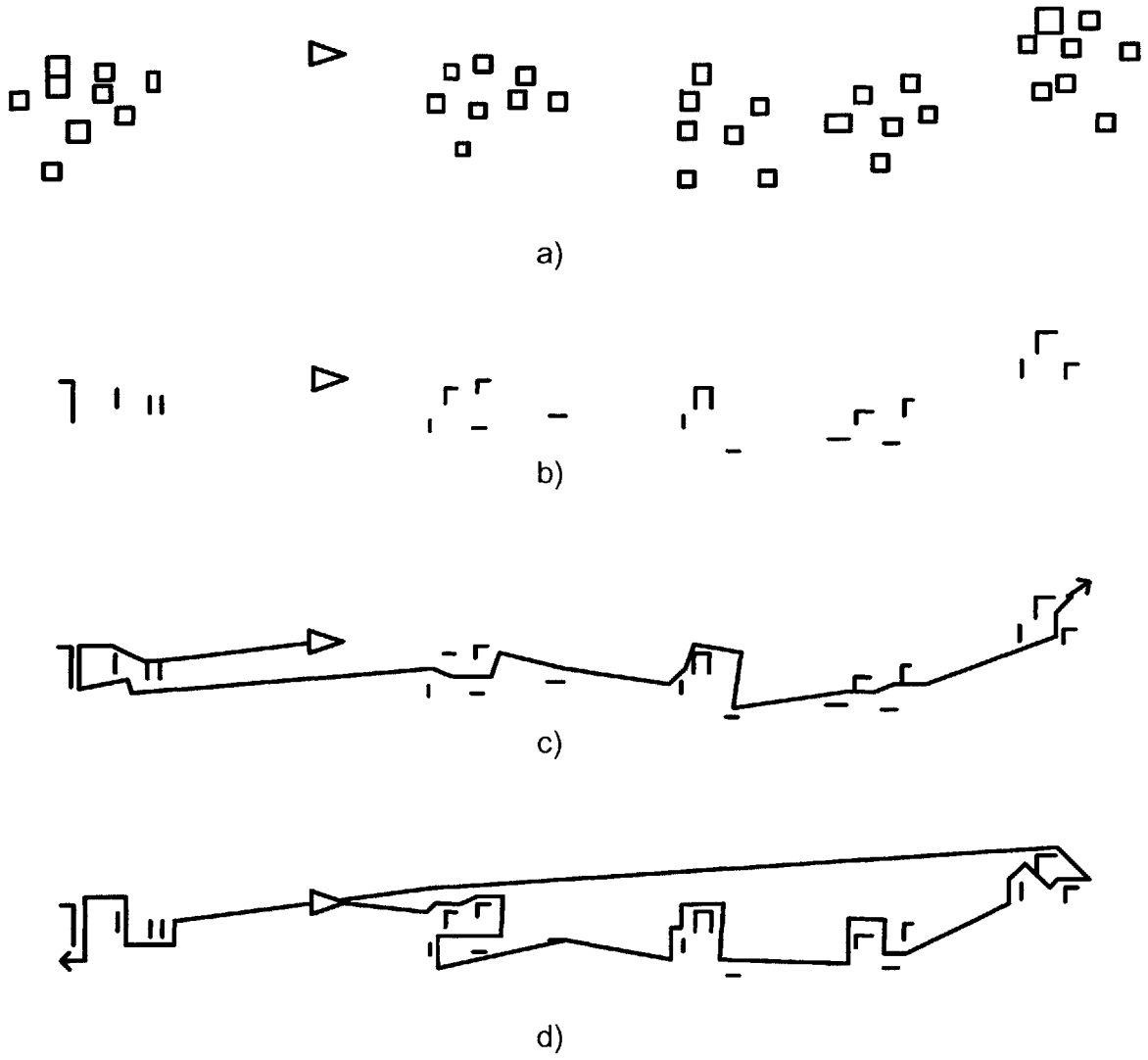


Figure 1.3 Example of Knowing Large Scale Structure Beforehand

There are a number of ways in which the robot could start off knowing the overall structure of the environment. In some situations, it is easy for us to build large scale maps of an environment using low resolution sensors. We can then provide the robot with such a map a priori. For example, overhead satellite images can produce large scale maps of an environment in the case of Mars exploration. Such large scale maps do not

contain enough detail for the robot to use for navigation or path planning, however; therefore, the robot must explore in order to flesh out its map.

Another situation in which the robot starts off knowing some of the overall structure of the environment occurs when the environment is open. An open environment is an environment in which the density of obstacles is low enough that, from most positions, the robot can see most of its surroundings, out to the radius of its sensors. Although the robot will not start off knowing the structure of the *entire* environment, for open environments, the robot will know the structure of the environment out to the radius of its sensors. Then, if the robot plans its finite horizon paths over a horizon that is on the order of the length of this sensor radius, the robot should end up filling in the details of its map in parts of the environment for which it knows the basic structure. Therefore, we expect that changes to the set of candidates will not hurt the efficiency of a robot's finite horizon path in open environments.

Ultimately, however, in order to determine for certain in which situations the finite horizon approach outperforms the greedy and full horizon approaches, we must implement and test these approaches with particular candidate identification and scoring methods. Therefore, this thesis presents the results of testing these approaches with the Newman, Bosse, and Leonard candidate identification and scoring method.

1.3 Problem Statement

The problem that this thesis addresses is to design the greedy, full horizon, and finite horizon approaches to continuous observation planning, and evaluate their efficiency with respect to how efficiently a robot explores typical environments. Our approach to solving this problem breaks down into solving three sub-problems. First, we must implement these three approaches to continuous observation planning for a particular candidate identification and scoring method, and test this implementation on typical environments. In order to solve this sub-problem, we also must address the problem of finding an efficient method of solving the S-TSP. Second, we must identify objective and quantifiable measures of how well a robot explores its environment. And third, we must evaluate how well these three approaches work for all candidate

identification and scoring methods, based on the features that candidate identification and scoring methods have in common.

1.4 Technical Challenges

Solving the sub-problem of implementing and testing the three approaches to continuous observation planning for a particular candidate identification and scoring method presents a number of technical challenges. First, performing continuous observation planning using the full horizon or finite horizon approach is computationally difficult. In order to perform continuous observation planning with either of these two approaches, the robot must constantly find the least cost path that avoids obstacles between each pair of candidates, and then solve the TSP or S-TSP over the set of candidates using these least cost paths. Because the autonomous exploration implementation must control a robot in real time, the duration of the path planning and TSP or S-TSP solving stages must be on the order of seconds at most.

The fact that our implementation uses line feature maps as the map representation presents another challenge. As far as we know, there are no existing methods of planning least cost paths that avoid obstacles using a line feature map. Therefore, we must develop a principled, effective, and efficient method of path planning for line feature maps. Another challenge in implementing these observation planning methods is that the S-TSP is an NP-hard problem [30]. Therefore, finding a fast method of solving the instances of the S-TSP we are likely to encounter has the potential to be very difficult.

One final challenge that we face is that few if any researchers have attempted to measure quantitatively how well a robot explores its environment in order to increase the coverage of its map. Therefore, must develop reasonable and objective methods to quantify the quality of a robot's exploration.

1.5 Technical Approach

Recall that, in order to evaluate the performance of the greedy, full horizon, and finite horizon approaches to observation planning, we implemented these three approaches for the Newman, Bosse, and Leonard candidate identification and scoring method described in Section 1.1. We then used the implementation of these observation

planning approaches to control a robot's exploration of indoor and outdoor environments in simulation and in the real world. This implementation and the experiments we performed with it are important contributions of this thesis.

Specifically, we put together an experimental system that continuously takes as input, real or simulated sensor data, builds a line feature map from this data, and outputs commands to a real or simulated robot that causes the robot to explore its environment using a specific observation planning method. The continuous observation planning methods we implemented were a greedy method, the full horizon method, the receding horizon method, and the fixed horizon method. Our implemented greedy method selects to visit the candidate to which the robot has the lowest least-cost path.

Figure 1.4 shows the architecture of the experimental system. The system is structured so that the only module that changes when we switch to using a different observation planning method is the module labeled "solver." The solver module takes as input a graph with one vertex for each candidate, one vertex for the robot, and edges with weights equal to the cost of the least cost path through the robot's map between the endpoints of the edge. We refer to such a graph as a candidate graph. The solver module outputs a sequence of candidates for the robot to visit. The system then turns this sequence into a path for the robot to execute. We first describe all of the modules except the solver module. We then explain how we constructed the full horizon and finite horizon versions of the solver module.

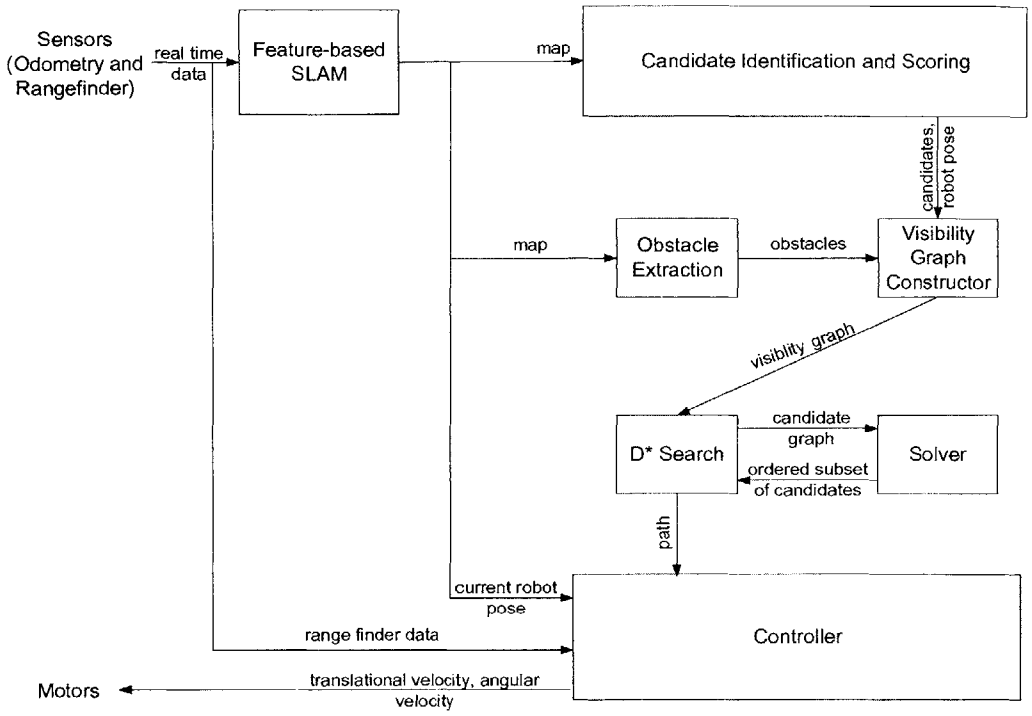


Figure 1.4 Architecture of Experimental System

1.5.1 Overall Architecture of Implementation

We examine the modules of the architecture depicted in Figure 1.4 from input to output. The feature-based SLAM module continuously takes data from the robot's odometer and rangefinder. The module uses this data to update a line feature map of the environment and to estimate the robot's position in the map. Figure 1.1 shows a typical line feature map. The SLAM module continuously passes the most recently updated map to the candidate identification and scoring module.

The candidate identification and scoring module uses the Newman, Bosse, and Leonard method described in Section 1.1 to generate a set of candidate observation points from a line feature map. These candidates are placed and scored to encourage a robot to expand its map. The module passes the set of candidates to the part of the system that performs path planning.

The part of the system that performs path planning finds a least cost path that avoids obstacles between each pair of candidates, in order to build the candidate graph. To find these least cost paths, the system builds a visibility graph over the current map and the set of candidates [51]. The module labeled "visibility graph constructor" in

Figure 1.4 builds the visibility graph from a set of polygonal obstacles and a set of waypoints. The corners of the obstacles and the waypoints make up the vertices of the visibility graph. The visibility graph constructor places edges between every pair of vertices such that the edge does not pass through an obstacle. The cost of the edge is the straight line distance between the two vertices. Therefore, any path formed by traversing edges in the graph from one vertex to another is guaranteed to avoid all known obstacles. In order to find the least cost path through the map between any two candidates, the system searches the visibility graph for the least cost path between the corresponding vertices.

The system uses the information in the line feature map to provide the visibility graph constructor with a set of polygonal obstacles. This thesis presents a novel method of extracting a set of obstacles from a line feature map, by turning each line in the map into a rectangle. The module labeled "obstacle extraction" in Figure 1.4 builds these rectangles from the feature map.

The system uses the D* algorithm [45] in order to search the visibility graph for the least cost path between every pair of candidates. As we note in Section 1.4, it is computationally challenging to constantly search the visibility graph for these shortest paths. D* addresses this challenge by incrementally searching graphs. In other words, D* saves its last calculated set of least cost paths. Then, when the map updates and a new visibility graph is built, the system tells D* what edges changed in the visibility graph. D* then searches the visibility graph only as much as it needs, in order to update its saved set of least cost paths, to accurately reflect the least cost paths through the visibility graph between every pair of candidates. The module labeled "D* search" in Figure 1.4 performs this incremental search and creates the candidate graph. The D* search module then passes the candidate graph to the solver module. The solver module returns an ordered subset of candidates to visit. The D* search module uses its stored set of least cost paths to fill in the path between these candidates. The D* search module then passes this path to the module labeled "controller" in Figure 1.4.

The controller module takes a path as input and outputs commands that cause the robot to follow this path. The controller module also performs low level obstacle avoidance to make sure the robot does not run into any obstacles that are not in the map.

In order to perform this obstacle avoidance, the controller module continuously reads the rangefinder sensor data.

1.5.2 The Solver Module

The solver module takes a candidate graph as input and outputs an ordered subset of candidates to visit. There are four versions of the solver module: the greedy version, the full horizon version, the receding horizon version, and the fixed horizon version. The greedy version simply searches through all of the edges leading out of the vertex representing the robot for the edge with the lowest weight and returns the candidate at the other end of this edge. The full horizon version solves the TSP on the candidate graph and returns the resulting sequence of candidates. The receding horizon version solves the S-TSP on the candidate graph for the constant horizon length L and returns the resulting ordered subset of candidates. The fixed horizon version solves the S-TSP on the candidate graph for a horizon length of L-d, where L is a constant and d is the distance the robot has traveled since the last horizon, and returns the resulting ordered subset of candidates.

We use the Concorde TSP code [54] directly to solve the TSP. Concorde implements an efficient branch-and-cut algorithm [1] for solving the TSP on undirected graphs. Instead of using existing algorithms to solve the S-TSP, however, we developed and implemented a new approach. In order to solve the S-TSP, we formulate the problem as an Optimal Constraint Satisfaction Problem (OCSP) [50]. An OCSP consists of a set of variables with finite domains, a set of constraints which map each assignment to the variables to true or false, and a utility function that maps each assignment to the variables to a real number. A solution to an OCSP is an assignment to the variables that maximizes the utility function such that the constraints are satisfied. A major reason that we developed an approach to solving the S-TSP by formulating it as an OCSP is that this formulation has not been previously explained. Powerful methods of solving OCSP's have recently emerged [50]; therefore, it is worthwhile to see how well these methods work for the instances of the S-TSP we are interested in.

In order to formulate the S-TSP as an OCSP, we create one variable for each candidate. Each variable can take the value of either 1 or 0. The candidate

corresponding to a variable that is assigned to 1 is included in the ordered subset of candidates that is the solution to the S-TSP, while the candidate corresponding to a variable assigned to 0 is not included. Each variable also has its own utility function, which we call an attribute utility function. This function maps a variable assigned to 1 to the utility of the corresponding candidate and a variable assigned to 0 to zero. The utility of an assignment to the entire set of variables is equal to the sum of the values of the attribute utility functions of the individual variables. In order to describe the constraint, let us consider the sub-graph formed by removing every vertex corresponding to a candidate whose variable is assigned to 0 (and every edge including such a vertex) from the candidate graph. The constraint over the OCSP variables is that the solution to the TSP on this sub-graph must have a length that is less than or equal to the horizon length L.

We solve the S-TSP formulated as an OCSP with the constraint-based A* algorithm [50]. Constraint-based A* is an efficient method based on A* search of enumerating the possible assignments to the variables from highest to lowest value of the utility function. Note that in order to maximize the utility function for a partial assignment to the variables, it is sufficient to assign each of the unassigned variables to a value that maximizes its attribute utility function. Constraint-based A* takes advantage of this fact in order to efficiently find the next best full assignment to the variables, and in order to efficiently calculate an admissible heuristic during the search.

In order to solve the S-TSP, constraint-based A* enumerates full assignments to the variables one at a time and checks the constraint for each assignment. Our approach checks the constraint on an assignment by running the Concorde TSP solver on the sub-graph corresponding to the assignment. The first full assignment that constraint-based A* finds is consistent must correspond to the subset of candidates in the solution to the S-TSP, since these assignments are generated in best first order. In order to turn this subset of candidates into an ordered subset, our approach orders the candidates in the subset in the same order that the TSP solver outputted.

1.6 Thesis Claims

1. When a robot knows the large scale structure of its environment early on during exploration for increasing map coverage, the finite horizon approach to observation planning will cause the robot to explore this environment more efficiently on average than either the greedy or the full horizon approach.

2. It is possible to solve the S-TSP as an OCSP using the constraint-based A* algorithm.

1.7 Thesis Layout

The rest of this thesis is laid out as follows. Chapter 2 characterizes the problem of autonomous exploration and provides background on map representations, existing approaches to exploration for decreasing map uncertainty, and existing approaches to exploration for increasing map coverage. In particular, the chapter describes in detail the Newman, Bosse, and Leonard candidate identification and scoring method that the implementation we tested uses. Next, Chapter 3 characterizes the general features of observation planning and motivates and defines the finite horizon approach. Chapter 3 also speculates about how well the greedy, full horizon, and finite horizon approaches should perform for all candidate identification and scoring methods in general by looking at the features that all candidate identification and scoring methods share. Chapter 4 describes in detail our approach to solving the S-TSP as an OCSP with constraint-based A*. Then, Chapter 5 explains the architecture of the system that we implemented to test the greedy, full horizon, and finite horizon approaches to observation planning. Chapter 6 presents and analyzes the results of testing the system we implemented in real and simulated environments. Finally, Chapter 7 discusses ideas for future work in areas touched upon by this thesis.

2 Autonomous Exploration

In this chapter we formulate the problem of autonomous exploration and give an overview of some of the methods for solving the problem that researchers have pursued. Finally, we describe in detail the method of exploration that the system we tested in our experiments is based on.

Specifically, in Section 2.1 we define the problem of exploration and introduce the two major variations of the problem: exploration for increasing map coverage and exploration for decreasing map uncertainty. In Section 2.1.2 we describe the general structure of an exploration algorithm that virtually all algorithms use. In Section 2.2 we review a number of the common approaches to building maps that exploration algorithms use, including the feature-based SLAM approach that the exploration algorithm we tested uses. Then, in Section 2.3, we characterize the problem of exploring in order to increase map coverage and describe a few important approaches to this type of exploration out of the many that exist. In Section 2.3.4 we describe the feature-based exploration strategy that we based the system that we tested on. Finally, in Section 2.4, we characterize exploration for decreasing map uncertainty and go over a few approaches to this type of exploration.

2.1 The Problem of Exploration

Being able to autonomously explore an environment in order to construct a map of this environment is integral to mobile robotics [10] [37] [22]. Most tasks that a mobile robot might have to perform, such as taking samples of rocks on Mars, locating ocean mines, performing urban search and rescue, or simply traveling from one location to another, require the robot to be able to navigate accurately. Yet in many environments, including indoors, underwater, or on Mars, GPS is not available to help. In addition, the odometry error for many robots is unacceptable and accumulates over time. As a result, the best way for a robot to navigate is often to use its sensors to localize itself within a

map of the surroundings. A map also functions as a model of the environment that the robot can use for path planning.

The main issue with relying on maps is that we often do not have an a priori map of the environment to give to the robot. Fortunately, algorithms now exist [38] [8] that enable a robot to use its sensors to build a map of its environment and at the same time localize itself within this map, all in real time. These algorithms are known as simultaneous localization and mapping (SLAM) algorithms. We discuss approaches to SLAM in Section 2.2.

SLAM algorithms, however, only solve part of the problem of building a map of an environment. A SLAM algorithm passively takes what the robot's sensors see and builds the best map possible from this data; it does not direct the robot to sense new areas of the environment. A robot can attempt to build its map as it moves around performing its other tasks, yet often this type of haphazard exploration of the robot's surroundings leads to an inadequate model of the environment. Therefore, we would like the robot to be able to drive itself around an environment before it performs its other tasks in order to build a good map of that environment. We call driving about for the purpose of building a map *autonomous exploration.*

2.1.1 Definition of the Problem of Autonomous Exploration

In order to be clear about what autonomous exploration algorithms do, we define the problem of autonomous exploration more formally. Given a partially completed map that is constantly updated to reflect the robot's sensor readings and the robot's estimated position, *the problem of autonomous exploration* is to have the robot control itself in order to improve this map. We explain the terms in this definition below.

By a "partially completed map," we simply mean a map that does not model every object in the robot's environment with one hundred percent accuracy. All map building algorithms output partially completed maps.

There are a number of ways that a robot can "control itself" while exploring. Some sensors allow the robot to control where and when they sense. For example, scanning sonar sensors can be told to take readings at certain angles and certain times [14]. Selecting when and where the robot's sensors take readings can decrease the

computational burden of processing this data on the robot. Yet in many robot setups, the sensors constantly scan the environment at all possible angles, and therefore the only thing the robot can control is where the robot drives to. Thus, most of the exploration algorithms we examine in this chapter only consider how to drive the robot during exploration.

Finally, there are two main ways a robot can "improve" its map of the environment: by decreasing the uncertainty in the map and by increasing the coverage of the map. A robot's sensors are inevitably noisy; therefore, SLAM algorithms represent the locations of objects in their maps with joint probability density functions. In order to decrease the uncertainty in its SLAM map, a robot must re-observe the objects in the environment that it has already mapped in such a way as to narrow the joint pdf over the locations of these objects in the map. The more focused the joint pdf is, the more certain the robot is about where the objects are located.

Conversely, in order to increase the coverage of its map, a robot must map objects and regions that it has never seen before. Therefore, these two aspects of improving a map compete. We examine methods of exploring to decrease map uncertainty and methods of exploring to increase map coverage in separate sections. Nevertheless, many exploration algorithms try to find an acceptable balance between these two ways of improving a map.

Now that we understand the requirements of the problem of exploration, we can make some general statements about how algorithms approach this problem.

2.1.2 General Features of Exploration Methods

As we have already noted, most approaches to autonomous exploration assume that the robot's sensors are constantly scanning their full range; therefore, the robot can only control how it drives around its environment. Most exploration methods deal with controlling the robot's driving by breaking down the problem into two sub-problems. The first sub-problem is to plan a path for the robot to execute that will improve the map, and the second sub-problem is to send the commands to the hardware to cause the robot to execute this path. The first sub-problem is where most of the interesting variation between exploration methods occurs, and we call it the exploration path planning

problem. More precisely, given a partially completed map and the robot's estimated position within the map, *the exploration path planning problem* is to output a path that the robot can use to effectively control its motion in order improve the map.

The simplest way for the robot to explore using this approach is to fully solve the exploration path planning problem and then execute the resulting path completely. Once the robot has executed the path to completion, it begins the loop over and solves the exploration path planning problem again. The pseudo-code in Figure 2.1 depicts this exploration method. The function `Mission_Completed()` on line 1 of Figure 2.1 returns true if the exploration mission has finished. The mission might be set to last either until the robot travels a set distance, until a certain amount of time passes, or until the user sends a command to halt the exploration. The function `Get_Most_Recent_Map(constantly updating map)` takes a dynamic map, which a map building algorithm is constantly updating, and returns a static map that reflects the most recent state of the dynamic map. Because SLAM algorithms include the robot as part of the map, the function `Get_Current_Robot_Pose(map)` can take this static map and return the location of the robot.

```
Explore(constantly updating map)
returns nothing

1.  while Mission_Completed() is false
2.    let map = Get_Most_Recent_Map(constantly
updating map)
3.    let robot pose = Get_Current_Robot_Pose(map)
4.    let path = Plan_Exploration_Path(map, robot
pose)
5.    Execute_Path(path)
6.  endwhile
```

Figure 2.1 Pseudo-code for Exploration using Basic Exploration Path Planning

The problem with this method of exploration is that the map is very likely to change as the robot executes the path it has planned. And if the map changes, the path might become sub-optimal or even impossible to execute. Therefore, another possible method of exploration is to periodically halt the execution of the path and solve the

exploration path planning problem again. If the path changes, then the robot begins to execute this new path. Figure 2.2 shows pseudo-code for this method of exploration. Line 5 is the only line that is different in Figures 2.1 and 2.2. We call this method of constantly solving the exploration path planning problem *continuous exploration path planning*. Sometimes in actual implementations of exploration algorithms, such as the implementation we used for testing, the exploration path planning code and the path execution code run in separate threads. In the path planning thread, the exploration path planning code constantly recalculates the best exploration path and notifies the execution thread when the path changes. In the execution thread, the path execution code constantly executes its current path and updates this path when it receives a notification from the path planning thread.

```
Explore_Continuous(constantly updating map)
returns nothing

1.  while Mission_Completed() is false
2.    let map = Get_Most_Recent_Map(constantly
updating map)
3.    let robot pose = Get_Current_Robot_Pose(map)
4.    let path = Plan_Exploration_Path(map, robot
pose)
5.    Execute_Segment_of_Path(path)
6.  endwhile
```

Figure 2.2 Pseudo-code for Exploration using Continuous Exploration Path Planning

All of the exploration methods that we examine in Sections 2.3 and 2.4 have a structure similar to what is depicted in either Figure 2.1 or 2.2. Therefore, our descriptions of these methods will focus on the unique ways they solve the exploration path planning problem. Before we delve into these specific exploration methods, however, we review the salient features of the SLAM algorithms that they use.

2.2 SLAM Methods

The map representation used by an exploration algorithm has a major affect on how the algorithm solves the exploration path planning problem. As a result, in this section we describe the map representations used by the exploration algorithms that we

present in Sections 2.3 and 2.4. The exploration framework we tested in our experiments uses the feature-based approach described in Section 2.2.3.

2.2.1 Scan-Matched Maps

In its basic form, scan matching is one of the most straightforward ways of building a map [18] [49] [21]. In basic scan matching, as the robot moves, it records full scans of the environment from a range-finding sensor (such as a laser scanner or scanning sonar), so that each newly recorded full scan overlaps with the previously recorded full scan. Figure 2.3a depicts two such consecutive scans for a sensor which scans 180 degrees in front of the robot. The scan-matching approach builds a map directly out of these scans. The first recorded scan forms the robot's initial map. When the second scan is recorded, the robot finds the part of the second scan and the part of the first scan that match the best. The robot translates and rotates the second scan such that the best matching parts of the first and second scan overlap. These two scans then form the new map. Figure 2.3c shows how this process would combine the scan in Figure 2.3a with the scan in Figure 2.3b. When the third scan is recorded, the robot finds the best overlap between this third scan *and the whole map*, since it is possible that the third scan overlapped with the first scan even more than it overlapped with the second scan. The robot then translates and rotates this third scan such that the overlapping regions line up, and these three scans form the new map. This map building process then repeats until the robot's mission ends.

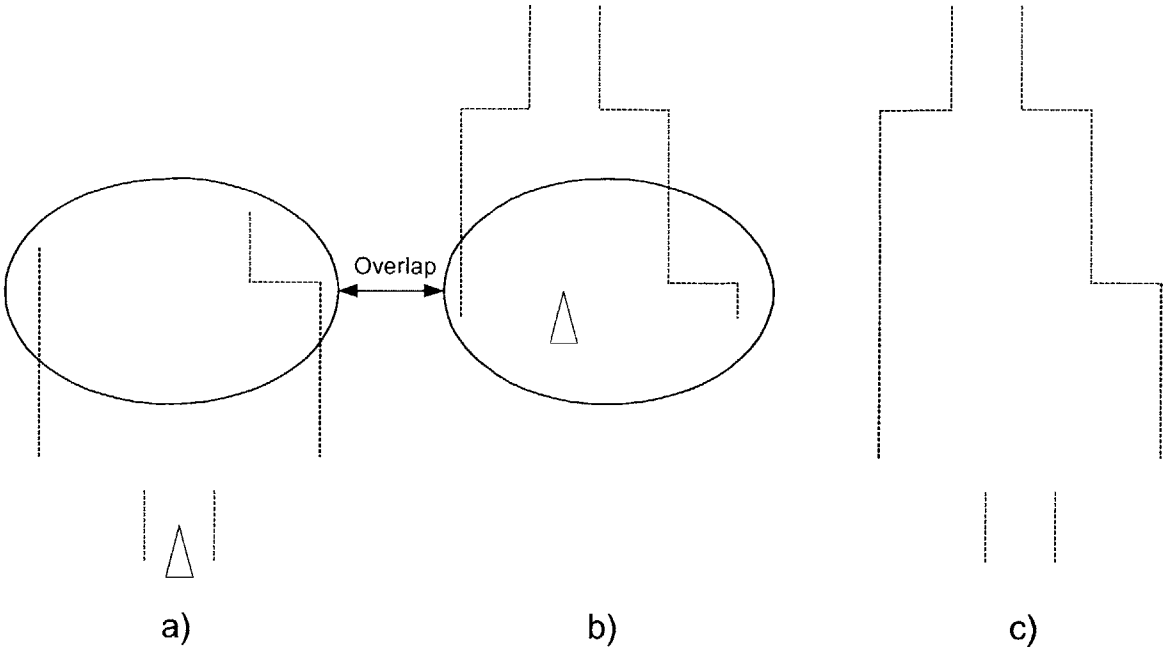


Figure 2.3 Matching Sensor Scans

It is important to note that this basic form of scan matching does not directly take into account noise in sensor scans. As a result, there is no measure of uncertainty in the map that the robot builds. In addition, once the robot has translated and rotated a new scan, this scan is stuck in that position. Therefore, the algorithm can never alter its matches to find the globally best arrangement of scans. More sophisticated versions of the scan matching approach exist which begin to address these issues [8]. However, the approach to exploration that Section 2.3.2 describes uses this basic version of scan matching.

The strength of scan matching is that the maps contain a lot of detailed information about the environment while requiring relatively little processing to create. The maps are detailed in that scan matching does not abstract or simplify what the sensor sees; scan matching builds a map directly out of the scan points. Scan matching (or at least the basic version) requires less processing than other SLAM methods because it does not perform global optimization, as we noted in the preceding paragraph.

2.2.2 Occupancy Grid Maps

Occupancy grid maps represent the environment as a collection of cells [13]. A SLAM algorithm assigns each cell in the map the probability that the location in the environment corresponding to the cell is occupied by an obstacle. There are a number of different SLAM algorithms for occupancy grid maps [47] [48], however, we do not need to cover how they work in order to understand the grid-based exploration algorithms we present in Section 2.3.3. Figure 2.4a shows the occupancy grid map corresponding to the environment in Figure 2.4b. The darker a cell is in Figure 2.4a, the more likely it is that the location in the environment that the cell represents is occupied by an obstacle.

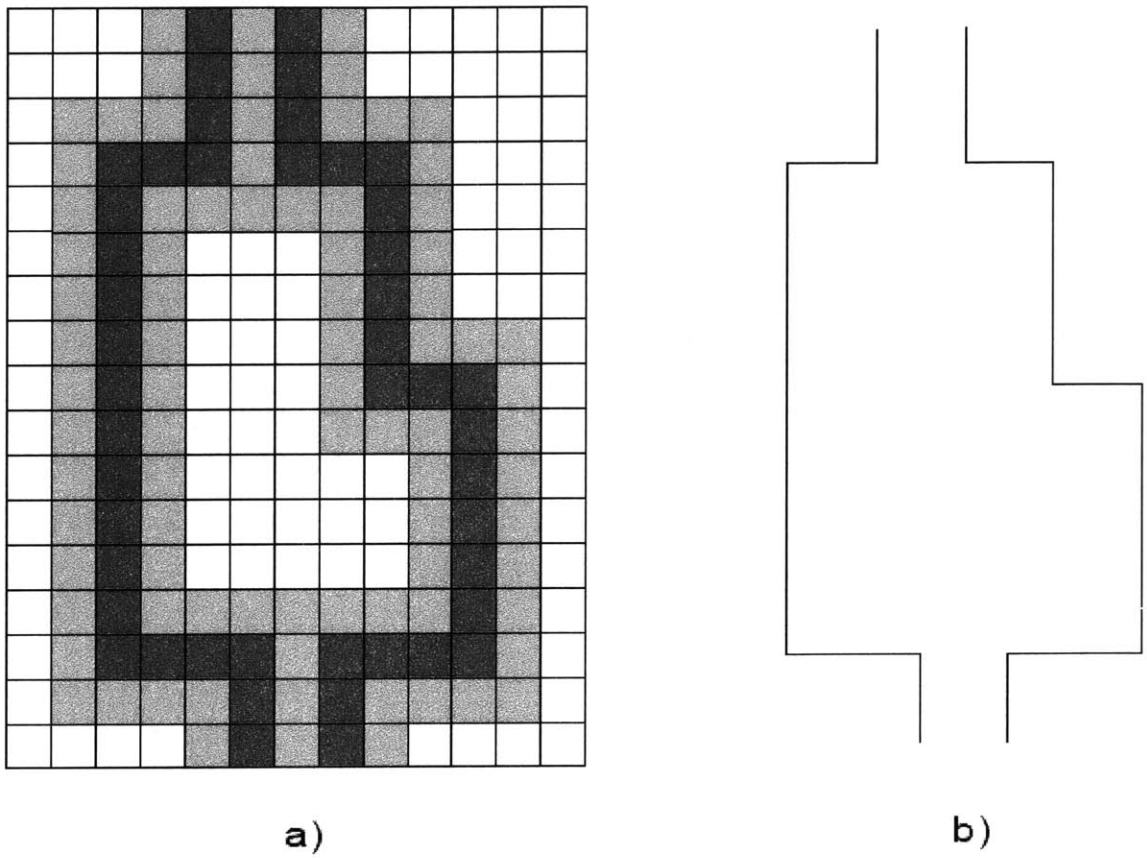


Figure 2.4 A Simple Occupancy Grid Map

Like scan-matched maps, occupancy grid maps provide a lot of detailed information about the environment they represent. If the cells are small enough, occupancy grid maps can have the same resolution as scan-matched maps. Unlike scan-matched maps, however, occupancy grid maps explicitly keep track of the areas that the

robot knows do *not* contain obstacles, by assigning the corresponding cells a low probability of being occupied. As we discuss in Section 2.3, keeping track of free space in this way can be useful in exploration for expanding map coverage.

The main drawback of occupancy grid maps is that it is very computationally intensive for SLAM algorithms to estimate the state of so many cells. The number of grid cells scales with the area (or volume) of the region covered by the map. Such scaling makes it computationally difficult to map large environments using the occupancy grid approach. Furthermore, for the purpose of navigation, it may be possible to get away with modeling the environment in much less detail. This reasoning is the motivation behind the feature-based approach to SLAM.

2.2.3 Feature-based Maps

The idea behind feature-based maps is to represent the environment with its salient landmarks instead of keeping track of every point that the robot's sensors have seen, as is done in the occupancy grid or scan matching approaches [44]. Feature-based SLAM picks out specific features, such as lines, circles, corners, or any other well-defined shape, from the sensor data. We used line feature SLAM in our experiments. The state of the world at any given moment is estimated using a vector of the position and orientation of the robot and each feature that the robot has picked out. In addition, the map represents the uncertainty in the pose of the robot and these features, by approximating the joint pdf over the state vector with a covariance matrix. A feature-based map therefore consists of this state vector and covariance matrix. Given a model of the sensor uncertainty and the vehicle dynamics, we can update the map using state estimation techniques (usually an extended Kalman filter). Figure 2.5 shows what a typical line feature SLAM map looks like. The thin line winding through the figure is the robot's estimated path through the environment.

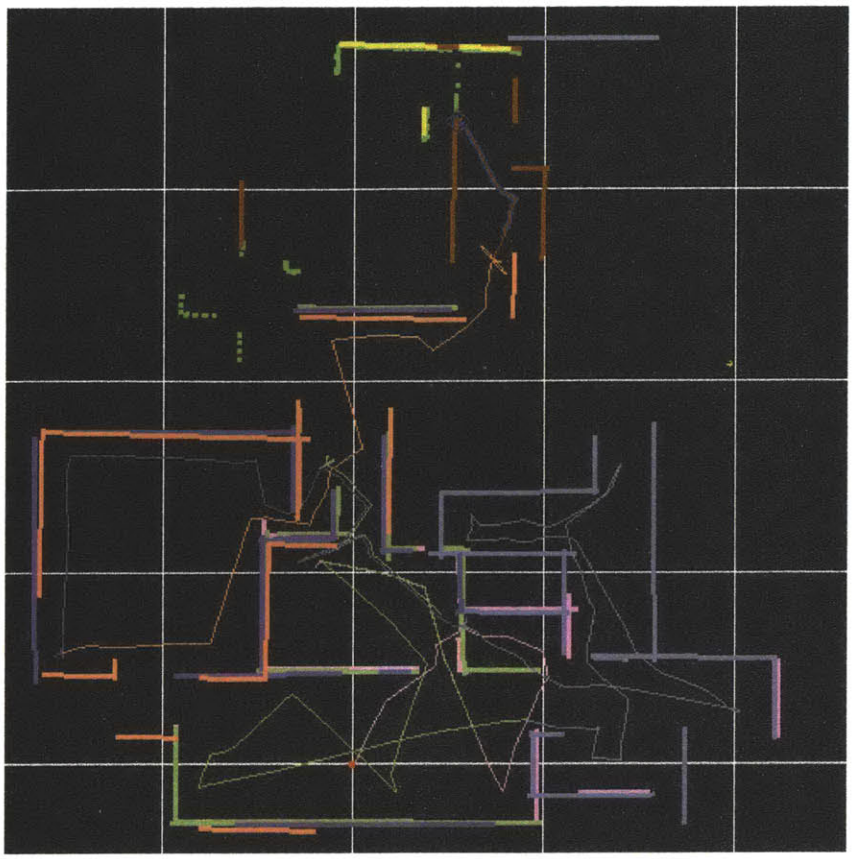


Figure 2.5 A Line Feature SLAM Map

Because they only keep track of salient landmarks, feature-based techniques are able to handle much larger environments than other approaches to SLAM. The SLAM code that we used in our experiments [5] has been used to map a complicated indoor environment that is 2.2 km in length, an order of magnitude larger than any previously published SLAM result. The code was able to autonomously close large nested loops within this environment. Feature-based SLAM also performs well with lower precision sensors, including sonar, which is important for underwater exploration.

A drawback to feature-based maps, however, is that even if the robot's sensor detects an obstacle, this obstacle may not show up in the map. Feature-based SLAM only records the parts of the scan data that look like features. Paths planned using a feature-based map are therefore not guaranteed to avoid all obstacles in the real world. In addition, it is difficult to determine what regions the robot's sensors have seen from a feature-based map, if not all of the sensor data appears in the map. This difficulty limits

the ways that we can perform exploration for increasing map coverage with feature-based maps, as we explain in Section 2.3.4.

2.3 Exploration for Increasing Map Coverage

In Section 2.1 we said that the goal of exploration for increasing map coverage is to map parts of the environment that the robot has never seen before. In other words, instead of improving the position estimates for obstacles that already exist in the map, exploration for increasing map coverage aims to add entirely new obstacles (and, in the case of occupancy grid approaches, free space areas) to the map. This goal is very general, however, and there are many different possible sub-goals that the robot can have in achieving it.

The most common sub-goal is that the robot should try to map as much new area as possible using a path with the least cost possible. Path cost can be measured in terms of distance traveled, time taken, or energy expended. In many applications of mobile robotics, including Mars rovers and AUV's [4] resources such as energy and time are very limited. Limited energy in particular is an issue with almost all mobile robots, for mobile robots rely on batteries with a finite supply of energy to perform the tough work of moving the robot. Therefore, having an exploration method that is efficient with respect to these resources is highly desirable.

As it has been stated, the robot must try to both maximize the amount of new area mapped and minimize the cost of the path. It is not obvious what it means to optimize these two competing quantities simultaneously, however. One approach is to try to optimize some combined measure of the new area mapped and path cost. The algorithm that Section 2.3.2 describes uses such a combined measure. Another approach is to have the robot try to maximize or minimize one quantity given an upper bound constraint on the other quantity. For instance, we might have the robot try to map the most new area possible without traveling more than 50 meters. This approach has the advantage that it does not rely on an arbitrary function to combine the path cost and area mapped into one quantity. In addition, the duration of many actual exploration missions is set by capping either the total path cost or the area mapped. For instance, if the robot is exploring an indoor environment, a number of exploration methods [18] [10] [22] have the robot

explore until it has seen the entire interior of the building. In Mars missions, the robot often has a set schedule to stay on; therefore, the robot might allot only a specific amount of time to exploring the environment. In missions like these where one quantity is limited, it is natural to define efficient exploration as exploration that optimizes the other quantity.

There are many other sub-goals that a robot can have when it is exploring, in order to increase map coverage. One such sub-goal is to explore the environment thoroughly. The term "thorough" is a difficult term to define, however. The way we define thorough when evaluating the results of our experiments in Chapter 6 is, after the robot has finished exploring, to draw a border surrounding the parts of the environment that the robot visited. We then evaluate how much of the environment inside of this border does not appear in the robot's final map. Fewer parts of the environment missing from the map means a more thorough exploration of the environment. If we can break down the environment into discrete regions (a room could be a region, for example), we might alternatively interpret thorough exploration to mean that the robot does not move on to a new region until it has mapped everything in its current region. Thorough exploration is important if the robot plans on using the map for path planning, because in path planning it is better if the robot knows where all the obstacles are along its path before it executes that path.

Another possible sub-goal that a robot might have, while exploring to increase map coverage, is to first build a rough global map that captures the large scale features in the environment, and then incrementally improve the resolution of the map. This sub-goal is therefore the opposite of the sub-goal of exploring thoroughly. The idea behind this sub-goal is that in some cases, it is much more valuable to have a map that completely covers the robot's environment, even if the map has very low resolution, than to have a high resolution map that only covers the robot's local region. In fact, the approach to planning observations that this thesis explores performs best when it starts out with a rough global map of the environment. If we do not know beforehand when the robot's mission will end, or if we want to have a global map to use as quickly as possible, then it makes sense to build a quick low resolution map and then slowly increase the resolution, even if doing so is not the most efficient way of getting to the final version of

the map. Note that in order to build a rough global map of the environment, we must have some way of limiting how far out "global" extends. Therefore, the sub-goal makes the most sense in enclosed indoor environments.

It is difficult to quantify how well an exploration method achieves this sub-goal. One important measure is how quickly the robot is able to build a map of any resolution that covers the whole global environment. Yet it is unclear where the boundary lies between a map not covering the entire environment and the map covering the environment, but being very low resolution. In addition, we usually want low resolution maps to pick out the important large-scale features of the environment, however, what the important large-scale features are in an environment is often difficult to define.

Another possible sub-goal that a robot might have is to get from one location to another. For example, if a robot was placed in a maze and wanted to get out, it would need to explore the maze and grow its map in order to find the exit. We can measure how well an exploration strategy works in terms of this sub-goal by evaluating the cost of the path that the robot executes in getting to its destination. Exploration strategies for getting to a destination in an unknown environment should probably direct exploration towards this destination somehow.

There are a large number of other possible sub-goals. For example, the robot might only be interested in mapping certain types of obstacles, might be looking for a particular obstacle, or might only be interested in mapping certain regions of the environment. These sub-goals are less general than the previous four we mentioned, however. We therefore will not look at approaches for achieving these sub-goals.

2.3.1 General Features of Methods of Exploration for Increasing Map Coverage

Before we examine specific methods of exploring to achieve the first four sub-goals that we mentioned, we analyze some of the general features of these methods. We begin by considering what an ideal approach to exploration for increasing map coverage would entail, and then explain how actual approaches approximate this ideal approach.

We assume that the overall exploration strategy is to use continuous exploration path planning; therefore, we consider what the ideal way of solving the exploration path

planning problem would entail. In Section 2.1.1 we noted that in many setups, the robot's sensors are constantly scanning the environment. In other words, the robot is not limited to driving to a location, taking a scan of the surroundings, and then driving to another location. Therefore, the robot's position and heading at each instant is potentially important, and the ideal exploration method for increasing map coverage would plan each infinitesimally small step of the robot's path in order to take into account the robot's pose at each instant.

In addition, no matter what sub-goal guides the robot's exploration, it is almost always desirable that the path that the robot executes in achieving this sub-goal have the least cost possible. Planning an efficient path in exploration for increasing map coverage is very difficult because every time the robot sees a new region, the map that the robot is planning an exploration path for changes. Therefore, if the robot was able to predict with one hundred percent accuracy what it would see everywhere it went, then it would generally be able to plan and execute a much more efficient path than the path it would execute if it did not make any predictions. An ideal approach to exploring to increase map coverage, therefore, would make probabilistic predictions as to what the robot will see at each new region it visits. The robot could make these predictions by using statistical learning techniques to learn regularities in the structure of the environment, or by having some a priori knowledge about these regularities.

Existing methods of exploring to increase map coverage only approximate these features of an ideal approach. As in exploration for decreasing map uncertainty, it is too computationally challenging to plan each infinitesimally small step in a robot's path when exploring for increasing map coverage. Therefore, existing exploration methods drastically reduce the number of possible paths that they consider, by placing a finite number of candidate observation points at the border separating the parts of the environment that the robot has and has not explored [18] [53] [10]. The robot then only needs to choose some subset of these candidate observation points to visit and an order to visit them. The final output path of the exploration path planner is the least-cost path that takes the robot to these candidate observation points in the chosen order, stays within the part of the environment that the robot has already explored, and avoids all obstacles. The justification for this simplification of the ideal approach to exploration path planning is

that even if the robot makes probabilistic predictions about what the unexplored regions of the map look like, it is still probably inefficient for the robot to try to plan a path through these unexplored regions. Therefore, when exploring to increase map coverage, the robot should try to plan exploration paths that stay within the region of the environment that the robot has explored already. In order to observe unexplored regions of the environment, the robot must travel to the edge of the explored region of the environment. Exploration methods therefore place candidates at the edge of the explored region, and the robot plans its exploration path to visit these candidates. The robot does not need to worry about what it will observe on the way to these candidates because the robot will be traveling through the part of the environment it has already explored. Therefore, these exploration methods might as well return paths that are composed of shortest paths between the candidates as the solutions to the exploration path planning problem.

Exploration methods also approximate making predictions about what the robot will see in unexplored regions in order to plan efficient paths. Although using statistical learning techniques or a priori knowledge to predict what the robot will see at unexplored locations may be computationally feasible, no published exploration method utilizes either approach to prediction. The closest that existing exploration algorithms come to predicting what the robot will see at unexplored locations, is to give a score to each candidate observation point, estimating how much new area the robot will see from that point. These scores enable the robot to greedily choose a candidate to visit next that should add a lot of new area to the robot's map. However, in order to be able to plan more globally efficient paths, the robot needs to be able to predict what the environment will look like at each unexplored location in detail, or at least be able to predict how the set of candidate observation points will change when it visits an unexplored location. Therefore, giving each candidate observation point a score is not the ideal method of improving the efficiency of the robot's exploration. Much work remains to be done in incorporating more sophisticated methods of prediction into exploration algorithms. In Chapter 7 we discuss ways in which future research could use probabilistic predictions as to what the robot will see in unexplored regions, in order to plan more efficient paths on average.

Nevertheless, placing candidate observation points near the boundary between explored and unexplored areas of the map and scoring these candidates is the basic approach that all of the following exploration methods take. The major differences between these exploration methods are in the particular ways these methods place and score their candidates. In the following three sections, we examine existing approaches to exploration for scan-matched maps, occupancy grid maps, and feature-based maps respectively.

2.3.2 The Gonzalez-Banos and Latombe Method

The Gonzalez-Banos and Latombe exploration method is one of the only existing methods that uses scan-matched maps as its sole map representation [18]. This method is, perhaps, the one that most closely follows the general characterization in Section 2.3.1; thus, it can be thought of as the prototypical exploration method. The key contribution of the Gonzalez-Banos and Latombe method is how it scores candidates. The method assigns a utility to a candidate that is an optimistic estimate of the amount of unexplored area the robot will see from the candidate. Figure 2.6 depicts such an optimistic estimate for two candidates in a partially completed map. The small circles in the figure are the candidates. The large circle around each candidate has a radius equal to the range of the robot's rangefinder. Therefore, the maximum amount of unexplored area a robot's sensor could see in a 360 degree scan from a candidate is given by the striped part of the surrounding large circle. The area of this striped part is the utility of the corresponding candidate.

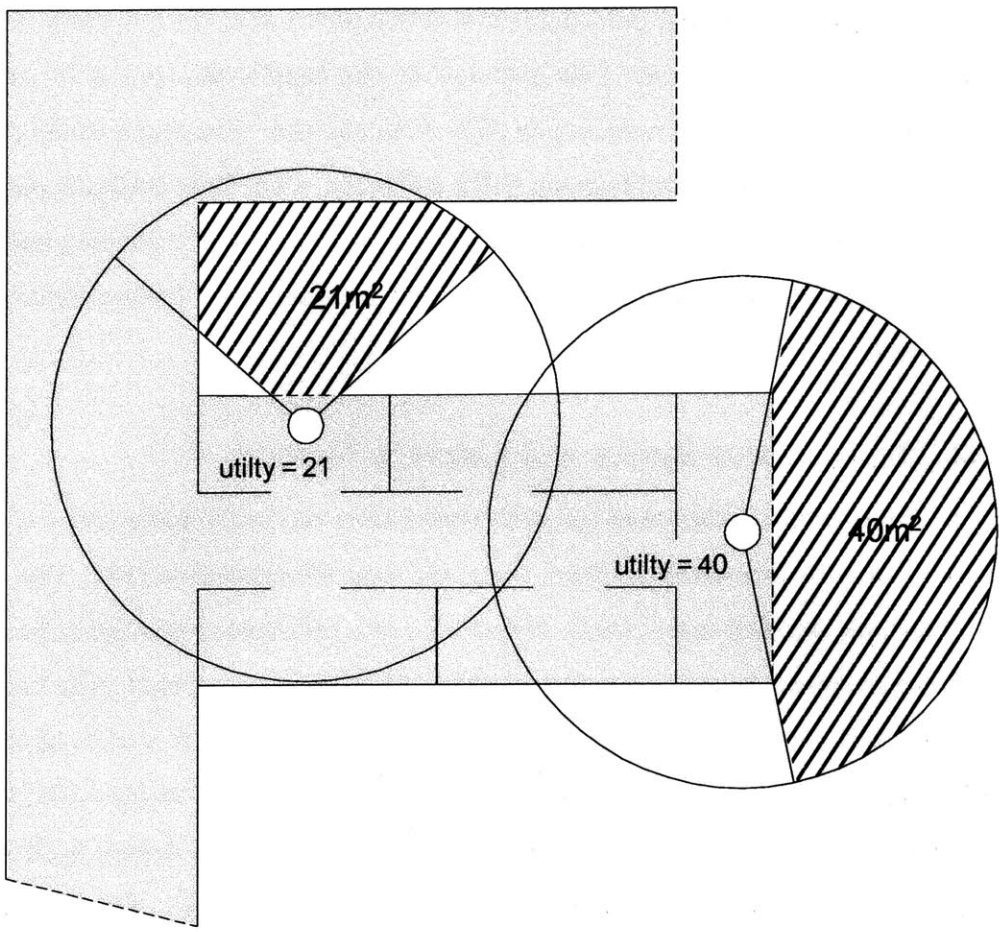


Figure 2.6 Candidate Scoring for the Gonzalez-Banos and Latombe Method

In order to estimate the maximum amount of unexplored area the robot will see from a candidate, Gonzalez-Banos and Latombe developed a method for estimating what regions of a scan-matched map the robot's sensors have definitely seen. Estimating these regions is difficult because, unlike occupancy grid maps, scan-matched maps do not keep track of areas that the robot's rangefinder scanned but did not detect any objects in. To address this difficulty, the method constantly calculates the "safe region," which is the region of the environment that the robot has sensed and did not see any objects in. In order to construct the safe region, at each candidate observation point (and the robot's initial position) the robot takes a 360 degree scan of its environment, constructs a safe region for that scan, and then adds this new safe region to the combined safe regions of all previous scans.

In order to construct a safe region for a scan, the method fits curves to the scan points that result from the scan. In other words, each scan of the robot's sensor results in a set of curves that can be described as a discontinuous function of the scan angle. This radial function is defined for all angles except those in which the nearest object was beyond the sensor's maximum range. The data points that result from a given scan represent detected obstacles; therefore, the curves that are extracted from these points are called "solid curves." Construction of the safe region for a scan continues by connecting adjacent solid curves with new curves called "free curves." Gonzalez-Banos and Latombe give a deterministic method of calculating free curves from a set of solid curves such that the interior of the region formed is guaranteed to have no obstacles in it. The method then adds the safe region for a given scan to the combined safe regions of all previous scans by matching the two safe regions, just like scans are matched in basic scan matching. Only solid curves are used in matching safe regions, however.

Once the safe region for a scan has been calculated and integrated into the combined safe region, the method places candidate observation points inside the combined safe region. More specifically, the method randomly places a set number of candidates in the combined safe region, subject to the constraint that each one is within sensor range of a free curve. We should only care about points that are near free curves because there is no chance that the robot could see through solid curves to a new location. The method does not keep all of these candidates, however. The lower the amount of known solid curve that is visible from a candidate, the harder it will be to match the resulting safe region to the combined safe region. Therefore, if the length of known solid curve that is visible from a given candidate is less than a set threshold value, then the candidate is removed. In addition, the robot removes all candidates to which it has no free path that stays inside of the safe region.

The method is then ready to assign each remaining candidate observation point a utility that is an optimistic estimate of the amount of new area the robot will see from the candidate. In order to calculate this utility, the robot assumes that every free curve in the combined safe region is entirely transparent. In Figure 2.6, the free curves are the dashed lines. The robot also assumes that there are no objects in the environment that are not in the map. The robot then measures how much area outside of the combined safe region its

rangefinder would be able to cover from the candidate if the robot took a 360 degree scan there. If there were no solid curves in the map, then in Figure 2.6, this area would be the area encompassed by the large circles that is not inside of the combined safe region. The combined safe region is shaded gray in Figure 2.6. However, the robot must take into account the fact that its sensors cannot penetrate solid curves when measuring the area outside of the combined safe region that its sensors will see from a candidate. Therefore, in Figure 2.6, the striped region inside of the large circles is the correct area. The utility of a candidate is equal to this area, as Figure 2.6 shows. Thus, the amount of area the robot actually sees from the candidate will be less than or equal to the utility of the candidate.

Finally, the method selects as the point to visit next the candidate that maximizes the function g(candidate) = utility(candidate) * exp(-c * L(candidate)), where c is a constant and L(candidate) is the length of the shortest path from the robot to the candidate that avoids all obstacles and stays inside of the safe region. The method outputs this shortest path from the robot to the next candidate to visit as the solution to the exploration path planning problem. The robot will then execute this path, take a new 360 degree scan at the end of this path, calculate a safe region from the scan, and so on.

An additional interesting feature of the Gonzalez-Banos and Latombe method is that the method tries to allow the user to have some choice over which of the exploration sub-goals from Section 2.3 the method should have. Specifically, the parameter c is meant to allow a user to tailor the exploration method for the sub-goal of thoroughness or the sub-goal of building a rough global map before filling in details. Note that setting the constant c in g(candidate) to be a large number causes the method to favor candidates that are close to the robot and to not pay much attention to the candidate's score. Therefore, increasing the size of c should cause the robot to explore more thoroughly. Conversely, setting c to be a small number causes the robot to favor candidates that have a high score and to not pay much attention to how far away the candidate is from the robot. Gonzalez-Banos and Latombe argue that because a candidate's score measures roughly how much new area the robot will add to the map by going to that candidate, setting c to a low value will cause the robot to at first explore its environment roughly by mapping only the most unexplored areas. As time passes, the robot will fill in its map in more and more detail.

2.3.3 Methods for Occupancy Grid Maps

There is a wide range of exploration methods that use occupancy grids as the main map representation [53] [10] [2] [22]. The exploration methods for occupancy grids that most resemble the general characterization of methods of exploration in Section 2.3.1 are those that use "frontiers" to guide exploration [53] [10]. We therefore focus on describing the way frontier-based exploration methods work in this section. Most methods that use frontiers aim to map as much new area as possible while executing a path with the lowest possible cost. In particular, the method that Burgard et al developed aims to explore the entire environment in the least amount of time possible. Frontier-based methods use the exploration path planning framework; therefore, we examine how these methods solve the exploration path planning problem.

The important feature to note about frontier-based methods of exploration is how they choose where to place candidates. The main idea is that, like the Gonzalez-Banos and Latombe method, frontier-based methods place candidates at segments of the boundary of explored territory where there is a clear view of unexplored territory. These segments of the boundary of explored territory are called frontiers. In order to identify frontiers, frontier-based methods classify each cell in the map as either unknown, occupied, or open. An unknown cell is a cell whose probability of being occupied is equal to the a priori probability that a cell is occupied. An occupied cell is a cell whose probability of being occupied is greater than this a priori probability, and an open cell is a cell whose probability of being occupied is less than the a priori probability. A frontier cell is an open cell that is next to an unknown cell. Because open cells correspond to areas of the environment that the robot has sensed and has reason to believe do not contain any obstacles, and because unknown cells usually correspond to areas of the environment that the robot has not sensed, placing candidates in frontier cells is analogous to placing candidates near free curves in the Gonzalez-Banos and Latombe method. Therefore, the robot should be able to see into unexplored territory from a frontier cell. A frontier is a group of adjacent frontier cells longer than a given length. Frontier-based exploration methods can place one candidate in every frontier cell [10] or place a candidate in the middle of each frontier [53].

Existing frontier-based exploration methods do not assign a utility to each candidate that directly estimates how much new area the robot should see from that candidate. Instead, these exploration methods simply choose the candidate that the robot has the least-cost path to as the next candidate to visit. The output of the exploration path planner is, therefore, the best path to this next candidate. However, there is no reason why these methods could not assign utilities to candidates in a way similar to the way the Gonzales-Banos and Latombe method assigns utilities.

The method of Burgard et al does assign utilities to candidates, but it does so only to take into account candidate interactions. The Burgard et al method plans paths for multiple robots exploring the same environment simultaneously. In the method, the utility of each candidate starts off at one. The method then finds the robot and candidate that are closest together and assigns the robot to visit that candidate. When the robot gets to this candidate, however, the new area the robot maps might overlap with the new area a robot would see from other candidates; in other words the candidates might interact. This candidate interaction means that once the robot gets to the candidate it was assigned to, the candidates that overlap with this candidate will be less desirable to visit than they were before. Therefore, before the method assigns a second robot to visit a candidate, the method decrements the utility of each candidate according to the estimated probability that the candidate is visible from the candidate that the first robot was just assigned to visit. The more probable it is that the candidate is visible from the candidate that the first robot was just assigned to visit, the more the candidate's utility is decremented. Once the exploration method has decremented the utility of every candidate, it finds the robot and the candidate that maximizes the value of the utility of the candidate minus the cost of the shortest path from the robot to the candidate and assigns the robot to visit the candidate. The method then decrements the utility of the remaining candidates, and the process repeats.

In order to estimate the probability that one candidate is visible from another candidate, the method of Burgard et al keeps a record of the distances that the robots measure with their sensors while exploring. The method then measures the straight line distance between the two candidates in question and assigns a probability that is equal to the fraction of the time that the robots have measured similar distances in the past. This

method of assigning utilities to the candidates is interesting because it introduces a new way of making a simple prediction about what the robot will see from a given candidate. Even though the method only uses these predictions to mitigate the effects of candidate interactions, these predictions could also be used to assign a utility to each candidate that reflects how much new area the robot should see from a candidate. In addition, candidate interactions are also a problem when a single robot tries to plan a path more than one candidate long. Therefore, future work could use this method of mitigating the effects of candidate interactions to improve the paths planned for single robot exploration.

There are many other approaches to exploration with occupancy grids besides the frontier approach. Some of these methods store information for each cell, in addition to the probability that the cell is occupied, such as number of times a sensor has scanned the part of the world corresponding to the cell [2], or how many times the robot has visited the part of the world corresponding to the cell [2] [22]. These methods also address other sub-goals, such as thoroughly exploring the environment [22]. One method can even mediate between many different sub-goals simultaneously, including the sub-goals of getting from one point to another and decreasing map uncertainty [2]. There are many interesting possibilities for methods of exploration for increasing map coverage using occupancy grids, and there is much room for more work in this field.

2.3.4 The Newman, Bosse, and Leonard Method

The Newman, Bosse, and Leonard exploration method is the basis of the approach to exploration that we implemented and tested in this thesis [37]. Chapter 6 presents and analyzes the results of these experiments. While a few exploration methods use feature-based SLAM maps in conjunction with occupancy grid maps [2] [9], the Newman, Bosse, and Leonard method is the only exploration method we know of that uses a feature-based SLAM map as the sole map representation. The Newman, Bosse, and Leonard method aims to achieve two sub-goals: to map as much new area with the least-cost path possible and to map the environment thoroughly. The method uses the continuous exploration path planning framework; therefore we focus on explaining how the method solves the exploration path planning problem.

Identifying locations to place candidates and evaluating how much new area the robot will see at these candidates is more difficult with feature-based maps than it is with occupancy grid or scan matched maps. This difficulty arises from the fact that it is not straightforward to identify areas of the environment that the robot has not explored with a feature map. One of feature-based SLAM's important properties is that it does not keep track of every point a robot's sensors have seen. This property makes it hard to determine whether an empty region on a feature map has never been explored, has been explored and is empty, or has been explored and is not empty, but the robot has not extracted any features there. Therefore, instead of trying to keep track exhaustively of regions of the environment that the robot's sensors have seen, the Newman, Bosse, and Leonard approach identifies features in the map that look like they have not been fully extracted and sends the robot to explore them further.

More specifically, each feature in the map generates candidate observation points near itself. The idea is that each feature should have its own theory about how to explore its surrounding area, in order to completely fill itself in or to discover new features nearby. Therefore, different types of features may have different strategies for placing candidate observation points. For example, line features place candidate observation points at either endpoint, in order to encourage the robot to discover the full spatial extent of the line. Figure 2.7 shows how candidate observation points can guide a robot in order to fill in a map of a room. Figure 2.7a gives the actual layout of the hypothetical room, and Figure 2.7b shows the candidate observation points (the circles) for a partial map. Note that the candidates encourage the robot to fill in the remaining details of the room.

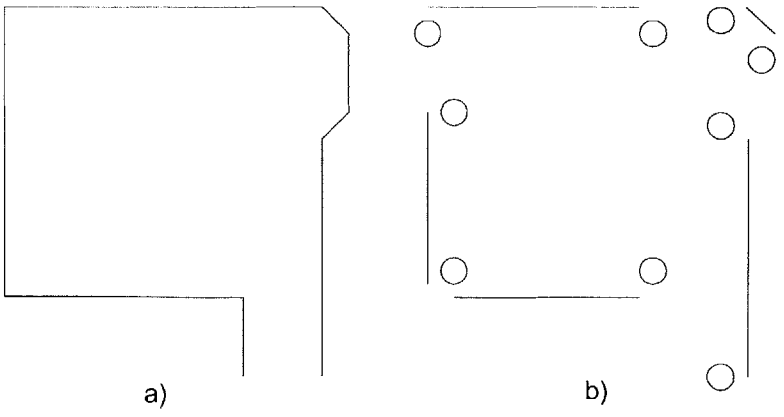


Figure 2.7 Candidate Observation Points for a Partial Map

The exploration method assigns each candidate a utility that estimates how much new area the robot will see from the candidate by estimating the amount of area in the immediate vicinity of the candidate the robot has not explored. The method estimates how unexplored the area around a candidate is in two ways. First, the method examines the robot's past trajectory and evaluates whether or not the robot's sensors should already have scanned the area around the candidate. The method keeps track of the robot's trajectory by dropping a "pebble" at the robot's estimated position whenever the robot moves a specified fixed distance. The method also places a specified fixed number of "sample points" in a circle of a given radius around the candidate in question. This circle of sample points delineates the boundary of the immediate vicinity of the candidate. In order to measure whether or not the robot should already have sensed the area in the immediate vicinity of candidate somewhere along its trajectory, the method tests whether or not there is a clear line of sight between any of the sample points surrounding the candidate and any pebble within sensor range of the sample points. The method invalidates each sample point that is within sight of a pebble. After the second step of the candidate evaluation has finished, the utility of the candidate will be the number of sample points around the candidate that are still valid divided by the original number of sample points.

Figure 2.8 depicts the evaluation of the two candidates G1 and G2. The pentagons in the figure are the pebbles that mark the robot's trajectory. The small unlabeled circles around the candidates are the sample points. Four of G1's sample points have a clear line of sight to pebbles, and these pebbles are within sensor range of the sample points. These four sample points are therefore invalidated. In contrast, none of G2's sample points are within sensor range of any of the pebbles. Therefore, even though there is a clear line of sight between some of G2's sample points and some pebbles, none of G2's sample points are invalidated.

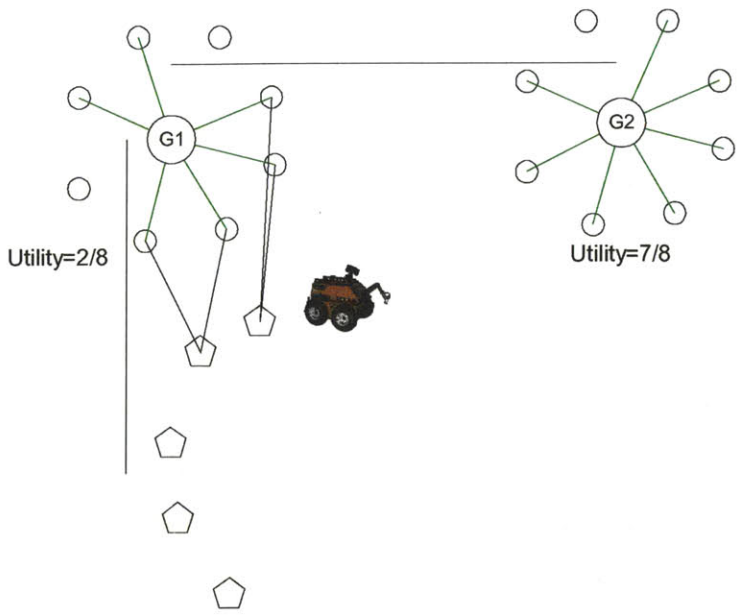


Figure 2.8 Evaluating Candidate Observation Points

The second way the method evaluates how much of the area around a candidate the robot has not yet explored is by counting how many features there are in the candidate's immediate vicinity. If a candidate is surrounded by many features, then it is given a lower utility because the robot has definitely already sensed the region. In addition, the robot will have a hard time moving through these surrounding features when it is trying to determine the spatial extent of the candidate's parent feature. More precisely, the method calculates how densely packed with features the area around a candidate is by seeing how many of the candidate's sample points have a clear line of sight to the candidate. If a sample point does not have a clear line of sight to the candidate, then there must be a feature between the sample point and the candidate. Because the sample points delineate the immediate vicinity of the candidate, this blocking feature must be in the immediate vicinity of the candidate. Therefore, each sample point that does not have a clear line of sight to the candidate is invalidated. This invalidation decreases the utility of the candidate because the utility of the candidate is the number of valid sample points of the candidate divided by the candidate's initial number of the sample points.

In Figure 2.8, two of candidate G1's sample points do not have a clear line of sight to G1. Therefore, these two sample points are invalidated. In addition, one of

candidate G2's sample points does not have a clear line of sight to G2. As a result, this sample point is invalidated. It makes sense that more of G1's sample points are invalidated in this way than G2's because G1 is more surrounded by features than G2 is. In the end, G1 is left with two valid candidates; therefore, the method assigns G1 the utility of 2/8 = 0.25. G2 is left with seven valid candidates; therefore, the method assigns G2 the utility of 7/8 = 0.875.

We have now seen how the Newman, Bosse, and Leonard method identifies and scores candidate observation points. After identifying and scoring the candidates, the method picks a candidate to visit roughly by choosing the candidate with the highest utility. The output of the exploration path planning is therefore a path to this candidate. The exact details of how the method chooses this candidate and calculates a path to it are slightly complicated, and we do not present them here. In the implementation we tested, we only used the candidate identification and scoring part of the method. In Chapter 5 we explain how our implementation computes a path for the robot to execute from the set of candidates.

2.4 Exploration for Decreasing Map Uncertainty

Almost all exploration methods take into account how the path the method plans affects the uncertainty of the robot's map. Even methods of exploration for increasing map coverage try to make sure the robot does not get lost. For instance, the Newman, Bosse, and Leonard method places candidates near features in part to encourage the robot to stay localized by keeping within sight of features it knows about. Yet there is a class of exploration methods whose sole goal is to decrease the map's uncertainty in the positions of the objects already in the map and the position of the robot. These methods do not aim to add new objects to the map at all (although most of the methods can handle adding new objects to the map if they are sensed). Most of these methods aim to explore efficiently by decreasing the overall uncertainty of the map as much as possible in the least amount of time possible.

In order to decrease the overall map uncertainty, all of these exploration methods face a tradeoff 0. The way the robot decreases its uncertainty in the location of an object

is to sense that object again. Yet in order to sense an object, the robot needs to move to the object. Moving increases the robot's localization error because all robots have noisy odometry to some degree. This increase in localization error increases the overall map uncertainty by increasing the uncertainty in the position of the robot. In addition, the more uncertain the robot is about its own position, the less certain the robot will be about the position of an object that it re-observes. The amount of information the robot gains by re-observing an object therefore depends strongly on the path that the robot takes to the object. So unlike exploration methods for increasing map coverage, methods for decreasing map uncertainty cannot simply pick out a set of candidate observation points and assign to each candidate a utility that estimates how much information the robot will gain by visiting the candidate independent of how the robot gets there. As we explain in Chapter 3, this fact means that we cannot really apply the approach to improving the efficiency of exploration that this thesis investigates to the case of exploration for decreasing map uncertainty.

Nevertheless, we briefly examine how methods for decreasing map uncertainty that use feature-based SLAM work in order to have a concrete example of exploring to decrease map uncertainty. We look at methods for decreasing map uncertainty that use feature-based SLAM because many of the methods for decreasing map uncertainty that have been developed are this kind.

2.4.1 Feature-based Methods

The goal of most feature-based methods is to decrease the overall map uncertainty as much as possible in the least amount of time possible [14] [9] 0. By map uncertainty we mean the feature-based SLAM map's uncertainty in its estimate of the position and orientation of the features it knows about and the robot. Feature-based SLAM's extended Kalman filter keeps track of this uncertainty in the state covariance matrix. Therefore, feature-based methods aim to decrease some overall measure calculated from this covariance matrix as much as possible and as quickly as possible. The overall measure that is calculated from the covariance matrix depends on the exploration method. Some possible measures are the determinant of the covariance matrix 0, the sum of determinants of individual feature and robot covariance matrices 0, or the trace of the

covariance matrix 0. Trying to decrease the determinant of the covariance matrix as much as possible corresponds to trying to decrease the entropy of the probability distribution of the state vector 0. The other measures calculated from the covariance matrix are also motivated by information theory.

Most methods of exploration for decreasing map uncertainty follow the exploration path planning framework. These methods attempt to find the sequence of actions for the robot take that will result in the largest drop in the overall map uncertainty, and output this sequence as the result of the exploration path planner. In order to find such a sequence of actions, these methods estimate how taking an action will cause the state covariance matrix to change. These methods perform this estimation by using the state update equations of the extended Kalman filter to find the expected value of the position of the robot after taking the action. The methods then use the sensor model and the estimated position of the features in the map to find the expected value of the measurements that the robot will make after taking the action. Finally, the methods use this expected value of the measurements to see how the state covariance matrix will update as a result of taking the action.

Most exploration methods use this ability to predict the result of taking an action to greedily choose the best next action for the robot to take [14]. More precisely, these methods discretize the set of actions the robot can take and then choose the action that should cause the overall measure of the uncertainty of the covariance matrix to decrease the most. This action is the output of the exploration path planner. Recently, however, Sim and Roy have developed a method to find a sequence of actions for the robot to take that is expected to decrease the overall map uncertainty by the maximum amount possible 0. Specifically, Sim and Roy's method places a grid over the feature map and determines the best path to each grid cell from the robot's initial position. A path is a sequence of grid cells, and so the possible actions that a robot can take are to move to one of the grid cells adjacent to the robot's current position. The best path to a grid cell is the path that results in the largest decrease in the overall map uncertainty. The method then looks at all of the best paths to all of the cells in the map and returns the path that results in the largest decrease in the overall map uncertainty. Sim and Roy's method is similar to the approach to exploration that this thesis investigates in it that plans such a globally

optimized path for the robot. However, their method is targeted towards exploration for decreasing map uncertainty, whereas the approach investigated by this thesis is targeted towards exploration for increasing map coverage. These two methods are the only methods we know of that plan globally optimized paths.

We now understand the characteristics of the general problem of exploration. We have seen that there are two major categories of exploration: exploration for increasing map coverage and exploration for decreasing map uncertainty. We also have a sense now of a number of different methods for performing these two types of exploration. The next chapter builds upon this knowledge and proposes a new way of computing a path from a set of candidates that should improve the efficiency of many of these methods.

3 The Finite Horizon Approach to Continuous Observation Planning

The main claim of this thesis is that performing observation planning over a finite horizon will, on average, improve the efficiency of many exploration methods for increasing map coverage (see Chapter 2). In this chapter we formally introduce the finite horizon approach to selecting candidates and predict those situations for which the approach should work well, based on its general properties. Then, in Chapter 6, we present the results of experiments comparing the finite horizon approach to observation planning with other approaches to observation planning using a specific candidate identification and scoring method (the Bosse, Newman, and Leonard method described in Chapter 2).

Section 3.1.1 defines the observation planning problem more precisely, and Section 3.1.2 gives our requirements for solutions to this problem. Section 3.2 motivates and presents informally the finite horizon approach to observation planning. Section 3.2.3.1 shows how we can formulate the finite horizon approach as solving the Selective Traveling Salesman Problem (S-TSP) and formally defines the S-TSP. Section 3.2.3.2 presents the finite horizon approach to continuous observation planning, and Section 3.2.3.3 defines this approach more formally. Finally, Section 3.3 provides a general analysis of how well we expect the finite horizon approach to continuous observation planning to perform across all exploration methods and all environments.

3.1 Observation Planning

All of the exploration methods that Chapter 2 describes have the robot solve the exploration path planning problem. Recall that an exploration path planner takes the robot's current position and a partially completed map as input and computes a path for the robot to execute that is meant to add information to the map. Furthermore, it is important to note that in all methods of exploration *for increasing map coverage* that we described in Chapter 2, the exploration path planner can be broken down into two components. The first component decides where to place candidate observation points,

and assigns to each of these candidates a utility that reflects how much information the robot would add to its current map, by making an observation at that point. The second component takes these candidate observation points and their utilities as input, and computes a path that will take the robot to some subset of the candidates. Each of the methods of exploration for increasing map coverage in Chapter 2 has a very different strategy for implementing the first component. We call these strategies *candidate identification and scoring strategies*. Similarly, we call strategies for implementing the second component *observation planning strategies*. The reason we break down exploration path planners into these two components is that it is a claim of this thesis that a particular observation planning strategy, the finite horizon continuous observation planning strategy, is ideal for many different candidate identification and scoring strategies, and many different environments.

Note that in the case of exploration for decreasing map uncertainty, we cannot break down exploration path planners into these two components. As we point out in Chapter 2, the problem is that, when exploring in order to decrease map uncertainty, the amount of information that the robot gains, by going to a given point and making an observation depends, on the path that the robot takes to get to that point. The reason for this dependency is that the robot's localization uncertainty depends strongly on the path that the robot executes, and the robot's localization uncertainty is part of the overall map uncertainty. Therefore, the amount that visiting an observation point decreases the overall map uncertainty can be thought of as the sum of the amount that the map uncertainty decreases from making the observation, and the amount that the map uncertainty increases, by driving to the observation point.

From the above reasoning, one might think that we could use the amount the map uncertainty decreases, by making an observation at a point, as the utility of that observation point. We would then use the amount that the map uncertainty increases, from going to the observation point, as the path cost in observation planning. The problem with this idea is that we cannot separate the amount that the map uncertainty decreases, from making an observation at a point, from the amount that the map uncertainty increases, over the path that the robot takes to get there. The amount that the map uncertainty decreases from making an observation at the point depends on the

localization uncertainty of the robot. In turn, the localization uncertainty of the robot depends on the path that the robot takes to the observation point. Therefore, there is no way to assign a utility to an observation point that does not depend on the path that the robot takes to the point, during the task of exploration for decreasing map uncertainty.

As a result, we will not consider what it means to perform observation planning for this type of exploration.

3.1.1 Definition of Observation Planning

For the sake of clarity, we will define our terms more precisely. In Chapter 2 we defined the problem of exploration path planning as follows. Given the robot's current pose, and a partial map that is being constantly updated, based on the robot sensor readings, the exploration path planning problem is to output a path that the robot can use to effectively control its motion, in order to improve the map. Recall that improving the map can mean increasing the coverage of the map, decreasing the pose uncertainty of the objects already in the map, or some combination of the two.

When improving the map means increasing map coverage, we break the exploration path planning problem down into two sub-problems. Given a partially completed map of the environment, *the candidate identification and scoring problem* is to produce a set of candidates, $\mathbf{C} = \{c_i\}$, for this partially completed map. Each candidate c_i has associated with it a position vector, $\mathbf{x}_i$, and a real-number utility, u_i. The candidate's position vector gives the location of the candidate in the map. The candidate's utility provides an estimate of how much information the robot will add to its map if it makes an observation from the candidate's location. Larger utilities correspond to larger amounts of information. Candidate identification and scoring strategies are strategies for solving the candidate identification and scoring problem.

The observation planning problem is to take a partially completed map, a set of candidates **C**, and a vector $\mathbf{x}_r$ giving the robot's current pose, and output a path for the robot to follow that will cause the robot to make observations at some subset of the set of candidates C. We call methods for solving the observation planning problem observation planning methods.

In Chapter 2 we gave the name *continuous exploration path planning* to approaches to controlling the robot that constantly re-solved the exploration path planning problem, in order to keep the path optimal with respect to changes in the map. When the robot constantly re-solves the exploration path planning problem, the robot must constantly re-solve the candidate identification and scoring problem and the observation planning problem. We therefore give constantly re-solving the candidate identification and scoring problem the name *continuous candidate identification and scoring*, and constantly re-solving the observation planning problem the name *continuous observation planning*.

3.1.2 Goals of Observation Planning Methods

Now that we have defined the two main components of most exploration path planning methods, candidate identification and scoring and observation planning, we can discuss what makes a good observation planning method. We focus on the observation planning problem because, while researchers have studied many different approaches to candidate identification and scoring, they have not put much effort into developing a good observation planning method.

The goals of an observation planning method are inherited from the goals of the exploration method. For example, if the goal of the exploration method is to expand the map in all directions as efficiently as possible, then the goal of the observation planning method is also to expand the map in all directions as efficiently as possible. Similarly, if the goal of exploration is to thoroughly map the local region before moving on to a new region, then this goal is the goal of the observation planning method.

Most exploration methods aim to expand map coverage in all directions as efficiently as possible. We can roughly evaluate how well an observation planning method achieves this goal by looking at the utilities of the candidates that the robot visits. We can focus on utilities because the utility of a candidate should estimate how much the robot will expand its map by visiting the candidate. By looking only at utilities, we make our evaluation of the efficiency of an observation planning method independent of the particular map representation the robot uses.

We will evaluate the efficiency of an observation planning method in a given exploration mission by looking at the total utility of the candidates that the robot reached before the cost of the robot's path exceeded a given maximum value. This maximum cost might be a maximum distance that the robot can travel, a maximum time, or a maximum amount of energy that the robot can expend. The higher the total utility of the candidates that the robot visited before reaching this maximum cost, the more efficient we consider the observation planning method to have been. Chapter 6 refines this concept of efficiency.

3.2 Finite Horizon Methods for Observation Planning

In this section we motivate and introduce the finite horizon observation planning method. In order to motivate the finite horizon method, we first describe the greedy observation planning method that existing exploration methods use. The greedy method selects a locally optimized path; therefore, the greedy method is suited for when planning a globally optimized path is futile, because the set of candidates changes constantly during exploration. We then introduce the full horizon observation planning method. The full horizon method selects a path that is globally optimal over the entire set of candidates; therefore, the full horizon method is suited for when the robot is certain that the set of candidates will not change as the robot explores. Finally we introduce the finite horizon observation planning method. The finite method plans a path that is optimal over a given finite distance; therefore, the finite horizon method is suited for when the set of candidates changes anywhere between frequently and never. In other words, the motivation for the finite horizon method is to cover the middle ground between the greedy and full horizon methods.

3.2.1 The Greedy Method for Observation Planning

Recall that, currently, virtually all exploration strategies that have the goal of exploring efficiently use a greedy continuous observation planning method. In a greedy continuous observation planning method, the robot constantly plans a path to visit one observation point. This path is a feasible least-cost path between the robot's current position and the observation point. The greedy method chooses this observation point by

finding the candidate that minimizes some function of the robot's current location and the candidate. This function can simply return the length of the least-cost path from the robot to the candidate, in which case the strategy amounts to the robot always going to the closest candidate to it [53]. The function can also somehow increase with increasing shortest path length and decrease as the candidate's utility increases [18] [10] in order to favor choosing candidates with a high utility. However, the way that the function combines distance and utility into a single score tends to be somewhat arbitrary.

Figure 3.1 shows the path planned by two different greedy functions at a certain point during exploration. In the figure, the circles are candidates, the two rectangles are obstacles, and the lines connecting the candidates represent possible paths between the candidates. The figure depicts the shortest path between a pair of candidates with a solid line. The figure also shows two sub-optimal paths between candidates as dotted lines. The robot should never choose travel along a sub-optimal path. In Figure 3.1a, the greedy function simply returns the distance between the robot and the given candidate. The function `position(`v_i`)` returns the location of the candidate v_i. Because candidate v_1 is the closest candidate to the robot, the chosen path in Figure 3.1a, denoted by the line with the arrow at the end, goes from the robot to v_1. The greedy function in Figure 3.1b is the negative of the function that Gonalez-Banos and Latombe's exploration method [18] uses, because we are looking for the candidate which *minimizes* the greedy function. The function `util(`v_i`)` returns the utility of the candidate v_i. Candidate v_3 minimizes the greedy function in Figure 3.1b, for $f(\mathbf{x}_r, v_3) = -0.8\exp(-10/100) = -0.72$, $f(\mathbf{x}_r, v_1) = -0.48$, and $f(\mathbf{x}_r, v_2) = -0.19$. Therefore the path in Figure 3.1b leads the robot to candidate v_3.

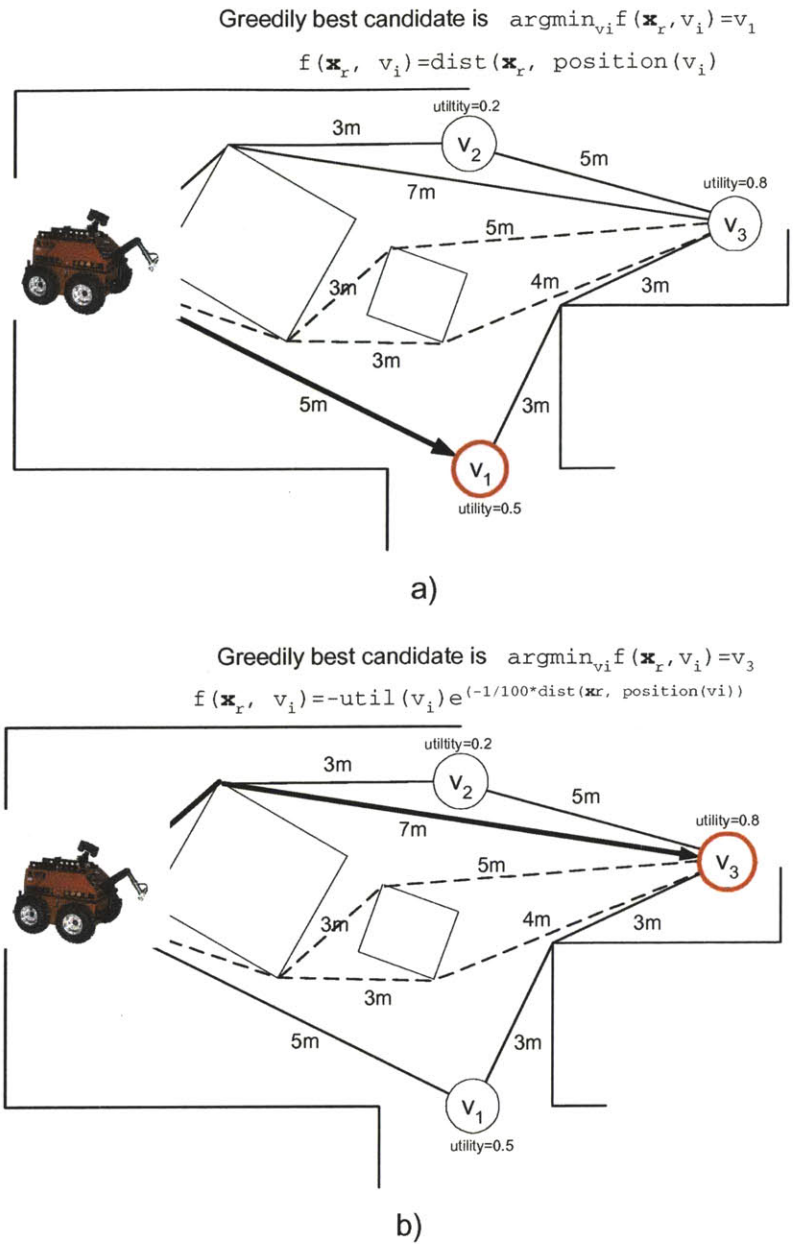


Figure 3.1 Possible Greedy Functions

Intuitively, the general problem with greedy approaches is that they do not consider how well the robot will be able to explore after it gets to the candidate it has chosen to visit next. The danger, therefore, is that the robot will go to the greedily-best next candidate, and then find itself forced to do something horribly inefficient to get to the greedily-best candidate after that. For example, Figure 3.2 depicts a situation in which, if the robot blindly continues to pick the next closest candidate to it, it will get

dragged down a long hallway that eventually dead-ends. The robot then is forced to back-track all the way back down this hallway in order to reach the next set of candidates. Figure 3.2a shows this greedy path, and Figure 3.2b shows a much more efficient path that goes down the hallway last. Candidates in the figures are the circles, and paths are depicted as a line with an arrow at the end.

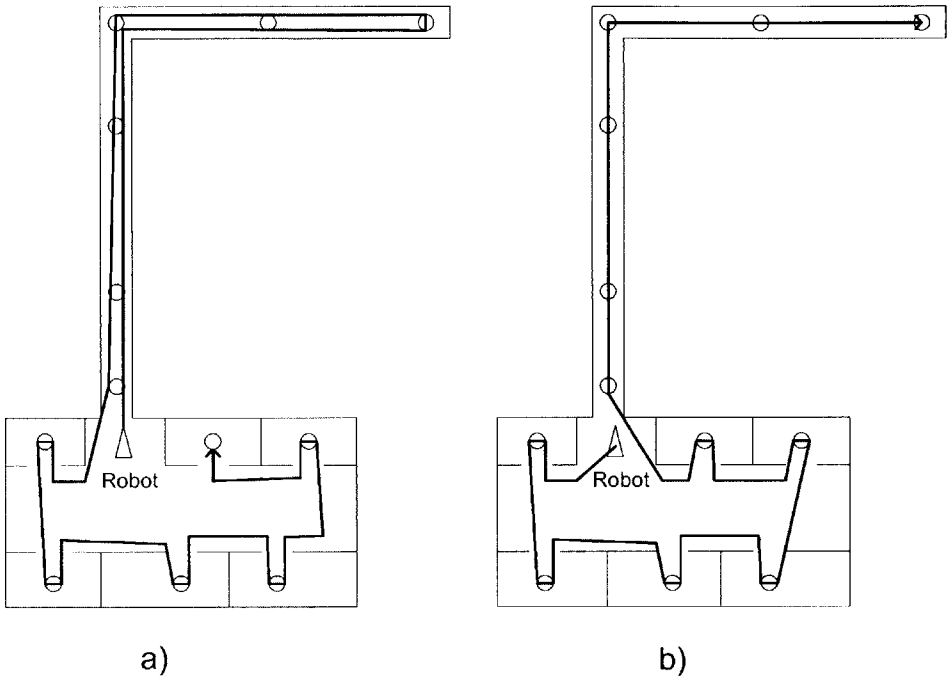


Figure 3.2 The Inefficiency of Greedy Paths

3.2.2 The Full Horizon Method for Observation Planning

The obvious solution to this problem with the greedy approach is to plan ahead. Given the current set of candidates, it is possible for the robot to find a minimum-cost path through the map that visits every candidate. We call the observation planning method that outputs a minimum-cost path visiting every candidate the full horizon observation planning method. We call the paths that the full horizon method outputs full horizon paths. Finding a full horizon path corresponds to solving the Traveling Salesman Problem (TSP) over the set of candidates. Recall that the Traveling Salesman Problem is to find a least-cost cycle in a graph that visits every vertex in the graph. Therefore, the only difference between a path that is a solution to the TSP over the set of candidates and a full horizon path is that a TSP solution path requires the robot to return to its start point,

while a full horizon path does not. Chapter 4 describes a method for altering any graph so that the solution to the TSP on the altered graph yields a full horizon path on the original graph. This mapping between the TSP and calculating full horizon paths is important because the TSP has been studied in depth, and many good algorithms for solving the TSP already exist [39].

By definition, a full horizon path must have a total cost that is less than or equal to the total cost of any other path visiting every candidate, including any path generated by the greedy approach. In other words, at the moment the full horizon method plans a path for the set of candidates, this path is globally optimal. Furthermore, if the robot executes this path in its entirety without the set of candidates ever changing (except for when candidates disappear once the robot reaches them), then the path that the robot *executes* using the full horizon method is also guaranteed to be globally optimal.

Unfortunately, as the robot explores an environment the set of candidates almost always changes. During the course of exploration, candidates appear, disappear, change location, and change utility. In order to immediately adjust the path to account for these changes, the robot could constantly recalculate the full horizon path over the candidates, thus performing continuous observation planning. However, if the set of candidates ever changes and the robot must recalculate the full horizon path, then there is no longer any guarantee that the path that the robot executes will be at least as efficient as the path the robot would have executed with a greedy method. In fact, no matter what observation planning method the robot uses to plan its paths, there are ways the set of candidates can change that make the path the robot actually executes very inefficient. Figure 3.3 shows an example of a case when the set of candidates changes in such a way so as to make the path that the robot executes with the full horizon observation planning method less efficient than the path the robot executes with the greedy method. Figure 3.3a depicts the robot's partially completed map midway through the robot's exploration of its environment. The circles in Figure 3.3a are candidates, the triangle is the robot, and the line connecting the candidates is a least-cost path that visits every candidate in the map. This least-cost path is what we refer to as the full horizon path through the candidates. Figure 3.3b depicts the greedy path for the same moment in the robot's exploration as in Figure 3.3a. The greedy path is significantly longer than the full horizon path shown in

Figure 3.3a. Figure 3.3c, however, shows what the environment actually looks like and what will happen if the robot tries to execute the full horizon path. The robot finds that there is a wall blocking its path to the first candidate in the full horizon path. This blockage has the same effect as if the first candidate in the path had suddenly become much farther away from the robot. The line connecting the candidates in Figure 3.3c is the new full horizon path for this situation. Note the path basically does what the greedy path in Figure 3.3b does. Therefore, the path the robot actually executes using the full horizon observation planning method is longer than the path the robot executes using the greedy observation planning method.

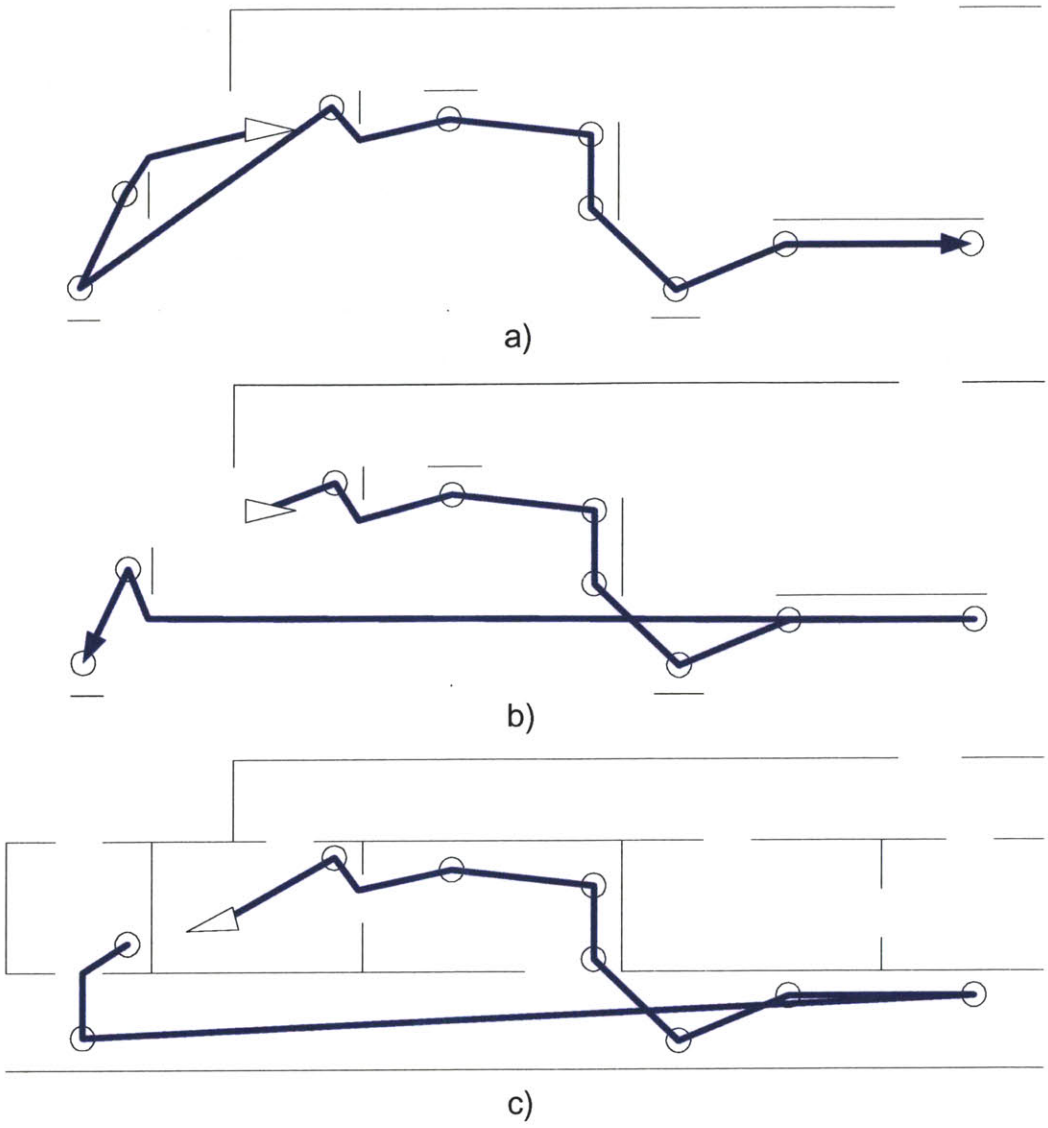


Figure 3.3 When the Full Horizon Path Performs Worse than Greedy: a) Initially calculated full horizon path. b) Initially calculated greedy path. c) Full horizon path recalculated part way through the robot's execution of the initial full horizon path.

The set of candidates can also change in such a way as to make the new path that the robot executes more efficient than it would have been. For example, Figure 3.4 shows a situation in which the change in the set of candidates allows a robot using the full horizon method to visit a subset of the candidates with a higher total utility before the mission ends, than the robot would have visited if the candidates had not changed. Figure 3.4a shows the initial full horizon path. There is 15m left in the robot's mission at this point; therefore, if the candidates did not change, the robot would be able to execute

its 14m full horizon path to completion before the end of its mission. Figure 3.4b shows the updated map after the robot gets to the first candidate (candidate a) and the full horizon path for this updated map. In this figure, the robot sees a room it has not seen before; therefore, the set of candidates changes to include four new candidates with high utilities. The robot will then visit all four of these new candidates (but not candidate b), before it travels the final 13m in its mission. As a result, instead of visiting a subset of candidates with a total utility of 0.4 in these 15m, the robot visits a subset of candidates with a total utility of 3.2 in 15m.

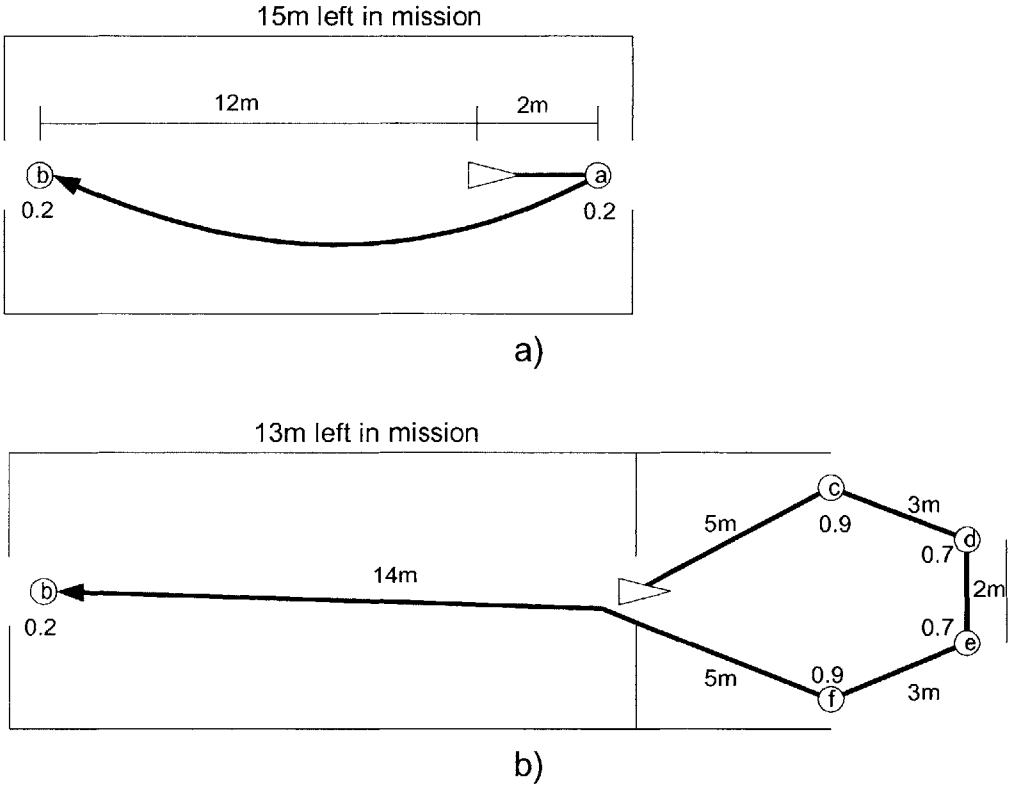


Figure 3.4 When a Change in the Set of Candidates Helps the Full Horizon Method

While the effect of candidates changing can be positive or negative, on average we do not expect a robot using the full horizon method to perform well when the set of candidates changes. To understand where this expectation comes from, first note that there are currently no good methods of predicting when or how the set of candidates will change during exploration. We do know, however, that the set of candidates usually changes because the robot sees new objects. And generally, the further the robot travels,

the more likely it is to see new objects that will cause the set of candidates to change. Therefore, no matter what exploration method the robot is using, we can model the way the candidates change as an arrival-type stochastic process (for example, a Poisson process)[3]. The arrivals in this model are instants when the set of candidates changes. And instead of using time as the measure of the length between arrivals, this model uses the distance that the robot travels.

We thus view an exploration trial as a sequence of intervals in which the set of candidates does not change and the robot executes its plan without disruption. Each interval is separated from the intervals before and after it by instants when the set of candidates does change. As we have seen, when the set of candidates changes, the effect that this change has on the path that the robot executes during the next interval is randomly good or bad. We say the effect is random because we assume that the robot cannot predict how the set of candidates will change and therefore what the effect of the change will be. As a result, the best that the robot can do is to try to plan an optimal path for each interval over which the candidates do not change. Yet the full horizon method does not plan an optimal path over each interval separately; the full horizon observation planning method plans a path that is optimal over all of the candidates in the map. This full horizon path is likely to be much longer than the length of the average interval. As a result, the robot will usually only get to execute part of a full horizon path, and there is no guarantee that this part of the path will be very efficient. However, as long as the robot is able to make it to one candidate before the set of candidates changes, a robot using the greedy method should at least execute locally efficient paths. Therefore, because a robot using the full horizon method is not guaranteed to do anything efficient during the intervals in which candidates remain constant, and because the result of the set of candidates changing does not consistently favor the full horizon over the greedy method, we do not expect the full horizon method to perform better than the greedy method on average when the set of candidates changes during exploration.

---

[3] One might object to modeling the way the set of candidates changes as a random process by saying that in fact we can be pretty certain that the set of candidates will change every time the robot visits a candidate. In exploration for increasing map coverage, the candidates are intentionally placed in areas that will cause the robot to map new area, and mapping new area usually causes the set of candidates to change. One response is that the set of candidates changes at other times as well, and these other times are much more difficult to predict. Another response is that the set of candidates does not always change when the robot visits a candidate. We discuss how the set of candidates changes in more detail in Section 3.3.2.

3.2.3 The Finite Horizon Method for Observation Planning

The finite horizon observation planning method is meant to plan the most efficient path possible when the robot cannot predict how or when the candidates will change. As we have seen, the best that the robot can do when the candidates change unpredictably is to plan a path that is optimally efficient *only over the distance of the average interval in which the candidates do not change.* Specifically, we would like to calculate a path that has a length that is less than or equal to a set threshold distance and that visits a subset of the candidates that has the maximum possible total utility. This threshold distance should be set to the length of the average distance that the robot can travel before the set of candidates changes[4]. Using a threshold distance to find a path in this manner is similar to using a finite horizon to evaluate the optimality of policies for Markov Decision Problems (MDP's) [27][5]. Therefore, we refer to this method of calculating paths as the finite horizon observation planning method.

Figure 3.5 shows two paths that the finite horizon observation planning method produces for the situation from Figure 3.1. In Figure 3.5a, the threshold cost (L) is 5m. The robot can only reach one candidate, candidate v_1, within 5m. Therefore, the best path with cost less than or equal to 5m is the path that goes straight to candidate v_1. The total utility of the candidates the robot visits along this path is 0.5. Note that this path is the same as the greedy path in Figure 3.1a. If we set the threshold cost very low, the finite horizon observation planning method is similar to the greedy method that always chooses the closest candidate.

In Figure 3.5b, the threshold cost is 11m; therefore, the finite horizon observation planning method must consider many possible paths. For example, the solution path from Figure 3.5a has a cost less than 11 and a total utility of 0.5. The shortest path that visits candidates v_1 and v_2, which we denote $<v_1, v_2>$, has a cost of 5m + 4m = 9m and a total utility of 0.7. The shortest path that visits candidates v_2 and v_3, $<v_2, v_3>$, has a cost

[4] We discuss what we mean by the "average distance the robot can travel before the set of candidates changes" in more detail in Section 3.2.3.2.

[5] Another common method of evaluating the optimality of policies for MDP's is to use a discounted reward over an infinite horizon [27]. We could use a discounted reward instead of an additive reward in the observation planning methods we develop in this thesis in order to take into account our uncertainty about when the set of candidates will change. We discuss this possibility and why we do not pursue it in this thesis in Section 3.2.3.2.

of 3m + 3m + 5m = 11m and a total utility of 1.0. Table 3.1 enumerates all of the least-cost paths that have a cost less than or equal to 11m and gives the cost and total utility of each of these paths. The path that visits the candidates with the highest total utility is the path $\langle v_1, v_3 \rangle$, and therefore this path is the output of the finite horizon observation planning method.

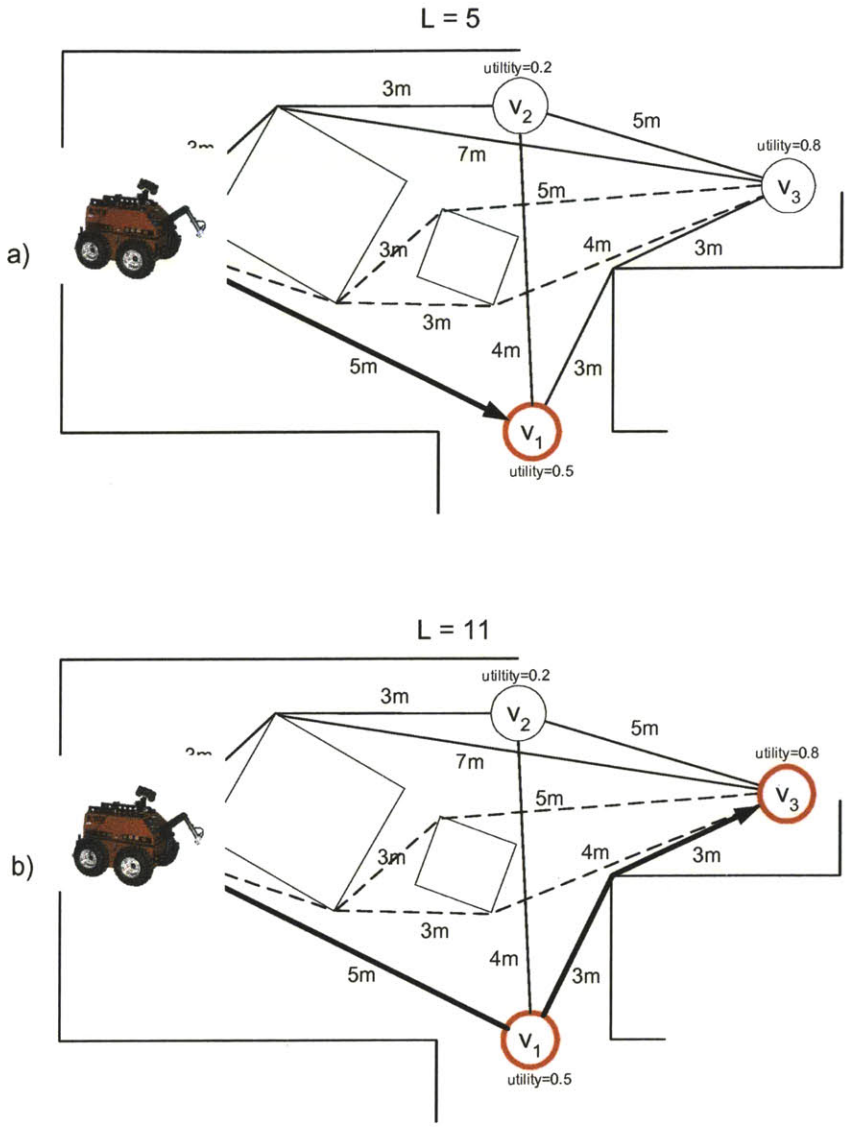


Figure 3.5 Finite Horizon Paths for L=5 and L=11

Path	Path Cost	Total Utility
$< v_1 >$	5m	0.5
$< v_2 >$	6m	0.2
$< v_3 >$	10m	0.8
$<v_1, v_2>$	9m	0.7
$<v_1, v_3>$	11m	1.3
$<v_2, v_3>$	11m	1.0

Table 3.1 Least-Cost Paths Between Candidates with Path Cost ≤ 11m

We can view the finite horizon observation planning method as a balance between greedy and full horizon observation planning methods. Therefore, our rough intuition is that if the set of candidates does not change at all during exploration (no candidates are added, removed, change score, or move), then the robot should use the full horizon observation planning method. If the set of candidates changes a medium amount during exploration, then the robot should use the finite horizon observation planning method. And if the set of candidates constantly changes, then it should not attempt to plan ahead at all, but use a greedy observation planning method. The reasoning behind this last rule is, that if the set of candidates changes a lot, even a path chosen with the finite horizon observation planning method can get caught, sacrificing in the short term for a reward that will never come to fruition. For instance, if the set of candidates changes drastically, every time the robot arrives at an observation point in its path, then if the robot ever goes to a point that is a sacrifice, the robot will probably not be able to capitalize on this sacrifice, because the candidates will change. Therefore, in this situation, we postulate that it is most efficient on average to always choose the greedily-best candidate.

One desirable feature of the finite horizon observation planning method is that it combines the goal of keeping path cost to a minimum with the goal of maximizing the total utility of the candidates that the robot visits in a principled manner. As we discussed above, the greedy method must use some arbitrary function in order to combine the utility of a candidate with the cost of the best path from the robot to the candidate. Researchers will probably choose this function empirically, and choosing the function based on past experience may mean that the function is tuned for the environments for

which the researchers ran tests and may not perform well in new environments. In addition, the full horizon approach to observation planning does not take into account the utilities of candidates at all. The approach finds a least-cost path that visits all of the candidates, regardless of their utilities.

A final nice feature of the finite horizon observation planning method is that we can adjust the planning horizon, L, to reflect how often we expect the set of candidates to change during exploration. In other words, if the robot starts off knowing a lot about its environment, perhaps because it can see a lot of its environment from its initial position, then the set of candidates should not drastically change very frequently during exploration and the robot can set its planning horizon to be very long. However, if the set of candidates is constantly changing, we can set the planning horizon to be very short so that the robot still has a chance of executing its path to completion. Recall that the only time we know for sure that the robot is going to do something efficient if it is able to execute its path to completion. The shorter we set the planning horizon, the closer the finite horizon method becomes to the greedy method. And if we set the planning horizon to be longer than the length of a full horizon path over the set of candidates, then the finite horizon method and the full horizon method are equivalent. Therefore, the finite horizon observation planning method is very adaptable.

Just as calculating a full horizon path corresponds to solving the TSP over the set of candidates, calculating a finite horizon path corresponds to solving the Selective Traveling Salesman Problem [15] over the set of candidates. We can state the S-TSP informally as follows: given a cost L and a directed graph with utilities for each vertex and costs for each edge, find an ordered subset of vertices, denoting a cycle, such that the total utility of the subset is maximized, and the cost along the cycle does not exceed L. The S-TSP has been studied in some depth, and efficient algorithms exist for solving it [15]. Therefore, if we show how to formulate the finite horizon approach to observation planning in terms of the S-TSP, it will be possible to make use of these efficient algorithms. In the next section, we examine the S-TSP and its relation to the finite horizon method in more detail.

3.2.3.1 Formal Definition of the Selective Traveling Salesman Problem

For the purposes of this thesis we use the following definition of the Selective Traveling Salesman Problem [15]. Take a complete directed graph G = (**V**, **E**), where **V** = $\{v_0, v_1, \ldots, v_{n-1}\}$ is the set of vertices, and **E** = $\{(v_0,v_1), (v_0, v_2)\ldots\}$ is the set of edges. Vertex v_0 is called the depot (the start vertex). The edge cost function $c(v_i, v_j)$ maps each edge to a real number. This cost function must satisfy the triangle inequality. In other words, the cost function must satisfy the following constraint: for all i, j, and k such that (v_i, v_j), (v_j, v_k), and (v_i, v_k) are members of **E**, $c(v_i, v_k) \leq c(v_i, v_j) + c(v_j, v_k)$. In addition, define a vertex utility function $u(v_i)$ mapping each vertex to a real number. Then given some real number L, the Selective Traveling Salesman Problem is to find a maximum utility Hamiltonian cycle over some sub-graph of G that includes v_0 and that has a total cost less than or equal to L. Recall that a path in a graph is a sequence of vertices $<u_0, u_1,.., u_k>$ such that for i = 0 to k, u_i is a vertex in the graph, and for i = 1 to k, (u_{i-1}, u_i) is a member of **E**. A cycle is a path that starts and ends with the same vertex and contains at least one edge. A Hamiltonian cycle is a cycle that includes the start vertex exactly twice and every other vertex in the graph exactly once. The utility of a cycle is defined to be the sum of the utilities of the vertices in the cycle.

It is important to note that even though the Selective Traveling Salesman Problem finds an optimum cycle through the graph, we do not actually want the robot to return to its start point. Therefore, what we want to find is a maximum utility Hamiltonian *path* through some sub-graph of G that starts at v_0 and that has a total cost less than or equal to L (a Hamiltonian path is a path that includes every vertex in the graph exactly once). Nevertheless, it is still important to formulate the fixed horizon method as solving the S-TSP. Many good algorithms for solving the S-TSP have been developed [16] [15] [17] [22] [50] [42], and we would like to be able to use these algorithms in implementations of finite horizon observation planners. Although we do not make use of existing S-TSP algorithms in this thesis, we still perform finite horizon observation planning by solving the S-TSP, in order to make it easy to swap in a more efficient S-TSP algorithm if we find one. In addition, we formulate the finite horizon method as solving the S-TSP even though we do not want to find cyclical paths because there are many very good algorithms for solving the TSP. In order to use these TSP solvers as black boxes in the

finite horizon algorithm we present in Chapter 4, we must be able to handle finding cycles instead of paths through our graphs.

Chapter 4 shows how to transform an undirected graph U into a directed graph D such that the solution to the S-TSP on D maps to a finite horizon path through U. This transformation is the reason why we define the S-TSP on directed graphs, for otherwise most real world graphs are undirected. On undirected graphs, the S-TSP is NP-hard [30]. Chapter 4 also explains a method for transforming a directed graph into an undirected graph by at most doubling the number of vertices. Therefore the S-TSP is also NP-hard for directed graphs. Furthermore, it turns out that for the graphs we are interested in, the transformation in Chapter 4 only adds one vertex to the undirected graph. Thus defining the S-TSP on directed as opposed to undirected graphs does not make the problem significantly harder for us.

The S-TSP is also known as the Orienteering Problem (OP) [16], for the goal in orienteering is to collect prizes at various locations on a map within a set amount of time. The S-TSP has been studied by a number of authors [16] [17] [22] [50] [42] [15] [30]. The most common approach to solving the problem is through branch-and-cut algorithms. In Chapter 4 we describe a novel method for solving the S-TSP by framing it as an optimal constraint satisfaction problem [50].

3.2.3.2 Finite Horizon Continuous Observation Planning

In order to immediately adjust the path for any changes in the set of candidates, we would like to perform continuous observation planning with the finite horizon method. However, if we continuously recalculate a finite horizon path for the set of candidates, then we need to decide how to set the horizon length, L, for each recalculation. For both non-continuous and continuous observation planning, the most obvious guideline for how to select L is as follows: whenever the robot plans a finite horizon path for the set of candidates, set the horizon length, L, to the expected distance that the robot will travel before the set of candidates changes. This guideline has different implications for different models of how the set of candidates changes.

A simple model is to say that the set of candidates changes at regular intervals during the robot's exploration, or in other words to say that the set of candidates changes

every time the robot travels exactly k meters, where k is a constant. Using this model, we find that, if the robot has traveled d meters since the last time the set of candidates changed, then we expect that the robot will only be able to travel $k - d$ meters before the set of candidates changes again. In order to perform continuous observation planning using this model, therefore, our guideline tells us to set L equal to $k - d$ meters, every time that the robot calculates a finite horizon path. We call this method of continuous observation planning the fixed horizon method.

In Section 3.2.2, however, we said that a more appropriate way to model when the set of candidates changes is as an arrival-type stochastic process. Using such a model, at any instant we can calculate the expected value of the distance that the robot will travel before the set of candidates next changes. This expected distance is the value our guideline tells us to set L to whenever we compute a finite horizon path. Unfortunately, the expected value of the distance that the robot will travel before the set of candidates changes usually depends on how the set of candidates have changed in the past (in other words, the process is not independent). For example, if recently the set of candidates has been changing frequently, one possible reason is that the robot is in an area for which it has mapped very little. In such an area, everywhere the robot turns it maps new objects, and these changes in the map cause the candidates to change. If recently the set of candidates has been changing frequently, therefore, then we have some reason to believe that the set of candidates will change frequently in the near future, because the robot may not leave the area for which it has mapped very little. Thus, in order to perform continuous observation planning, we would have to construct a filter to estimate the expected distance that the robot will travel before the set of candidates changes, based on studies of the patterns in how the candidates change.

In this thesis, we do not pursue this method of continuous observation planning. Instead, we see how far we can get by making the simplification that the expected value of the distance that the robot will travel before the set of candidates changes is independent of how the set of candidates has changed in the past (in other words, we assume the process is independent). We also assume that at each instant, the probability that the set of candidates will change is the same as the probability that the set of candidates will change at any other instant (in other words, we assume the process is

identically distributed). In this simplified model, the expected value of the distance that the robot will travel before the set of candidates changes is some constant p; it does not depend at all on how far the robot has traveled since the set of candidates last changed. Therefore, in order to perform continuous observation planning using this model, we always calculate the finite horizon path with L set to p. We call this method of continuous observation planning the receding horizon method.

It is worthwhile to note that modeling the way the candidates change as an arrival type stochastic process also suggests a method of continuous observation planning that this thesis does not pursue: using discounted rewards [27]. In other words, when calculating the total utility of a potential plan, we could decrease the utility of candidates that come later in the plan. The later a candidate occurs in the plan, the more we would decrease the utility of the candidate. We would then sum the modified utilities of the candidates that a plan visits and choose the plan that has the highest total utility over either a finite or full horizon. Decreasing the utilities of candidates in this way should cause the robot to favor visiting high utility candidates immediately, before the set of candidates changes, while still allowing the robot to take the future into account in its plan.

In MDP's, discounting is usually performed using a discount factor, γ, where $0 < \gamma < 1$. The discount factor can be thought of as the probability that the robot's "life" will last at least one more discrete step [27]. If the robot receives a reward of r_i at step i in a sequence of actions, then the total utility of this sequence is $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^n r_n$, where n is the length of the sequence. In other words, the reward at each step is multiplied by the probability that the robot will live to that step. In our case, we would like to multiply the utility of each candidate in a plan by the probability that the set of candidates will not change before the robot gets to that candidate. We can calculate this probability using the stochastic process that we use to model the way that the candidates change.

Using discounted utilities in calculating the total utility of a plan is a promising approach that should be investigated. However, if the utility of each candidate depends on the path that the robot takes to the candidate, then the problem of finding an optimal path over a finite or full horizon becomes much harder. Because none of the algorithms

that this thesis develops can be easily adapted to handle discounted utilities, we leave the investigation of discounting to future work. Therefore, when we refer to the fixed horizon, receding horizon, or full horizon methods, one may assume that these methods use simple, additive utilities.

Despite the fact that the fixed horizon method is justified by a simplistic model of the candidate dynamics, neither the fixed horizon method nor the receding horizon method is clearly better than the other. The main advantage that the fixed horizon method has over the receding horizon method is that, as long as the set of candidates does not change over the length of the horizon, a robot using the fixed horizon method is guaranteed to execute the path it planned to completion. As we saw with the full horizon method, we can only guarantee that the path the robot executes using a given observation planning method will be efficient if the robot is able to execute the path that it has planned to completion without the set of candidates changing enough to change the path. Therefore, if the set of candidates does not change over the horizon length, we can guarantee that a robot using the fixed horizon method will execute an efficient path. In the receding horizon method, however, the path that the robot plans constantly gets longer as the horizon recedes. Therefore, if the set of candidates *ever* changes enough to change the planned path, we cannot guarantee that a robot using the receding horizon path will do anything efficient.

For example, it is possible for a robot using the receding horizon method to constantly plan a path that makes sacrifices initially, and only reaps the reward of these sacrifices at the very end. In other words, it is possible for a robot using the receding horizon method to constantly put off doing something efficient until later. In this situation, if the set of candidates ever changes, then the robot may not get to capitalize on the sacrifices it has made. Figure 3.6 depicts an example of such a situation. In both diagrams, once the robot travels 30m the set of candidates changes enough to completely alter the robot's planned path. Therefore, we only look at the efficiency of the path that the robot executes in 30m. In Figure 3.6a, the robot explores using the receding horizon method with a 12m horizon. The initial path visits candidate a, then b, then c, since these are the only candidates within 12m of the robot. As the robot moves towards candidate a, it continues to recalculate its path over a 12m horizon, but the optimal route does not

change. As soon as it reaches candidate a, however, the robot finds that if it just sacrifices going to the relatively highly valued candidates b and c, it can reach the even more highly valued candidates e and f. Thus the robot travels the long distance to candidate d. When the robot reaches candidate d, however, it finds itself in the same situation as before. If the robot takes the immediately unattractive option of traveling a long distance to get to the low-utility candidate g, it can reach candidates h and i which have the highest utility in the map. Yet once the robot reaches candidate g, the set of candidates changes because the robot has traveled 30m. Therefore, the robot is not able to reap the reward of visiting candidates h and i, and the robot ends up visiting candidates whose total utility is 0.3 before the set of candidates changes.

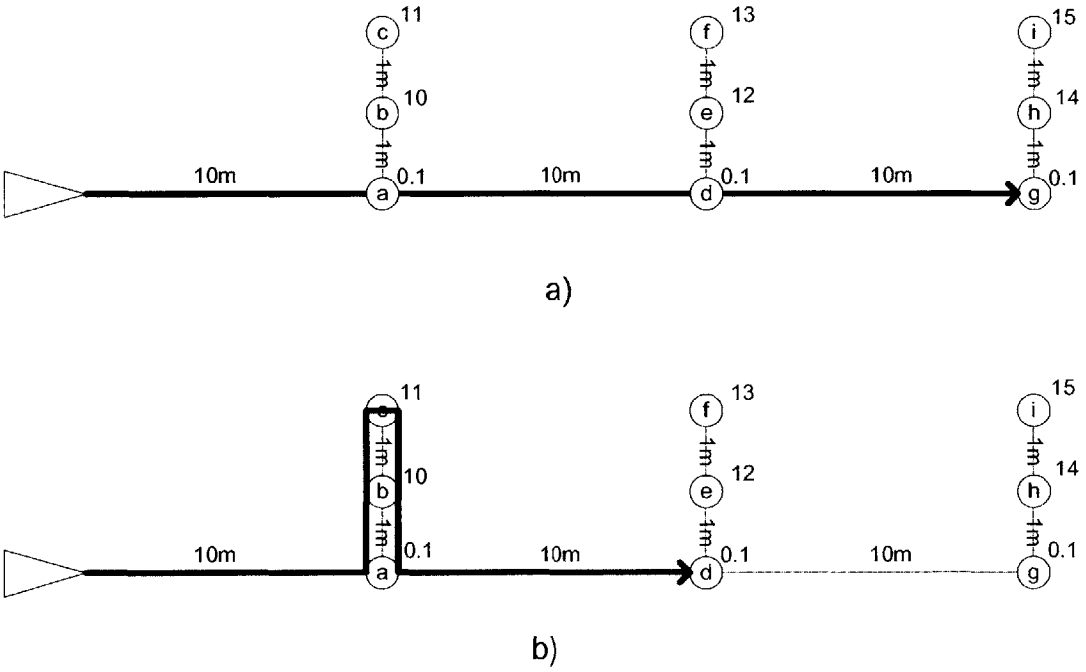


Figure 3.6 Receding Horizon (a) versus Fixed Horizon (b)

The problem with the receding horizon path in this case is that the robot constantly makes the sacrifice of traveling 10m to a candidate with a utility of only 0.1 and never gets to capitalize on this sacrifice by going to the highly valued points nearby. The fixed horizon method avoids this problem, since the robot gets to execute two of the

paths that it plans to completion[6]. Figure 3.6b depicts the same set of candidates, but with the robot using the fixed horizon method with a 12m horizon to select its route. Initially the path is the same, visiting candidates a, b and c. Once the robot reaches candidate a, however, it finds that it only has L–10m=2m left to plan for. Thus the best route still takes the robot to candidates b and c. Upon reaching candidate c, the robot starts planning for the next 12m path. The best path from c takes the robot back to candidates b and a, and then to candidate d. Once the robot reaches d, it has traveled a total distance of 24m. The robot then plans for a new 12m horizon and finds that the best path visits candidates g, h, and i. After the robot has traveled 6m towards candidate g, however, the set of candidates changes, because the robot has traveled a total of 30m.

Therefore, using the fixed horizon method, the robot visits candidates with a total utility of 21.2 before the set of candidates changes, while using the receding horizon approach, the robot visits candidates with a total utility of only 0.3. In addition, note that we could have simply lengthened this example by adding more groups of three candidates 10m apart and made the path that the robot executes using the fixed horizon approach an arbitrary amount better than the path that the robot executes using the receding horizon approach.

Nevertheless, the receding horizon method has the major advantage of always planning as far ahead into the future as we are willing to. In contrast, the fixed horizon method seems to unnecessarily become more and more near-sighted in its planning as the robot moves. Thus the danger in the fixed horizon case is that the robot will constantly ignore the fact that there is a very good set of candidates just beyond the fixed horizon that is within L meters of the robot's current position. For example, if we change the distance that the robot travels before the set of candidates changes to 22m in the previous example and get rid of candidates g, h, and i, then using the receding horizon approach suddenly becomes better than using the fixed horizon approach. In this altered example, the fixed horizon approach does not realize when the robot is at candidate a that candidates d, e, and f are better to go to than candidates b and c. Therefore, from candidate a, the robot goes to candidates b and c. Once the robot reaches candidate c, it

[6] Note that using discounted rewards is another way of avoiding situations like these in which the robot constantly sacrifices and never gets to reap any reward. However, investigating discounted rewards is outside of the scope of this thesis.

plans to go back to candidates b and a, and then to candidate d. However, the robot only makes it back to candidates b and a before the set of candidates changes. Therefore, the robot collects a total utility of 21.1 over the mission. In the receding horizon approach, however, the robot realizes at candidate a that visiting candidates d, e and f is better than visiting candidates b and c. Therefore, the robot visits candidates a, d, e, and f for a total utility of 25.2.

In summary, the major disadvantage of the receding horizon approach is that, unless the set of candidates never changes enough to alter the planned path, a robot using the receding horizon method is never *guaranteed* to execute an efficient path. The major disadvantage of the fixed horizon approach is that if the robot travels farther than the horizon length without the set of candidates changing, then the robot will usually plan less efficient paths using the fixed horizon approach that it will using the receding horizon approach. Therefore, neither approach is clearly better than the other. The only real way to decide on which method to use is to test and see how well each performs in the specific scenarios the user is interested in. In Chapter 6, we present the results of experiments testing the performance of a specific candidate identification and scoring method with both approaches.

One final point to make is that in the case of finite horizon continuous observation planning methods, there is one situation in which the set of candidates changing will probably not hurt the efficiency of the path that the robot executes. If no candidates ever disappear, move, or change their utility, then even if new candidates appear as the robot moves, a robot using either the fixed or receding horizon method will on average execute a path that is at least as efficient as it would have if nothing had changed at all. In order to see why the efficiency of the path that the robot executes should not decrease in the fixed horizon case, we look at the robot's exploration over one horizon. If new candidates are added to the map before the robot has reached the end of its horizon, then the path that the robot executes *over the interrupted horizon* is guaranteed to be at least as efficient as the path that the robot would have executed otherwise. The robot will only change the path it has planned if the new path is more efficient over the remaining distance to the horizon than the old path was. The new path would have to be more efficient because none of the old candidates changed; therefore, the robot could still

execute its old path if it was better. As long as the candidates do not change again, a robot using the fixed horizon method will execute to completion whatever path it decides on. Therefore, what we say about the path that the robot plans is also true about the path that the robot executes. However, if the new candidates caused the robot to change its path, then the robot will end up in a different location than it would have if new candidates had not appeared. This location might be much worse for exploration than the location the robot would have ended up in otherwise. Therefore, we cannot guarantee that the path the robot executes over the entire mission will be at least as efficient as the path the robot would have executed if no new candidates had appeared.

In order to see why the efficiency of the path that the robot executes should not decrease in the receding horizon case, we examine the path that the robot plans the instant after new candidates appear. This path will only be different from the path that the robot would have planned if the new path is more efficient than the old path. Yet because a robot using the receding horizon method constantly recalculates its path for a whole new horizon, even if the set of candidates does not change again, we cannot guarantee that the robot will execute to completion whichever path it decides on. However, if the set of candidates does not change again, the robot will usually execute a path that is at least as efficient as the path that the robot decided on after new candidates were added to the map. Therefore, even if new candidates are added to the map, a robot using the receding horizon method will on average execute a path that is at least as efficient as it would have if the set of candidates had not changed.

Yet if candidates disappear, move, or change their utility as the robot moves, then a robot using either the fixed horizon or the receding horizon may no longer be able to use the path that it initially planned. Therefore in this scenario, we have no reason to believe that a robot using either the fixed horizon or the receding horizon method will execute a path that is at least as efficient as the path it would have executed otherwise. And in many candidate identification and scoring methods, candidates are very likely to at least move and change score as the robot moves. Therefore, we usually will assume that we cannot predict whether the set of candidates changing will be good or bad for the efficiency of the path the robot executes.

Now that we understand the basic motivation behind the receding horizon and fixed horizon continuous observation planning methods, we are ready to define these methods more precisely.

3.2.3.3 Definition of Finite Horizon Observation Planning Methods

In order to define the receding horizon and fixed horizon continuous observation planning methods, we first define the basic finite horizon observation planning method. *The finite horizon observation planning method* takes as input a partially completed map of the environment, a set of candidates **C**, a vector giving the robot's current pose $\mathbf{x}_r$, and a real-valued threshold cost L. The method must then output a least-cost feasible path starting at the robot's current position and going to a subset of the candidates (the *chosen points*) such that the total utility of the subset of candidates is the maximum possible and the path cost is less than L[7].

We can break the finite horizon observation planning method down into solving two sub-problems. First, given a partially completed map, the set **C**, and the vector $\mathbf{x}_r$, the robot must find and store a least-cost path between each pair of candidates that avoids obstacles. Figure 3.7a depicts an example of a partially completed map. The candidates are labeled v_1-v_4, and the least-cost path between each pair of candidates is drawn with a solid line. The robot then must extract from these pair-wise paths a complete graph G of candidate observation points in which the edge cost between any two candidates is the least-cost path length between them. Figure 3.7b shows what the graph for Figure 3.7a would look like. Note that the robot's current location must also be included in the graph. Second, given the graph of candidates G, the candidate utilities, and a threshold cost L, the robot must solve the S-TSP. The output of the S-TSP solver will be the chosen points and a sequence to visit them in. The robot can then use its stored least-cost path for each consecutive pair of chosen points and execute the plan.

[7] If there are no candidates such that the least-cost path from the robot to the candidate has a cost less than L, then by this definition the finite horizon method outputs a path that has the robot not move anywhere. Yet in most implementations of the finite horizon method we do not want the robot to ever stop exploring if there are still candidates in the map. Therefore, in the implementation that we test in Chapter 6, the finite horizon method outputs the least-cost path to the greedily best candidate in this situation.

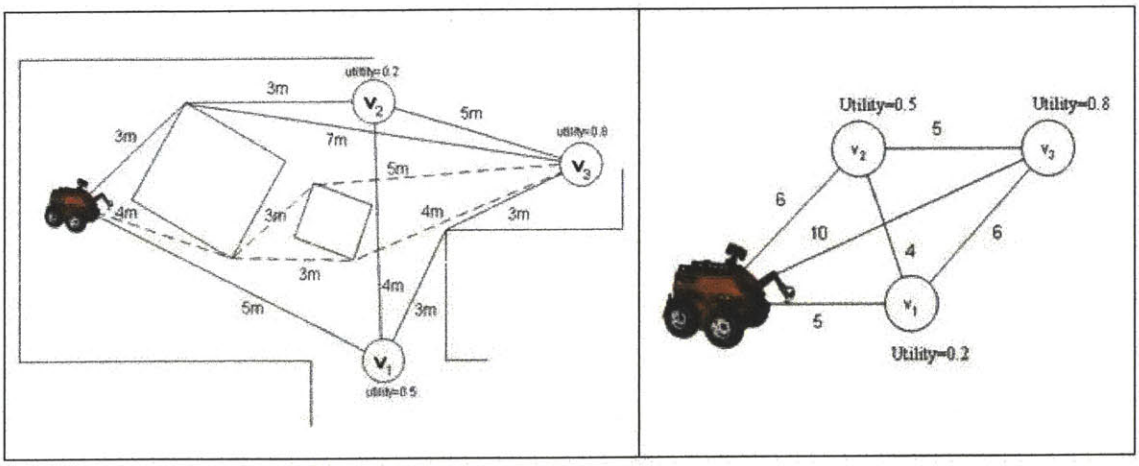


Figure 3.7 Extracting a Graph for the S-TSP

We refer to the first sub-problem as *the candidate graph extraction problem.* The second sub-problem is simply the Selective Traveling Salesman Problem. The benefit of dividing the problem into these two sub-problems is that the first sub-problem captures all of the interesting variations that a particular situation might require, while the second sub-problem captures the underlying combinatorial optimization problem. In the candidate graph extraction problem, the robot is free to use any map representation it wants (e.g. grid-based, feature-based, scan-matched) as long as there is some method for finding a feasible least-cost path between candidate pairs. In addition, the cost of a path does not have to be its distance, it could also be the time or even the energy required to traverse the path. No matter what options we choose in solving the candidate graph extraction problem, however, the input and output of the S-TSP remains the same.

Figure 3.8 gives pseudo-code for performing finite horizon observation planning by breaking the problem down into these two sub-problems. In particular, the function `Extract_Candidate_Graph()` on line 1 solves the candidate graph extraction problem, and the function `Solve_S-TSP()` on line 2 solves the S-TSP. The function `Fill_In_Path()` on line 3 takes an ordered subset of the set of candidates, the partially completed map, and the robot's position, and returns a shortest path avoiding all obstacles that takes the robot from its position to each of the candidates in the subset in order.

```
Plan_Observations_Finite_Horizon(partially completed map, candidate
set, robot pose, threshold cost)
returns exploration path starting at robot pose

1.  let graph = Extract_Candidate_Graph(partially completed map,
candidate set, robot pose)
2.  let candidate list = Solve_S-TSP(graph, threshold cost)
3.  let path = Fill_In_Path(candidate list, partially completed map,
robot pose)
4.  return path
```

Figure 3.8 Pseudo-code for the Finite Horizon Observation Planning Method

Reviewing the big picture of our definitions, most exploration methods perform exploration path planning. We break the exploration path planning problem down into two sub-problems: the candidate identification and scoring problem and the observation planning problem. A specific approach to solving the observation planning problem is the finite horizon observation planning method. We break the finite horizon observation planning method into two more sub-problems: the candidate graph extraction problem and the S-TSP. In Chapters 4, and 5, we describe a specific implementation of an exploration method using the finite horizon observation planning method. The implementation uses the candidate identification and scoring approach of the Newman, Bosse, and Leonard exploration method that we described in Chapter 2. In Chapter 4 we present a novel method for solving the S-TSP. In Chapter 5 we describe the implementation's candidate graph extraction method. Finally, in Chapter 6 we present the results of testing the performance of this implementation in real world and simulated environments.

Now that we have defined the basic finite horizon observation planning method, we can build upon this definition to define the finite horizon continuous observation planning methods. In *the receding horizon continuous observation planning method* the robot constantly recalculates the path using the finite horizon observation planning method with the most recently updated partial map, the most recently computed set of candidates **C**, the robot's current pose vector $\mathbf{x}_r$, and the threshold cost L. In *the fixed horizon continuous observation planning method* the robot constantly recalculates the path using the finite horizon observation planning method with the most recently updated

partial map, the most recently computed set of candidates **C**, the robot's current pose vector $\mathbf{x}_r$, and the threshold cost L minus d, where d is the cost of the path the robot has executed since the last horizon. Initially d is set to zero. Whenever L-d is less than or equal to zero, d is reset to zero and we say that the method has started a new horizon[8].

Figure 3.9 shows pseudo-code for a generic exploration method using the receding and fixed horizon continuous observation planning methods. Figure 3.9a shows the top level exploration pseudo-code. This pseudo-code is almost identical to the exploration pseudo-code depicted in Figure 2.2. The main difference is that the user must pass in the planning horizon (threshold cost) as a parameter to the exploration method. Figure 3.9b shows an implementation of the `Plan_Exploration_Path()` function that performs receding horizon continuous observation path planning. The function `Identify_and_Score_Candidates()` on line 1 solves the candidate identification and scoring problem. We gave the pseudo-code for the function `Plan_Observations_Finite_Horizon` in Figure 3.8. Figure 3.9c shows an implementation of the `Plan_Exploration_Path()` function that performs fixed horizon continuous observation path planning.

[8] When L-d gets to be small, it becomes likely that there are no candidates with a least-cost path between the robot and the candidate that is less than L-d. Therefore the robot often gets stuck without a path to execute in the fixed horizon method. In the implementation of the fixed horizon method that we test in Chapter 6, we fix this problem by returning the greedily-best path in this situation.

```
Explore_Continuous_Finite_Horizon(threshold cost, constantly updating map)
returns nothing

1.  while Mission_Completed() is false
2.     let map = Get_Most_Recent_Map(constantly updating map)
3.     let robot pose = Get_Current_Robot_Pose(map)
4.     let path = Plan_Exploration_Path(map, robot pose, threshold cost)
5.     Execute_Segment_of_Path(path)
6.  endwhile
```

a)

```
Plan_Exploration_Path(current robot pose, partially completed map,
threshold cost)
returns exploration path starting at current robot pose

1.  let C = Identify_and_Score_Candidates(partially completed map)
2.  let path = Plan_Observations_Finite_Horizon(partially completed
map, C, current robot pose, threshold cost)
3.  return path
```

b)

```
Plan_Exploration_Path(current robot pose, partially completed map,
threshold cost)
returns exploration path starting at current robot pose

1.  let C = Identify_and_Score_Candidates(partially completed map)
2.  let d = Get_Total_Executed_Path_Cost()
3.  let new threshold = threshold cost - d
4.  let path = Plan_Observations_Finite_Horizon(partially
completed map, C, current robot pose, new threshold)
5.  return path
```

c)

Figure 3.9 Pseudo-code for Receding and Fixed Horizon Approaches to Exploration

3.3 General Analysis of Finite Horizon Observation Planning Methods

Even though we only had time in this thesis to test how well finite horizon observation planning methods work with one particular candidate identification and scoring algorithm, we can still gain some understanding of how well finite horizon methods should work for most other candidate identification and scoring algorithms through general analysis. In order to evaluate the performance of finite horizon observation planning methods, we look at how well finite horizon methods work relative to greedy and full horizon methods. In particular we look at the continuous observation planning version of each of these methods, for when the set of candidates changes, non-

continuous observation planning only performs well out of luck. We focus on exploration in order to expand the map in all directions as efficiently as possible because this is the most common sub-goal of methods of exploration for increasing map coverage. We discuss the possible sub-goals of methods of exploration for increasing map coverage in Chapter 2.

3.3.1 Strengths and Weaknesses of Observation Planning Methods

Section 3.2 gave us the basic intuition that greedy observation planning approaches should perform most efficiently when the set of candidates changes frequently during exploration, finite horizon approaches should perform most efficiently when the set of candidates changes moderately often, and the full horizon approach should perform most efficiently when the set of candidates barely ever changes at all. We now refine this intuition.

In Section 3.2 we saw that the general motivation for considering the full horizon method was that planning ahead should allow the robot to compute more efficient paths than the greedy method. More precisely, the efficiency of the full horizon path for a set of candidates is guaranteed to be greater than or equal to the efficiency of any other path that visits every candidate in the set. Therefore, if the robot explores until it has visited every candidate in the map, and if the set of candidates does not change during exploration, then the efficiency of the path that the robot executes using the full horizon method is guaranteed to be greater than or equal to the efficiency of the path the robot executes using a greedy method or any other method. The reason that we did not then conclude that the full horizon method is the final word in observation planning was that, as the robot executes its planned path, it is likely that the set of candidates will change enough to cause the robot to significantly alter its path if it is performing continuous observation planning. If the robot is not able to execute the entire path that the full horizon method has planned, then there is no guarantee that the robot will do anything efficient at all. The first portion of a full horizon path is not guaranteed to be efficient; only the whole path is planned to be efficient.

Our solution to this problem with the full horizon observation planning method was to say that, if major changes in the set of candidates occur at somewhat regular

intervals, then the robot can use the finite horizon approach to plan a path that is optimal only over the period of time that we expect the candidates to stay constant. Because the robot should be able to execute these finite horizon paths completely, we expect that the robot will explore more efficiently on average using the finite horizon approach, than it would using a greedy approach. We can only say that we expect the exploration to be more efficient *on average* because a finite horizon path is only calculated to be optimally efficient over the planning horizon. If the robot's mission lasts longer than one planning horizon, then the robot must string multiple finite horizon paths together. These concatenated finite horizon paths are not guaranteed to be optimally efficient over their combined length. Therefore, it is possible that a robot using the greedy approach could get lucky and execute a path that is more efficient over the length of the mission than these concatenated finite horizon paths. Yet because the efficiency of a finite horizon path is guaranteed to be greater than or equal to the efficiency of any other path (including greedy paths) over the length of the horizon for the same set of candidates and initial robot position, it is more likely that the concatenated finite horizon paths will be more efficient over the length of the mission than a greedy path.

Unfortunately, we cannot predict when the set of candidates will change enough to alter the path that the robot has planned. As a result, we cannot choose horizon lengths for the robot to plan its paths over that will guarantee that the robot will be able to execute all of the paths it plans to completion before the set of candidates changes. And if the execution of a path that the robot has planned is interrupted by the set of candidates changing, we cannot guarantee that the part of the path the robot executes before the interruption will be efficient. Therefore, it is important to determine how often the robot not being able to execute some of its finite horizon paths to completion will significantly hurt the efficiency of the robot's exploration. The finite horizon paths that the robot *can* execute to completion should be more efficient than corresponding greedy paths of the same length. Therefore, in order to determine how efficiently a robot using the finite horizon approach should explore on average (as compared to the greedy approach), we must weigh how much the finite horizon paths that the robot completes should improve efficiency against how much the finite horizon paths that get interrupted should hurt efficiency.

Having the execution of an finite horizon path interrupted will hurt the efficiency of the robot's exploration if the part of the finite horizon path that a robot executes before the path changes is significantly worse than the path the robot would have executed using a greedy method. In other words, if the finite horizon path makes big sacrifices early on and the execution of the path gets interrupted before the robot is able to reap the reward of these sacrifices, then getting interrupted will hurt efficiency. Figure 3.10 shows an example of the robot getting caught making a sacrifice in this way. In Figure 3.10a, the robot passes up visiting the nearby candidates a and b and makes the sacrifice of going all the way to candidate c in order to next reap the reward of visiting candidates d through i. Figure 3.10b shows what the environment actually looks like. Once the robot reaches candidate c, it finds that a wall it had not seen before blocks the rest of the path it had planned. If the robot had followed a greedy strategy by initially going to candidates a and b, it would have succeeded in visiting candidates with a higher total utility in a shorter distance. Therefore in this situation, the greedy method performs more efficiently than the finite horizon method. In addition, had the robot gone to candidates a and b it would have discovered an entirely new room off to the right with many high utility candidates. Using the greedy method, the robot would have been drawn off into this new room and may have never had to visit the distant candidate c.

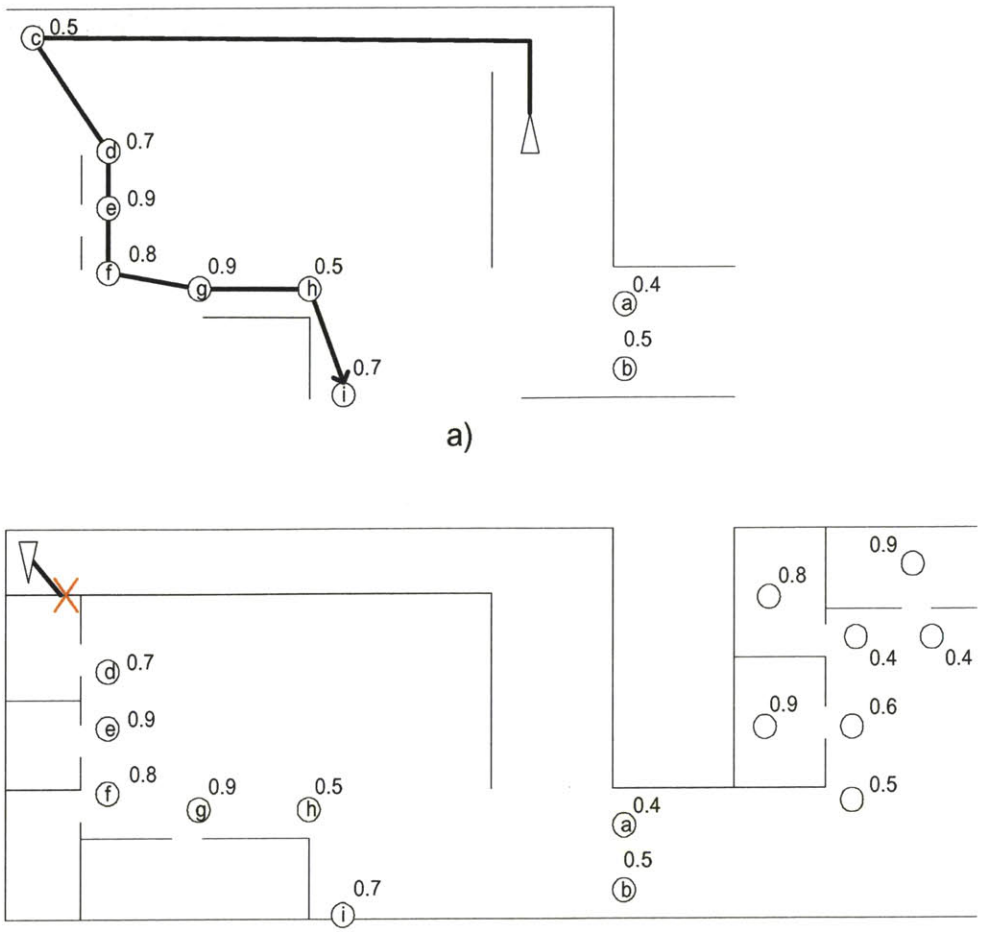


Figure 3.10 Getting Interrupted after Making a Sacrifice

Of course, there is no reason why the part of a finite horizon path that the robot executes before being interrupted must be less efficient than a corresponding greedy path. If the robot did not make any big unrewarded sacrifices in this part of the path, then the efficiency of this part of the path could have been greater than or equal to the efficiency of a corresponding greedy path. In the end, therefore, we cannot say on face whether the finite horizon approach will be more or less efficient than the greedy and full horizon approaches. The efficiency of the path the robot executes using the finite horizon method strongly depends upon the arrangement of the candidate observation points and the way they change. The arrangement of the candidates and the way they change in turn depends on the shape of the environment and the exploration and SLAM algorithms the robot is

using. Some questions we need to answer in order evaluate the relative efficiency of the finite horizon approach for a particular situation are:

- Does the set of candidates ever change enough to cause the robot to drastically alter its planned path?
- Do these changes occur regularly enough for the robot to be able to minimize the number of times its finite horizon path gets interrupted by choosing a proper horizon length?
- How often does the robot make big sacrifices, and how often does it get caught not having collected the reward for these sacrifices?
- When the robot is able to execute its finite horizon path to completion, how much more efficient is this path than a corresponding greedy path?

We now examine what we can say about the answers to these questions for the general case of exploration to increase map coverage in all directions.

3.3.2 Analysis of the Finite Horizon Approach in Exploration to Increase Map Coverage

The major problem with using the finite horizon approach when exploring to increase map coverage is that the robot's map, and, therefore, the set of candidates, changes substantially every time the robot visits an unexplored region. It might seem then that the robot needs to be able to predict what it will see at these unexplored regions in order for it to be able to effectively plan ahead. Nevertheless, no current exploration method can make such predictions. Therefore, we examine how well the finite horizon approach should perform without the robot making predictions about what it will see in unexplored regions. Our claim is that as long as the robot's planned paths getting interrupted does not significantly hurt the efficiency of the exploration, then if the robot is able to execute any of its finite horizon paths to completion, on average the finite horizon method should perform better than all other methods.

In order to estimate how frequently the robot's execution of its finite horizon path will get interrupted and what effect this interruption will have on the efficiency of exploration, we must understand how the set of candidates will change during exploration for increasing map coverage. Unfortunately, the dynamics of the set of candidates is

noticeably different for different methods of exploration for increasing map coverage, as a result of the wide variety of these methods. These differences limit the general analysis that we can perform; therefore it is important to perform experiments to truly evaluate how efficient finite horizon methods are. Nevertheless, we can find many significant similarities between the ways candidates change in most methods of exploration for increasing map coverage. In particular, mapping previously unexplored areas, the utility of candidates changing as the robot passes by them, and small adjustments by the SLAM algorithm to the location of objects already in the map cause the set of candidates to change in most exploration methods. We examine all but the last cause, for the changes to the set of candidates that result from SLAM updates to objects in the map are usually minor.

3.3.2.1 Mapping Previously Unexplored Areas

The source of change in the set of candidates that we must worry about most is mapping previously unexplored areas. If mapping previously unexplored areas only added new candidates to the set of candidates, then as we saw in Section 3.2.3.2, we still would expect a robot using a finite horizon continuous observation planning method to execute an efficient path on average. When a robot visits an unseen area, however, it pushes back the frontier of the part of the environment that it has seen. Most exploration methods place candidates along this frontier; therefore, when the frontier moves so do the candidates. In addition, a robot usually maps new objects when it visits previously unexplored areas. These new objects can block the robot's path to an existing candidate and thereby increase the cost of the least-cost path to the candidate. Unfortunately, the entire point of exploring to increase map coverage is to map previously unexplored areas. In fact, if the candidate identification and scoring algorithm is doing its job, then every time the robot visits a candidate it should see a previously unexplored area. It is a problem for the finite horizon observation planning method if the set of candidates changes significantly each time the robot reaches a candidate. If the robot's execution of its planned path is interrupted at every candidate, then there cannot be any benefit to planning ahead farther than the next candidate, and the robot should use a greedy method.

Planning ahead with the finite horizon approach is not doomed, however, as long as there are cases when the set of candidates does not change drastically at every candidate the robot visits. In these cases, the robot should still be able to execute a finite horizon path with an appropriate horizon length to completion and be more efficient than a greedy method would be for this one horizon. Figure 3.11 depicts one possible situation in which the set of candidates does not change at all as the robot explores. In the figure, the triangle is the robot and the circles are candidates. The solid lines in the figure are lines in the robot's map, and the dotted lines correspond to lines in the real world which are not yet in the robot's map. Figure 3.11a depicts the initial path and map of the robot. In Figure 3.11b, the robot has reached candidate b and mapped the wall there. The robot does not push back the frontier of the explored region at all, however, because the new wall blocks the view beyond the candidate. Therefore, the robot does not add any new candidate to its map and continues to execute its original path. The set of candidates does not change at candidates c, d, or a either for the same reason. Figure 3.11c shows the robot at candidate e. The robot pushes back the frontier a short distance, yet the newly explored region (the small office in the figure) is enclosed entirely by walls. The robot cannot see through any of these walls to push this frontier out further, and therefore the robot does not add any new candidates at point e either. Over the entire path depicted in Figure 3.11, therefore, the set of candidates never changes.

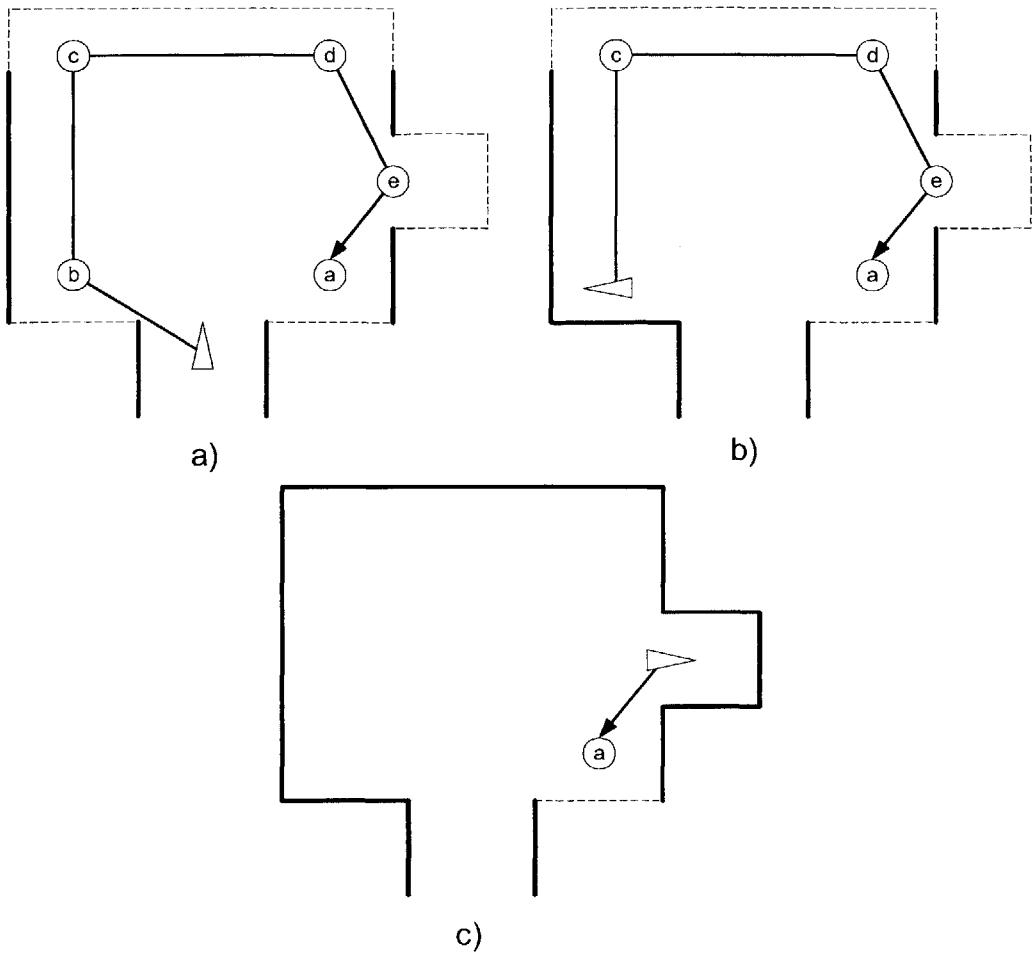


Figure 3.11 Exploration Without Changing the Set of Candidates

There are also cases when the set of candidates changes over the robot's path, but not enough to alter that path significantly. One such case is when the robot has deduced the basic structure of the environment with its map, but small gaps in the map still need to be filled in. The finite horizon approach should calculate an efficient path that takes the robot between these gaps in the map. What the robot sees at each gap may add a few more candidates for the robot to visit in order to fully fill in the gap, but relative to the length of the whole path, the distance to these new candidates will be insignificant. As a result, the order that the path visits the gaps in will not change and the large-scale structure of the finite horizon path will remain the same. There are a number of situations in which the robot's map might capture the basic structure of the environment but still have gaps that need to be filled in. In Mars exploration, satellite images can give

a rover a rough sense of the structure of the environment it is in, however, these images are not very high resolution. Therefore, finite horizon approaches might be ideal for filling in maps of Mars at resolutions useful to a rover.

Another case in which the robot could start out knowing the overall structure of its environment is if the environment is open enough for the robot to see most of the large-scale features from its initial position. For example, outdoor environments are usually very open in this manner. Large scale features might correspond to spread-out clusters of rocks, trees, or other objects. The robot would see some of the objects in these clusters and therefore place candidates at each cluster in order to fill in the details there. If the horizon that the robot plans over is long enough for the finite horizon path to go to multiple clusters, then candidates appearing, disappearing, and moving within the clusters should not significantly change the path the robot plans. The finite horizon path might only change the order in which the robot visits the clusters if whole clusters move or if all candidates within a cluster drastically decrease or increase in utility. Because there are no walls to block paths between clusters outdoors, it is unlikely that the shortest distance between clusters will change much in these environments. In cases such as these, when visiting candidates changes the set of candidates a relatively small amount, the robot should be able to execute a finite horizon path to completion without the efficient large-scale structure of the path changing.

One concern is that if the set of candidates never changes enough to cause the robot to significantly alter its path, then we might think that the full horizon approach should perform better than the finite horizon approach. The full horizon approach plans a globally optimal path, while the finite horizon approach only plans a path that is optimal over the length of the horizon. One observation is that, as we mentioned in Section 3.2, if we set the S-TSP horizon to be greater than or equal to the cost of a TSP path over the candidates, then the S-TSP is equivalent to the TSP. Thus we can always mimic the full horizon approach to selecting waypoints with a finite horizon approach, and as a result the finite horizon approach can work well even when the set of candidates does not change enough to significantly alter the robot's path.

Another observation is that even if the set of candidates never changes drastically enough to cause the robot to significantly alter its planned path, it is possible that the

combined effect of many small changes to the set of candidates will be enough to alter the robot's path. The longer the path the robot plans, the more the robot exposes itself to this possibility. Therefore, even if the set of candidates never changes drastically, it still may not be a good idea to plan a path to visit every candidate in the map by using the full horizon approach.

Finally, one important difference between the full horizon and finite horizon approaches to selecting waypoints is that the full horizon approach does not consider the utilities of the candidates at all in calculating a path. As a result, the full horizon approach does not prioritize visiting candidates with a high utility. On the other hand, the finite horizon approach chooses a subset of candidates to visit that has maximal total utility. Therefore, if the exploration mission ends before the robot has visited every candidate (as is often the case), it is just as likely that that robot will have visited low utility candidates as it is that the robot will have visited high utility candidates using the full horizon approach. Using the finite horizon approach, however, the robot is likely to visit candidates with a high total utility as long as it has gotten to execute at least one full finite horizon path before its mission ends. In addition, even if we ignore the utilities of the candidates, if the mission ends before the robot visits every candidate, then the part of the full horizon path that the robot gets to execute is not guaranteed to be efficient in terms of length either. Therefore, if the effect of small changes to the set of candidates accumulates as the robot explores, or if the mission ends before the robot gets to visit every candidate, then it is better to use the finite horizon approach and plan over a limited horizon than to use the full horizon approach.

3.3.2.2 Candidate Interactions

The second common cause of the set of candidates changing is candidate interactions. *Candidate interactions* happen when the robot's path to one candidate goes by other candidates, causing the utilities of those other candidates to change. In exploration for increasing map coverage, candidate interactions usually occur when the robot arrives at one candidate and maps part of a region covered by other candidates. Figure 3.12 gives an example candidate interactions for the case of the Gonzalez-Banos and Latombe exploration algorithm described in Chapter 2. Figure 3.12a shows what the

environment actually looks like, and Figure 3.12b shows the map that the robot starts off with. The semicircle near each candidate shows the predicted amount of area that the robot's sensor will see from the candidate. The fact that these semicircles overlap means that it is possible that the robot will see from one candidate part of the region covered by another candidate. Figure 3.12c shows what happens if the robot visits candidate b. The office that the robot sees from candidate b cuts off part of the area that the robot could see from candidates a or c. Therefore, going to candidate b causes the utilities of candidates a and c to drop. Candidate interactions also occur in the Newman, Bosse, and Leonard exploration algorithm and the grid-based approaches to exploration described in Chapter 2.

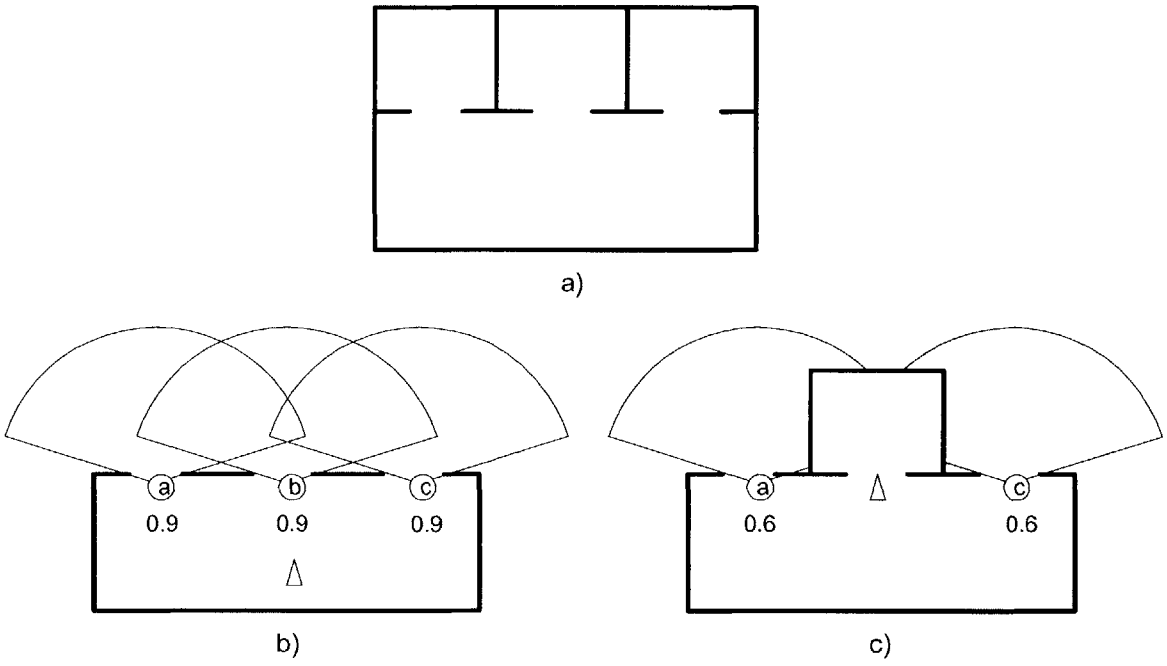


Figure 3.12 Candidate Interactions

Changes in the set of candidates caused by candidate interactions are not usually as drastic as changes caused by mapping new regions. In many environments, candidate interactions can occur frequently, however. Indoor environments that are densely populated with objects will cause candidates to interact frequently because the candidates will be close together and highly overlapping. In such environments we therefore must worry about the robot's finite horizon path getting interrupted because of candidate

interactions, as well as new regions being mapped. So once again we see that the finite horizon approach is suited for open environments.

When the regions visible from various candidates overlap with each other, this overlap causes another problem besides candidate interactions: finite horizon paths will favor visiting regions that the robot has explored moderately well over regions the robot has barely explored at all. When there are many candidates close together, it is usually because they are marking the small gaps in the map in an area that has been explored moderately well. Even though the regions visible from these candidates probably overlap, the utility of each candidate will be as large as it would be if there were no other candidates nearby in most exploration approaches. A finite horizon path will sacrifice to get to such a cluster of candidates because the candidates are close together and have a higher total value than the total amount of new area visible from them warrants. In contrast, largely unexplored regions usually only have a few frontiers or features to place candidates near. Therefore there will be relatively few candidates in largely unexplored regions, and their total utility will be much lower than the total utility of all of the candidates in moderately explored regions. A finite horizon path will therefore choose to visit a moderately explored region over an unexplored region. Figure 3.13 gives an example of such a situation. If the robot's planning horizon is not long enough to go to visit both ends of the hallway, then the finite horizon path will choose to visit the mostly explored room on the left rather than explore the entirely unmapped right side of the hallway.

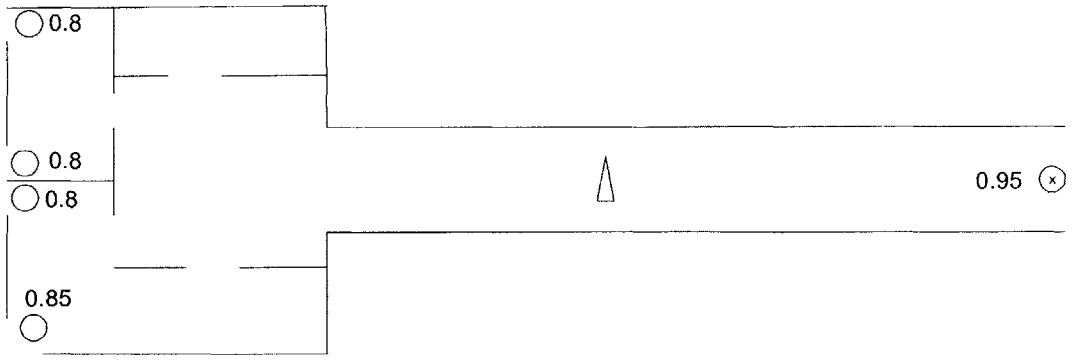


Figure 3.13 Going to Explored Regions over Unexplored Regions

If the robot's goal is to try to map as much new area as quickly as possible, however, then going to moderately explored regions over unexplored regions is exactly the opposite kind of behavior that we want. If the robot can plan a path that takes into account the fact that after the robot visits one candidate, the score of the other candidates will drop, then the robot can avoid this problem somewhat. We do not explore planning to take into account candidate interactions in this thesis, however. Yet visiting moderately explored areas over unexplored areas is desirable if the robot's goal is to map thoroughly. Therefore, the finite horizon method might be good for exploration for thoroughness.

3.3.2.3 Frequency of Unrewarded Sacrifices

We have seen so far that the set of candidates will change frequently in exploration for increasing map coverage, and that finite horizon paths will sometimes cause the robot to go to moderately explored regions over unexplored regions. We also have seen that when the environment is open, or in other situations in which the robot knows the large-scale structure of its environment early on, the robot should be able to execute many of its finite horizon paths to completion. In these situations, there should also not be too many highly unexplored regions; therefore choosing moderately explored regions over unexplored regions will not be an issue. Finite horizon observation planning methods should, therefore, perform better in these situations than greedy or full horizon methods, as long as the execution of these finite horizon paths does not often get interrupted in a way that has the robot make many unrewarded sacrifices. Therefore, we would like to characterize how likely it is that the robot will make unrewarded sacrifices while exploring to increase map coverage.

In estimating how often the robot will get caught making unrewarded sacrifices, we need to estimate how often a finite horizon path makes large sacrifices and how likely it is that the execution of the path gets interrupted after such sacrifices but before the robot can gather any reward. The first point to note is that in most environments, finite horizon paths do not often make many large sacrifices. Usually there are plenty of good candidates nearby for the robot to explore. In addition, a finite horizon path cannot make sacrifices larger than the length of the planning horizon. In exploration for increasing

map coverage, the planning horizon is not usually very long because the set of candidates change so often. However, even if the finite horizon path does not make any large sacrifices, occasionally it is still possible for the finite horizon path to improve upon greedy paths by making small sacrifices that have big payoffs.

If the robot does make a large sacrifice, however, we would like to know how likely it is that the robot will get caught not being able to capitalize on this sacrifice. Unfortunately, in exploration for increasing map coverage, the most likely time for the set of candidates to change enough to significantly alter the robot's path is when the robot is trying to reap some reward. Often making a big sacrifice equates to the robot driving a long distance to the next point, and the reward equates to exploring a relatively unexplored area. Yet the robot is usually very uncertain about the structure of unexplored areas (that is the point of exploring). Therefore once it gets to its destination, the robot may find that it is impossible to gain the reward that it thought it would. Figure 3.10 shows an example of this type of situation. The robot passes up the greedy solution of visiting the nearby candidates and instead travels a long distance to get to an open and largely unexplored region. Upon arriving at the distant region, however, the robot finds out that a wall blocks the path it had planned to explore the new territory. The robot is therefore unable to capitalize on its sacrifice of driving far away from its initial position.

Hence it is not very likely that the robot will make many large sacrifices when exploring. But when the robot does make a big sacrifice, it is somewhat likely that the robot will not be able to capitalize on this sacrifice. Whether or not the finite horizon approach performs better than the full horizon or greedy approaches depends on the particular environment, exploration method, and SLAM algorithm. In some situations the robot may be able to make up for the few times it gets caught making unrewarded large sacrifices by the efficiency of the finite horizon paths the robot is able to execute to completion. In other situations, the robot might get caught making unrewarded sacrifices too often, or the finite horizon paths that the robot executes to completion may not be much more efficient than greedy. However, if the robot knows the large-scale structure of its environment at the beginning of its exploration, then it should be able to execute many of its finite horizon paths to completion. In addition, we have seen that when the robot knows the large-scale structure of its environment, it is less likely that the set of

candidates will change so much that the robot's planned path will change significantly during exploration. Therefore, even if the robot makes large sacrifices, it should be less likely that the robot gets caught not having capitalized on the sacrifice when the robot knows the structure of the environment. Our hypothesis therefore is that the finite horizon approach should excel when the robot knows the large-scale structure of its environment early on in its exploration.

We now understand the properties of the finite horizon observation planning approach of planning an optimally efficient path over a given horizon. We have seen that finite horizon continuous observation planning methods should be ideal for exploration for exploration for increasing map coverage when the robot knows the large scale structure of its environment early on in its mission. In the following chapters we examine a specific implementation of an exploration framework that uses the finite horizon approach. We also present the results of experiments comparing the finite horizon, full horizon, and greedy approaches to observation planning in simulated and real environments using this exploration framework.

4 Optimal Constraint Satisfaction Problem Methods for the S-TSP

In this chapter we present a novel algorithm for solving the S-TSP by casting the problem as an Optimal Constraint Satisfaction Problem (OCSP) [50]. Chapter 3 introduces and formally defines the S-TSP. In an OCSP, as in a Constraint Satisfaction Problem (CSP), the goal is to find an assignment to the variables that satisfies a given set of constraints over possible assignments. However, the solution to an OCSP has the additional requirement that this satisfying assignment must maximize a given utility function over the variables. In the OCSP formulation of the S-TSP, a particular assignment to the variables corresponds to a particular subset of the vertices in the graph. The goal of the Selective Traveling Salesman Problem is, therefore, to find the subset of vertices with the maximum total utility, subject to the constraint that the least-cost path visiting all vertices in the subset must have a cost less than the given threshold cost. The algorithm described in this chapter searches through the possible subsets of vertices in best-first order and checks the constraint on each subset.

Recall that, currently, the most popular methods for solving the S-TSP are branch-and-cut algorithms [16] [15]. Until now, no one has attempted to use OCSP solution methods to solve the S-TSP. Yet powerful methods of solving OCSP's [50] have recently been developed that have proven to be very effective on difficult problems in the domain of Model-based Programming. These methods may also prove to perform better than existing branch-and-cut algorithms at solving the S-TSP for the instances we are interested in. While it is not within the scope of this thesis to perform in depth research into solving the S-TSP as an OCSP, in this chapter we explain how to formulate the S-TSP as an OCSP, and we describe the basic algorithm for solving the S-TSP as an OCSP. Chapter 7 discusses ideas for future research into improving the performance of OCSP-based algorithms for the S-TSP.

The structure of this chapter is as follows. Section 4.1 details how to formulate the S-TSP as an OCSP. Section 4.2 reviews the constraint-based A* search strategy [50] and explains how it can efficiently search through the space of possible subsets of

vertices in best-first order. Section 4.3 goes over methods of checking the cost constraint for each subset of vertices produced by constraint-based A*. In particular, the section shows how to deal with the issue that solving the TSP finds a shortest cycle through a graph, while we are not interested in returning to the start point.

4.1 The S-TSP Viewed as an Optimal Constraint Satisfaction Problem

An OCSP is a special type of CSP. In a CSP we have a set of variables, each of which has a finite domain, and a set of constraints that maps each assignment to the variables to true or false. In an OCSP we also have a set of variables with finite domains and a set of constraints over these variables [50]. In addition, however, we have a utility function that maps all assignments of a special subset of the variables, called the decision variables, to a real number. A solution to an OCSP is an assignment to the decision variables that maximizes utility and for which there exists some assignment to the non-decision variables such that the constraints are satisfied.

It is straight-forward to formulate the S-TSP as an OCSP. For each vertex v_i in the graph, other than the start vertex v_0, we create a corresponding variable x_i in the OCSP. These vertex variables have a domain of $\{0,1\}$. An assignment of $x_i = 1$ corresponds to vertex v_i being included in the sub-graph that the least-cost Hamiltonian cycle is calculated over. Conversely, an assignment of $x_i = 0$ corresponds to vertex v_i being excluded from the sub-graph. Since the utility function is defined for each vertex in the graph, every variable in the OCSP is a decision variable. The utility function for a single decision variable (called an attribute utility function) maps $x_i = 1$ to $u(v_i)$ and $x_i = 0$ to 0. The utility function for any assignment to a set of decision variables is the sum of the attribute utilities of the variables in the set. In order to check the cost constraint, given an assignment to the variables, we take the variables assigned 1, find the corresponding vertices in the graph, and check if the least-cost Hamiltonian cycle through the sub-graph consisting of these vertices and vertex v_0 has a cost less than or equal to L. If the least-cost Hamiltonian cycle has a cost that is less than or equal to L, the constraint is satisfied; otherwise, it is violated. Thus a solution to the OCSP will identify the sub-graph that the solution to the S-TSP is calculated over. Figure 4.1 demonstrates how the

simple S-TSP instance from Figure 3.7 is formulated as an OCSP. Recall from Chapter 3, however, that we will not actually solve the S-TSP on the graph in Figure 4.1a, but first convert the graph to a directed graph. We describe this procedure in Section 4.3. The function TSP(x_i=1) in the figure returns the cost of a solution to the TSP (the least-cost Hamiltonian cycle) for the sub-graph of this directed graph that contains the vertices corresponding to the variables that are assigned 1.

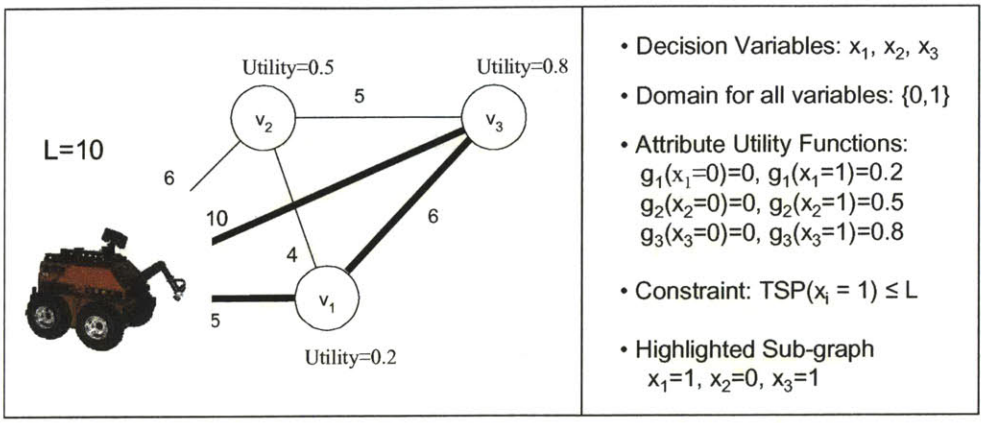


Figure 4.1 OCSP Formulation of an S-TSP Instance

There are a number of reasons to formulate the S-TSP as an OCSP. First, an OCSP is a novel way of formulating an S-TSP, which has not been explored in the literature. Powerful methods for solving OCSP's have recently emerged; therefore, it is worth exploring how well OCSP algorithms perform on our instances of the S-TSP. It may turn out that for the types of graphs we are interested in, OCSP solutions perform better than branch-and-cut methods. In addition, past work has unified branch-and-bound methods with OCSP solution methods [31]; therefore, it may be possible to unify OCSP methods with branch-and-cut methods for S-TSP's.

Another advantage of using OCSP solution methods is that OCSP's are very general. We can incorporate additional constraints over the variables without changing the underlying solution algorithm. For instance, if we want to add the constraint that certain vertices have to be included in the solution, we can do so without making any change to the way in which the algorithm enumerates assignments to the variables to test the consistency of in best-first order (Section 4.2 explains this best-first enumeration

method). We also would not have to change any of the improvements to our algorithm that Chapter 7 proposes that allow the best-first enumeration method to skip over assignments that we know must be inconsistent. We would only have to change the way we check the feasibility of each candidate assignment against the constraints. Finally, the algorithm can handle any utility function which is mutually preferential independent (this term is explained in the Section 4.2). Thus if for some reason we needed to calculate the utility of a set of vertices by taking the product of the individual vertex utilities instead of the sum, we could.

The main algorithm we use for solving the S-TSP as an OCSP is called constraint-based A* [50]. Constraint-based A* is an efficient method based on A* search for searching the space of variable assignments in best-first order. For each assignment, the algorithm checks the constraint by computing the solution to the TSP over the sub-graph corresponding to the assignment (line 8 in Figure 4.2a). Because the algorithm considers all assignments from the highest possible utility on down and returns the first one that satisfies the constraints, the algorithm is guaranteed to return an optimal solution if one exists (the algorithm is complete). In addition, the algorithm will not return an assignment unless it satisfies the constraint; hence, if the algorithm returns an assignment the assignment must satisfy the constraint and must be an optimal solution (the algorithm is sound). The next section describes constraint-based A*. Section 2.4 explains the details of how the algorithm checks the constraint.

4.2 Constraint-based A*

Figure 4.2 provides pseudo-code for constraint-based A*[9]. We explain this pseudo-code throughout this section.

[9] This pseudo-code and the pseudo-code in Figure 4.4 are a simplified version of the pseudo-code provided in the original paper [52]. We have achieved this simplification by specializing the code for solving the S-TSP, which is why we call the top-level function `Constraint-based_A*_S-TSP()`.

```
Constraint-based_A*_S-TSP(attribute utility functions, undirected graph, L)
returns an optimal consistent full assignment to the variables, if one exists

1.  let queue = Make_Queue(Make_Search_Tree_Node({}, no parent))
2.  loop do
3.     if queue is empty
4.            no consistent assignment so return empty assignment
5.     else
6.            let node = Remove_Best(queue, attribute utility functions)
7.            let state = Get_State(node)
8.            if Goal_Test(state, undirected graph, L) is true
9.                   return state
10.           else
11                   let child node = Expand_Variable(node, attribute utility
functions)
12.                  let queue = Enqueue(queue, child node)
13.           endif
14.    endif
15.  end loop
```

a)

```
Goal_Test (state, undirected graph)
returns true iff state is a consistent full assignment to the variables

1.  if state is a full assignment to the variables
2.     return Consistent(state, undirected graph, L)
3.  else
4.     return false
5.  endif
```

b)

```
Consistent(state, undirected graph, L)
returns true iff the sub-graph of the undirected graph corresponding to
state has a Hamiltonian path with a cost <= L

1.  let directed graph = Convert_to_Directed(undirected graph)
2.  let sub-graph = Get_Sub-graph(state, directed graph)
3.  let path = Solve_TSP(sub-graph)
4.  if Cost(path) <= L
5.     return true
6.  else
7.     return false
8.  endif
```

c)

Figure 4.2 Pseudo-code for Constraint-based A*

Constraint-based A* uses a form of state-space search to enumerate the variable assignments of an OCSP in best-first order. Constraint-based A* improves upon the efficiency of A* by exploiting a requirement that the utility function of an OCSP be mutually preferential independent (MPI). If each decision variable x_i has an attribute utility function $g_i(x_i)$ defined for it, and if the utility function for full assignments to the decision variables is a function of the values of the attribute utility functions, that is the utility function is of the form $G(g_1(x_1), g_2(x_2), \ldots, g_n(x_n))$, then the utility function for full

assignments is a multi-attribute utility function. An MPI utility function is a multi-attribute utility function which can be maximized by maximizing the attribute utility of each decision variable *independent of all of the other decision variables*. For example, an additive utility function $G(g_1(x_1), g_2(x_2), \ldots, g_n(x_n)) = g_1(x_1)+g_2(x_2)+\ldots+g_n(x_n)$ is MPI because we can find the assignment that maximizes G by finding the value for x_1 that maximizes $g_1(x_1)$, the value for x_2 that maximizes $g_2(x_2)$, and so on. Constraint-based A* takes advantage of MPI utility functions in order to limit the expansion of each search tree node to only its best child, and to efficiently calculate an admissible heuristic at each node.

In the constraint-based A* framework, search states are partial or full assignments to the decision variables. For example, in the OCSP in Figure 4.1, $\{x_1=0\}$ is a state, as is $\{x_1=0, x_2=1, x_3=1\}$. In order to move from one state to the next, constraint-based A* finds a variable that has not been assigned in the current state and assigns it one of its possible values. Given state $\{x_1=0\}$, if we choose the next variable to be x_2, then $\{x_1=0\}$ can transition to $\{x_1=0, x_2=0\}$ or $\{x_1=0, x_2=1\}$. The initial state of the search tree is the state in which no decision variables have been assigned a value, and leaves of the search tree are states in which all of the decision variables have been assigned a value. The search proceeds by expanding the search node with the best estimated utility until it reaches a leaf. The do loop in Figure 4.2a performs this expansion. The constraint checker then checks the full assignment represented by the leaf node (line 2 of Figure 4.2b), and if the assignment is consistent it is returned as the solution to the OCSP[10]. If the assignment is inconsistent the search continues to expand other search nodes until the search reaches another leaf. Figure 4.5 shows progression of the search on the simple example from Figure 4.1.

At each partial assignment in the search tree, constraint-based A* calculates a heuristic estimate of the utility of the best full assignment containing the partial assignment. Because the utility function is MPI, the best full assignment to the remaining variables does not depend on the partial assignment. The best possible assignment

[10] Alternatively, the constraint checker can check the resulting assignment every time the search node with the best estimated utility is expanded, regardless of whether the assignment is full or partial. This frequent constraint checking allows constraint-based A* to prune branches of the search tree before reaching the leaves.

assigns each of the remaining variables to the value that maximizes its attribute utility. This estimate of the utility of the best full assignment given the partial assignment is admissible, meaning that it never underestimates the actual utility of the best consistent assignment containing the given partial assignment. Because the heuristic is admissible, the A* search is guaranteed to find an optimal solution.

Figure 4.3 shows a fragment of the search tree for the example from Figure 4.1. Each node is labeled with the assignment that created it. The nodes list the utility of the partial assignment of the state at that node (the g value), and the heuristic estimate of the utility of the best consistent assignment to the remaining variables (the h value). For example, at the node for state $\{x_1 = 1\}$ (node n2 in Figure 4.3), the utility of this partial assignment is 0.5 and so g = 0.5. Because the utility function is MPI, the best possible assignment to the remaining variables must be x_2=1 and x_3=1. The utility of $\{x_2$=1, x_3=1$\}$ is 0.2+0.8=1.0, and therefore the h value for node $\{x_1$=1$\}$ is 1.0. Looking at the leaves under node $\{x_1$=1$\}$, we see that the leaf with the best possible utility reachable from the node is n4, which indeed corresponds to full assignment $\{x_1$=1, x_2=1, x_3=1$\}$. The fact that this best leaf has a utility that is greater than that of all the other leaves illustrates that the node's estimate is optimistic. This leaf node is inconsistent, however, illustrating that the leaf is only an *estimate* of the best consistent full assignment[11]. The best *consistent* full assignment is the leaf $\{x_1$=1, x_2=0, x_3=1$\}$. The score of a given search node is the sum of its g and h values, and nodes are expanded in order of best to worst score by A* search.

[11] Recall that the algorithm first transforms the undirected graph in Figure 2.3 into a directed graph before checking for consistency. As a result, a set of vertices in the undirected graph is consistent iff there is a maximum utility Hamiltonian *path* through their corresponding sub-graph with distance less than or equal to L.

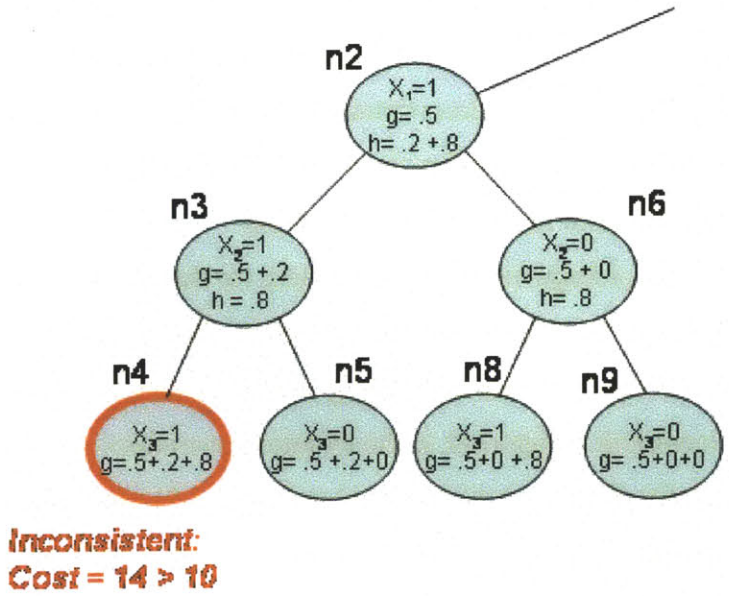


Figure 4.3 Partial Search Tree for Constraint-based A*

A key insight of constraint-based A* is that a given node in the search tree does not have to be expanded to all of its possible children, as in normal A* search. Remember that expanding a node entails choosing some unassigned variable in the state of the node and assigning it one of its possible values. Because the utility function is MPI, we know that the best possible full assignment (leaf) extending from a given node must assign to the next variable in the search the value that maximizes its attribute utility. Therefore the algorithm expands each node only to the node's best child. The function `Expand_Variable_Best_Child()` in Figure 4.4b shows how constraint-based A* expands a node only to the node's best child for the specific case of S-TSP OCSP's. Figure 4.5a shows the search tree for the example from Figure 4.1 after the first four node expansions.

```
Expand_Variable(node, attribute utility functions)
returns the best nodes expanded from node

1.  if Leaf_Node(node)
2.     let nodes = Expand_Next_Best_Sibling_of_Ancestors(node, attribute
utility functions)
3.  else
4.     let nodes = Expand_Variable_Best_Child(node, attribute utility
functions)
5.  endif
6.  return nodes
```

a)

```
Expand_Variable_Best_Child(node, attribute utility functions)
returns a child node of node with a best utility assignment

1.  let state = Get_State(node)
2.  if all variables are assigned in state
3.    return {}
4.  else
5.    let yi = some unassigned variable in state
6.    return {Make_Node({yi = 1}, node)}
7.  endif
```

b)

```
Expand_Next_Best_Sibling_of_Ancestors(node, attribute utility functions)
returns siblings of node and its ancestors with the next best assignment

1.  if Root(node)
2.    return {}
3.  else
4.    return Expand_Next_Best_Sibling(node, attribute utility functions) U
Expand_Next_Best_Sibling_of_Ancestors(Get_Parent(node), attribute utility functions)
5.  endif
```

c)

```
Expand_Next_Best_Sibling(node, attribute utility functions)
returns node's sibling with the next best assignment

1.  if Root(node)
2.     return {}
3.  else
4.     let assignment = Get_Assignment(node)
5.     let value = Get_Value(assignment)
6.     if value = 0
7.            return {}
8.     else
9.            variable = Get_Variable(assignment)
9.            return {Make_Node({variable = 0}, Get_Parent(node))}
10.    endif
11.  endif
```

d)

Figure 4.4 Search Node Expansion Functions

Once constraint-based A* has expanded nodes down to a leaf (as in Figure 4.5a), the algorithm checks the full assignment of the leaf for consistency. Recall that in our case, a full assignment is consistent if and only if the corresponding sub-graph has a Hamiltonian cycle through it with a cost less than or equal to L (see Figure 4.2c for pseudo-code). Because A* will only check a leaf if it has the highest score of all the nodes in the queue, and because constraint-based A*'s heuristic is admissible, if the constraint checker determines that a leaf is consistent, then that leaf must be a solution to the OCSP. If the constraint checker determines that a leaf is inconsistent, then the search must find the next best leaf node to test. Because the utility function is MPI, we know that the next best leaf node must assign the best possible value to each variable while still being different from the last leaf node. Thus, the next best leaf node will be the same as the last leaf node, except that exactly one of the variables will change its value to its next best value. Therefore, when constraint-based A* determines that a leaf is inconsistent, the search will expand every ancestor node of the leaf to the ancestor's *next* best value and place these next best nodes on the queue. The functions in Figure 4.4 summarize how constraint-based A* expands nodes. `Expand_Variable()` is the top level node expansion function that the `Constraint-based_A*_S-TSP()` function in Figure 4.2a calls. The function `Expand_Next_Best_Sibling_of_Ancestors()` in Figure 4.4c shows how constraint-based A* expands every ancestor of a leaf to the ancestor's next best value for the specific case of S-TSP OCSP's.

Figure 4.5b depicts an expansion of every ancestor node, which is performed right after the leaf in Figure 4.5a is found to be inconsistent. The full assignments that will result from expanding each of these new nodes in the queue down to its best leaf is shown in Figure 4.5c. These leaf nodes represent the next possible full assignments that the search could find. These full assignments are the same as the full assignment in Figure 4.5a, except that each candidate has exactly one of its variables assigned to its next best value. Thus, these next possible full assignments reflect that we know that the next best assignment must change exactly one variable to its next best value.

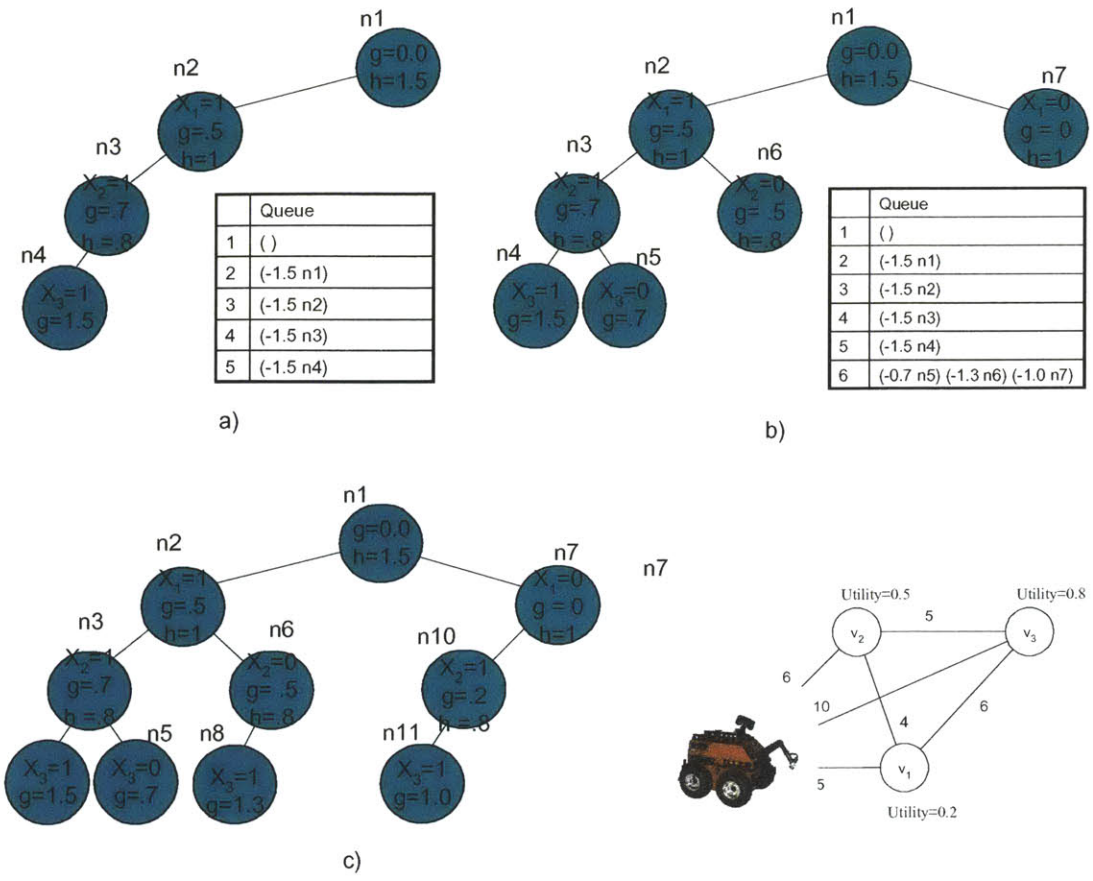


Figure 4.5 Node Expansion in Constraint-based A*

4.2.1 Full Example of Constraint-Based A* on a S-TSP

Figure 4.6 shows how constraint-based A* searches through all possible variable assignments, in best-first order, for the S-TSP example in Figure 4.1. For every step of the search shown, the figure also illustrates the queue at that point. The queue is a list of search tree nodes, and is created on line 1 of the top level pseudo-code in Figure 4.2a. For each node in the queue, the figure depicts the score and the name of the node. Therefore, the queue entry (1.5 n4) indicates that node n4 has a score of 1.5, which is the utility of the partial assignment plus the estimated utility of the rest of the assignment. At each step, constraint-based A* removes the node with the highest score from the queue (line 6 in Figure 4.2a) and expands it to its best child (line 11 in Figure 4.2a). The search then places this child node into the queue. If a node cannot be expanded because it is a leaf, the node's full assignment is checked for consistency (line 8 in Figure 4.2a). If the

node is inconsistent, the best child node of each ancestor of the leaf is put onto the queue (line 11 in Figure 4.2a). It is important to remember that the algorithm converts the undirected graph in the figure into a directed graph before checking consistency (line 1 of Figure 4.2c). Therefore, a subset of vertices of the undirected graph is consistent iff there exists a maximum utility Hamiltonian *path* of length less than or equal to L, through the sub-graph corresponding to the subset. Section 4.3 discusses how consistency is checked in more detail.

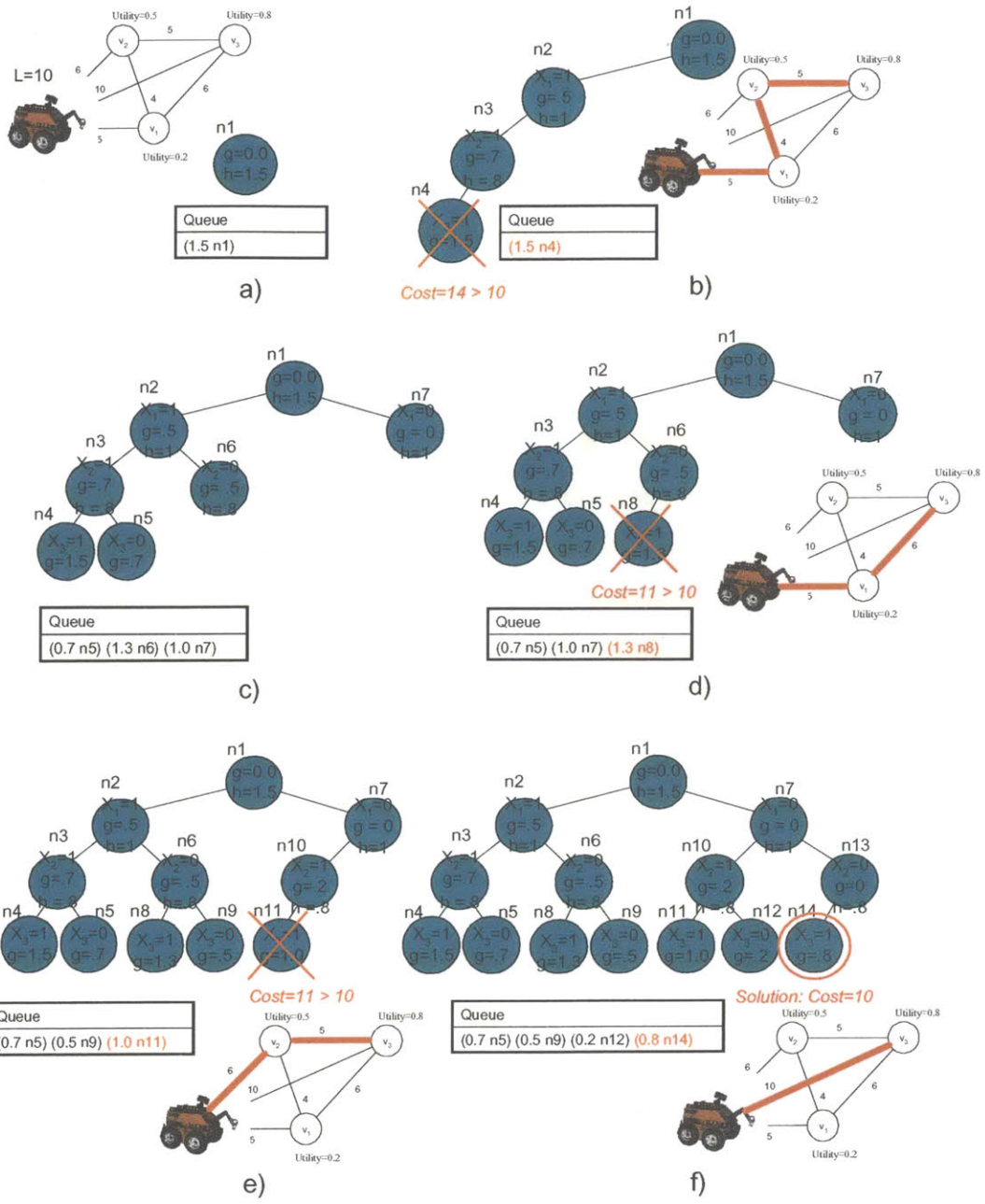


Figure 4.6 Solving an S-TSP with Constraint-based A*

The search starts with no variables assigned, as Figure 4.6a depicts. The search proceeds by removing one node from the queue and replacing it with the best child, thus keeping the queue at length one. Once node n4 is found to be inconsistent (Figure 4.6b), the queue length jumps to three elements - one for each possible variable that the search can change (Figure 4.6c). Since node n6 has the next highest score, at 1.3, it must

contain the next best full assignment. Indeed, node n8 is the next best leaf, but n8 is also inconsistent (Figure 4.6d). Node n6 can expand to another child, node n9, and thus node n9 is added to the queue. The next best full assignment is node n11's assignment, however. After n11 is found inconsistent, the next best full assignment found by the search is n14. Node n14 is consistent, and therefore its assignment, $\{x_1=0, x_2=0, x_3=1\}$ is the solution. The maximum utility Hamiltonian path through the sub-graph corresponding to this assignment goes from the start vertex v_0 straight to v_3.

4.3 Constraint Checking

The final remaining piece of the algorithm for solving the S-TSP as an OCSP is the method for checking the constraint for a given full assignment to the variables. The basic idea is that a full assignment to the variables is consistent iff the solution to the *TSP* on the sub-graph corresponding to the assignment has a length that is less than or equal to L. However, there is one problem with this basic approach. TSP solvers by definition find least-cost *cycles* through graphs; however, we do not want the robot to plan for returning to its starting point. Instead, we would like the robot to find is the least-cost Hamiltonian *path* through the sub-graph. Nevertheless, very efficient TSP solvers such as Concorde [54] are readily available, and we would like our algorithm to be able to use these programs. As a result, we need a method of reducing the problem of finding a least-cost Hamiltonian path through a graph to the TSP. This section describes one such reduction method. Our reduction method is derived from an existing reduction from the TSP to the problem of finding the least-cost Hamiltonian path through the graph starting at *any* vertex [39].

Given an undirected graph that we want to find a least-cost Hamiltonian path through, our approach constructs a directed graph and solves the TSP on it. The TSP on directed graphs is called the Asymmetric TSP (ATSP), and the TSP on undirected graphs is called the Symmetric TSP (STSP). The directed graph is the same as the undirected graph, except that it adds an edge of cost zero from each vertex to the start vertex. In other words, to construct the directed graph, we start with a directed graph that is equivalent to the undirected graph. Then, for each vertex in the directed graph, we set the edge cost from that vertex to the start vertex to 0. A least-cost Hamiltonian cycle in the

directed graph then yields a least-cost Hamiltonian path beginning at the start point in the undirected graph. In order to find this Hamiltonian path in the undirected graph, we simply follow the sequence of vertices in the Hamiltonian cycle, beginning at the start point, until every node has been visited.

Figure 4.7 depicts a simple example of converting an undirected graph into a directed graph in order to find a least-cost Hamiltonian path through the undirected graph. Figure 4.7a shows the undirected graph, and Figure 4.7b shows the corresponding directed graph. In these figures, vertex 1 is the start vertex. Note that the directed graph is equivalent to the undirected graph, except that the edge from 2 to 1 and the edge from 3 to 1 have a cost of 0.

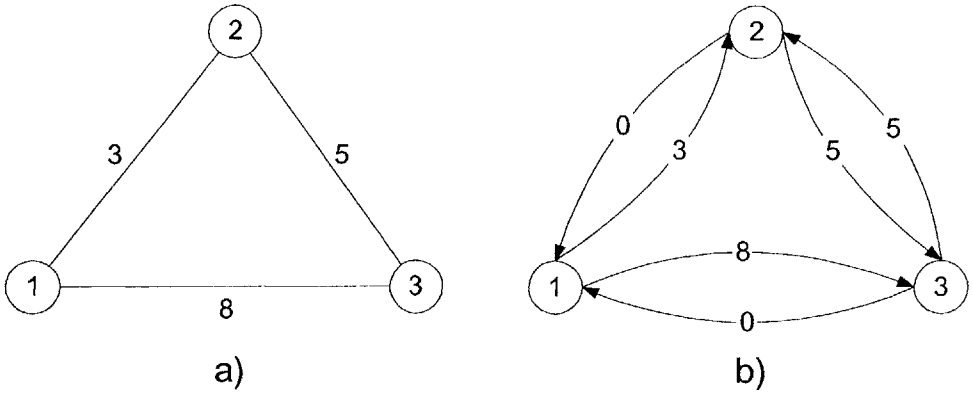


Figure 4.7 Converting an Undirected Graph into a Directed Graph

There are only two possible Hamiltonian paths through the undirected graph starting at vertex 1: path <1, 2, 3> and path <1, 3, 2>. Therefore, we can see directly that the least-cost Hamiltonian path is the path <1, 2, 3>, without running a TSP solver on the directed graph. Solving the TSP on the directed graph anyways, we see that there are only two possible Hamiltonian cycles: cycle <1, 2, 3, 1> and cycle <1, 3, 2, 1>. The cycle <1, 2, 3, 1> has the lower cost of 8; therefore this cycle is the solution to the TSP on the directed graph. And indeed, this TSP solution yields the least-cost Hamiltonian path through the undirected graph, <1, 2, 3>.

We can show how this mapping works informally, through a proof by contradiction. We want to show that any least-cost Hamiltonian cycle in the directed graph is a least cost Hamiltonian path in the undirected graph. First, we note that the cost

of any path in the undirected graph that never has the start node as a destination is equal to the cost of the equivalent path in the directed graph. Now take some least-cost Hamiltonian cycle through the directed graph. Assume (for contradiction) that the corresponding path in the undirected graph is not a least-cost Hamiltonian path, beginning at the start point. Then either the path must not be Hamiltonian (it must not visit every node exactly once), it must not begin at the start point, or it must not be least-cost. Yet the path must visit every node exactly once if the cycle in the undirected graph did, and the path is defined to begin at the start point. Therefore, the path must not be least-cost. However, if the path is not least-cost, then there must be some other Hamiltonian path beginning at the start point in the undirected graph that has a lower cost. By our first observation, if such a lower cost path existed in the undirected graph, then a corresponding path in the directed graph must exist with the same cost. Yet in order to make this path in the directed graph a cycle, all we have to do is add the start vertex to the end of the path. Traversing this final edge must have a cost of 0, and, therefore, the cost of the cycle must still be lower than the cost of the cycle that we initially said was a least-cost cycle. Thus, we have a contradiction.

Even though there are methods for solving the ATSP, we would still like to convert the directed graph back into an undirected graph before finding the optimal tour. Methods for solving the ATSP take advantage of the asymmetric nature of the distance matrix representing an input directed graph, and methods for solving the STSP take advantage of the symmetric nature of the distance matrix representing an input undirected graph. Because the directed graph produced by this mapping has an almost symmetric distance matrix, STSP algorithms should be better able to handle the graph [26].

In order to convert a directed graph produced by this mapping into an equivalent (in the eyes of the TSP) undirected graph [26], we first note that all of the directed edges involve the start vertex. Therefore, to create the undirected graph we copy each vertex in the directed graph and make two copies of the start vertex. One copy of the start vertex handles all of the edges that come into the start vertex in the directed graph, and the other copy of the start vertex handles all of the edges that come out of the start vertex in the directed graph. An edge with a very large negative cost –M connects the two start vertices. The rest of the edges in the directed graph are collapsed with their equivalent

dual edge into a single edge in the undirected graph. This transformation always makes the out start vertex occur immediately after the in start vertex in the optimal tour, thus mimicking the behavior of the directed graph. The length of the optimal tour on this undirected graph is equal to the length of the optimal tour on the directed graph minus M. Figure 4.8 shows an example of this conversion for the simple graph from Figure 4.7b.

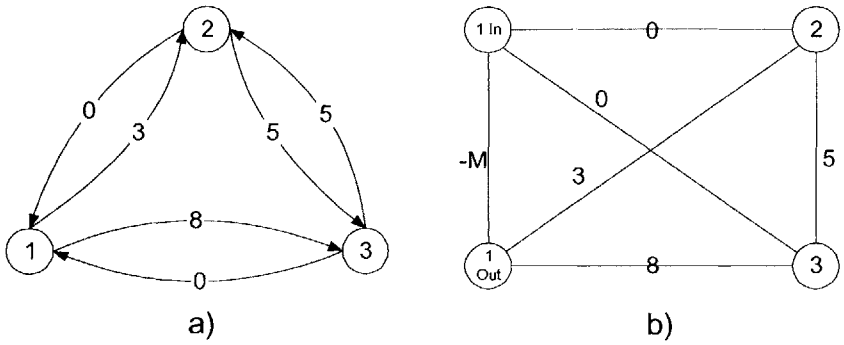


Figure 4.8 Converting a Directed Graph into an Undirected Graph

Thus, given some undirected sub-graph to check the constraint on, the algorithm maps the sub-graph to a directed graph, and then converts the directed graph back into an equivalent undirected graph. This final undirected graph is the input to a TSP solver. The length of the optimal tour calculated by the TSP solver must be less than or equal to L-M in order for the variable assignment to be consistent. Figure 4.9 shows these graph transformations for the full graph from Figure 4.1.

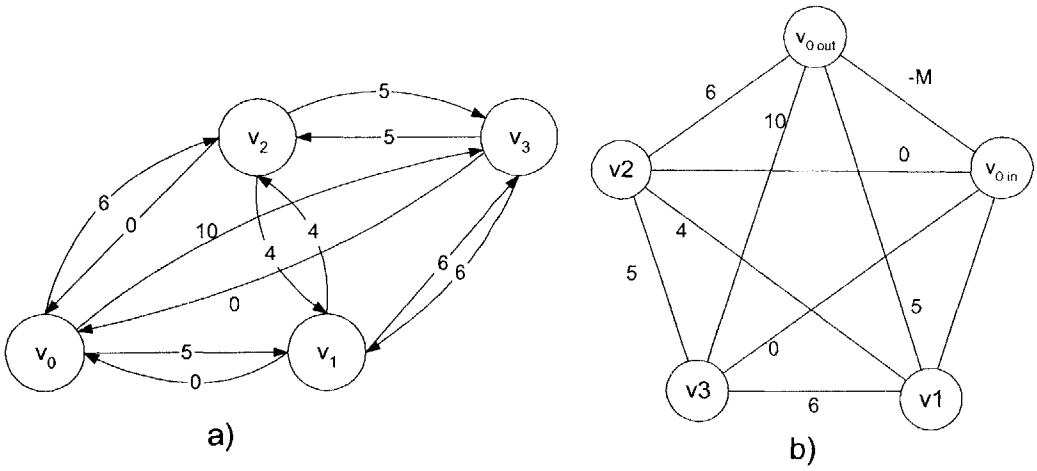


Figure 4.9 Graph Transformations for S-TSP Example

We now know how to formulate the S-TSP as an OCSP. In addition, we understand how to solve the S-TSP as an OCSP by using constraint-based A* to enumerate all possible full assignments to the variables in best first order, and checking each of these assignments for consistency. Our method for checking the consistency of an assignment finds the sub-graph corresponding to the assignment, converts this sub-graph into a directed graph, converts this directed graph into a new undirected graph, and solves the TSP on this new undirected graph. The assignment is consistent iff the path cost of the solution to the TSP is less than or equal to L.

In the following chapter, we describe the architecture of our experimental system and explain how we use this approach to solving the S-TSP to perform continuous observation planning.

5 Autonomous Exploration Using Fixed Horizon Observation Planning

In this chapter we describe the implementation of autonomous exploration with which we performed our experiments. Specifically, we give the architecture of the complete experimental system, which takes as input the sensor readings of a real or simulated robot and outputs commands to the robot's motors that will cause the robot to explore its environment. The experimental system also builds and updates a feature-based map of its environment using these sensor readings. The goal of the system's exploration method is to increase the coverage of this feature-based map as efficiently as possible. We discussed feature-based maps and exploration for increasing map coverage in Chapter 2.

The purpose of our implementation was to test the main claim of this thesis, that continuous observation planning using the finite horizon approach will increase the efficiency of many exploration methods in certain environments, for a particular candidate identification and scoring algorithm. In order to test this claim, we would like to compare how efficiently the robot explores using each of the continuous observation planning methods that we discussed in Chapter 3: the greedy method, the full horizon method, the fixed horizon method, and the receding horizon method. Our experimental system, therefore, needed to be able to swap in and out any of these four methods of continuous observation planning.

In summary, the specification of our experimental system was that the system must work on a real or simulated robot, cause the robot to explore to increase the coverage of a feature-based map as efficiently as possible, and be able to use any of the four methods for continuous observation planning above. In Section 5.1 we describe the overall architecture for this experimental system. Then, in Section 5.2 and Section 5.3 we explain how the components of the system that we have not covered already in this thesis work. Finally, in Section 5.4 we go over a simple example of the execution of the entire experimental system.

5.1 Overall Architecture of Implementation

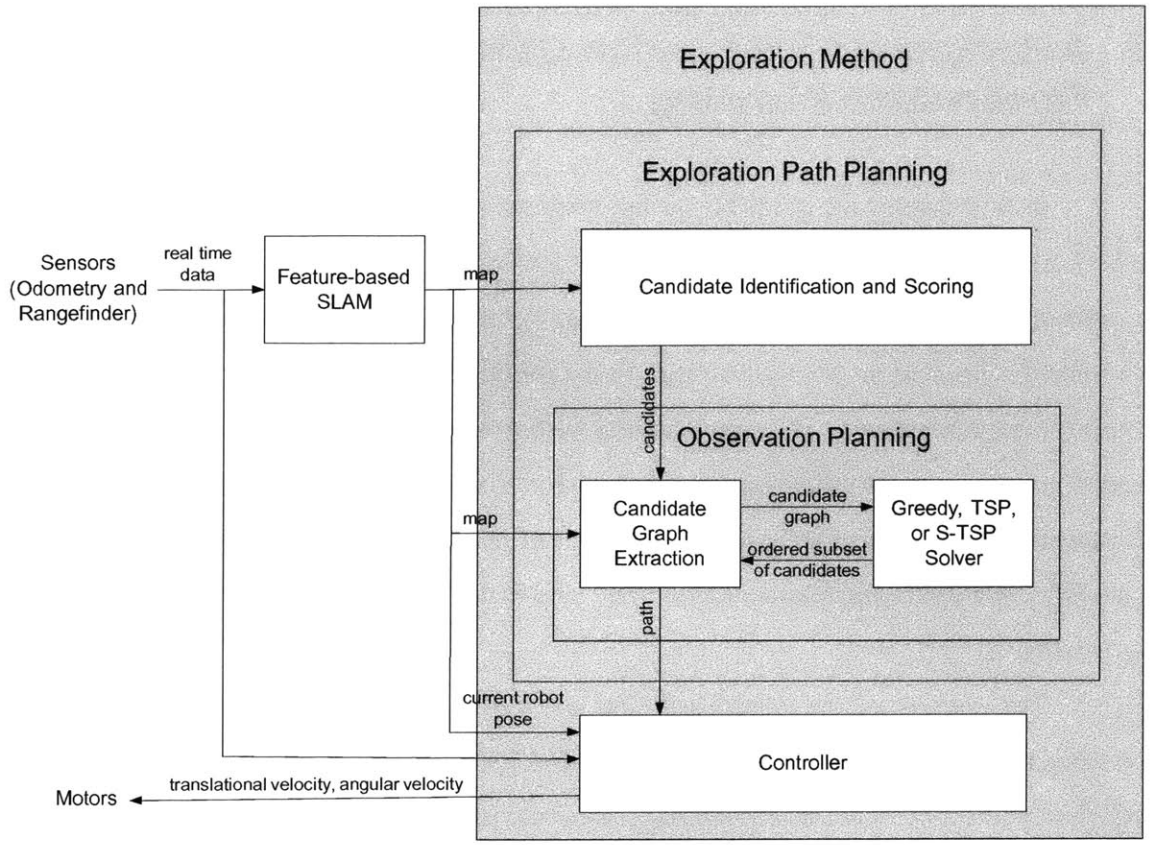


Figure 5.1 Architecture of Experimental System

Figure 5.1 shows the overall architecture for the system we used in our experiments. Looking at the flow of information through the components, the feature-based SLAM component constantly reads the robot's sensors and outputs the most recently updated feature map of the environment. In our experiments, the sensors that we used were the odometer and the laser scanner. Chapter 2 described how the feature-based SLAM component builds a map from such sensor data. In order for the SLAM component to produce the best estimate of the state of the world possible, it is important that the component read the robot's sensors and update the map as often as possible. Therefore, the SLAM component and the exploration method component run as separate processes in our implementation. An inter-process communication (IPC) framework allows the SLAM component to notify the exploration method component of the most

recent map. Figure 5.2 gives pseudo-code for these two processes[12]. We have altered line 2 of the pseudo-code for the exploration method process in Figure 5.2b from the pseudo-code in Figure 2.2 to reflect how the IPC framework works. Instead of the exploration method taking a "constantly updating map" as a one time input, the SLAM algorithm sends the exploration method a message describing the most recent map by calling the function `Send_Messege_To_Exploration_Method()` every time the map is updated. The IPC framework places this message in the exploration method's inbox. Then, when the exploration method reaches line 2 in its pseudo-code, it reads this message and reconstructs the most recent map by calling the function `Get_Most_Recent_Map()`.

```
SLAM()
returns nothing

1.  while Mission_Completed() is false
2.    let data = Read_Sensors()
3.    let current map = Update_Map(data)
4.    Send_Messege_To_Exploration_Method(current map)
5.  endwhile
```

a)

```
Explore_Continuous()
returns nothing

1.  while Mission_Completed() is false
2.    let map = Get_Most_Recent_Map()
3.    let robot pose = Get_Current_Robot_Pose(map)
4.    let path = Plan_Exploration_Path(map, robot pose)
5.    Execute_Segment_of_Path(path)
6.  endwhile
```

b)

Figure 5.2 Pseudo-code for SLAM Process and Exploration Method Process

[12] As a matter of fact, the S-TSP solver component and the controller component also run as independent processes in our implementation. By putting these components of the exploration method in separate processes, we pipeline the computation of the exploration method and thereby decrease the length of time that any component of the exploration method has to wait to perform its processing. For the sake of simplicity, however, we will not change our pseudo-code from Chapter 2 and Chapter 3 to reflect these two additional processes.

Once the block labeled "Exploration Method" in Figure 5.1 has the most recently updated map, the exploration method computes the controls to send to the robot's motors. We covered the structure of a generic exploration method in detail in Chapter 2 and Chapter 3. Besides the one change to line 2 of the pseudo-code for the function `Explore_Continuous` in Figure 2.2, the experimental system follows this generic exploration method structure exactly. We now provide a brief review of this structure.

An exploration method constantly reads the SLAM algorithm's most recently updated map and constantly outputs controls that cause a robot to explore its environment. As Figure 5.2 shows, we break an exploration method down into exploration path planning and executing paths. Exploration path planning entails taking a map and a robot position and computing a path for the robot to execute that should add information to the map. In particular, our implementation always performs continuous exploration path planning (see Chapter 2 for a discussion of continuous exploration path planning). Executing paths entails taking a path and the current robot position and outputting controls (in our case the desired translational and rotational velocities of the robot) that will cause the robot to move along this path. In Figure 5.1, the block labeled "Controller" handles the execution of paths.

As we saw in Chapter 3, most methods of exploration for increasing map coverage break down exploration path planning into candidate identification and scoring and observation planning (see Figures 3.9b and 3.9c for example pseudo-code). In candidate identification and scoring, the robot takes a map and the robot's position and outputs a set of candidate observation points. In observation planning, the robot takes this set of candidates, a map, and the robot's position, and outputs a path that takes a robot to some subset of the set of candidates. This path is the output of the exploration path planner as well as the observation planner.

When we looked at the finite horizon approach to observation planning in Chapter 3, we found it useful to break down the approach into candidate graph extraction and solving the S-TSP (see Figure 3.8 for pseudo-code). Candidate graph extraction involves taking a map, a robot position, and a set of candidates and computing a graph. There is one vertex in the graph for each candidate and one vertex for the robot's position, and the

cost of each edge (i, j) is the cost of the least-cost path through the map that avoids obstacles between the two candidates (or the candidate and the robot's position) i and j. The S-TSP solver then takes this graph as input and outputs an ordered subset of the candidates to visit. Finally, the observation planner uses the least-cost paths between each pair of candidates that it found in building the candidate graph to turn this ordered subset of candidates into a path the robot can execute.

Notice that we could substitute a TSP solver for the S-TSP solver in the above description of the finite horizon observation planning approach, and the observation planner would output a full horizon path for the robot to execute instead of a finite horizon path. It would also be easy to make a "greedy problem solver" that would take a candidate graph as input and output the candidate that minimizes a given greedy function (see Chapter 3 for an explanation of greedy functions). Therefore, in order to satisfy our requirement that the experimental system be able to use the full horizon, greedy, fixed horizon, and receding horizon continuous observation planning methods, we break observation planning down into candidate graph extraction and running a TSP solver, an S-TSP solver with an appropriate horizon, or a "greedy problem solver" on the candidate graph. We refer to the part of the experimental system that includes all components except the "solver" as the *exploration framework.* In all of our experiments, the exploration framework never changed. In order to perform an experiment using, say, the full horizon continuous observation planning method, we simply plugged a TSP solver into the exploration framework.

We now understand how all of the modules in our experimental system fit together. Furthermore, we have already covered how many of the modules in our implementation work in previous chapters. We described the feature-based SLAM algorithm that our system uses in Chapter 2. We used feature-base SLAM in our experiments largely because we already had an implementation of our feature-based SLAM algorithm available, but also partially because exploration using a feature map as the sole map representation has not been researched extensively. In order to perform candidate identification and scoring, we use the candidate identification and scoring part of the Newman, Bosse, and Leonard exploration method. Chapter 2 described the Newman, Bosse, and Leonard exploration method in detail. We base our candidate

identification and scoring component on this particular exploration method because it is the only exploration method that we know of made for exploration using a feature map as the sole map representation. Finally, in order to solve the S-TSP, we implemented the algorithm that Chapter 4 described. And in order to solve the TSP, both to plug directly into the exploration framework and in our implementation of the S-TSP solver, we used the free Concorde TSP solver [54].

The only components of our experimental system that we have not yet described, therefore, are the candidate graph extraction module and the controller module. Therefore, in Section 5.2 we describe the way our implementation performs candidate graph extraction. Then, in Section 5.3, we briefly describe the salient features of our controller. In Section 5.4 we provide an example of all of the components of our experimental system working together during exploration.

5.2 Candidate Graph Extraction

In order to perform continuous observation planning, our implementation continuously performs candidate graph extraction. Recall that in order to extract a candidate graph, the robot takes as input the current robot position, the most recently updated map, and a set of candidates, and returns a graph that contains one vertex for each candidate and one vertex for the robot. For a given edge (i, j), where i and j are vertices, the cost of the edge is equal to the least-cost path that avoids obstacles between the positions of i and j in the map. In our implementation, we use the physical length of a path as its cost because it makes the most sense when using the finite horizon method to compute a path over a set distance. As we discussed in Chapter 3, we would like the robot to be able to execute its finite horizon path to completion without the set of candidates changing significantly, and the farther the robot travels, the more likely the set of candidates is to change at some point along the path. We, therefore, limit the physical length of the finite horizon path by making the horizon, and thus the path cost, a distance.

We now explain how our implementation performs candidate graph extraction. Most of the work in extracting a candidate graph is in computing the least-cost path between every pair of candidates that avoids all obstacles. We usually call a method for computing a path between two points that avoids all obstacles a path planner. Many good

methods for path planning exist [6] [28] [33]. Because our exploration implementation performs *continuous* observation planning, we must constantly plan a least-cost path between every pair of candidates in the map. Therefore, the path planning algorithm we use must be very efficient. We choose to use a visibility graph path planner [51] because it is very simple and because it outputs approximately the shortest path between any two points that avoids all obstacles.

A visibility graph path planner takes as input a set of obstacles, a set of waypoints, and the current robot position. In our implementation, obstacles are polygons. The path planner can then output an approximately least-cost path between the robot's position and a waypoint or between any two waypoints. In order to calculate paths, the path planner constructs a visibility graph using the corners of the polygonal obstacles, the current robot position, and the waypoints as vertices. A visibility graph is a graph that contains an edge between every pair of vertices, as long as that edge does not pass through an obstacle. Thus, any path consisting of edges of the visibility graph will never run into an obstacle. The cost of each edge in a visibility graph is the straight line distance between the edge's endpoints. If the robot were a zero dimensional point, then the shortest path for the robot to take between any two waypoints that avoids all obstacles could be found by searching the visibility graph for the shortest path between the corresponding vertices. Even if the robot is not a zero dimensional point, however, we can still compute a path between any two waypoints that avoids all obstacles by growing each polygonal obstacle by a distance equal to the radius of a circle that circumscribes the robot. The path that we find between two vertices by searching a visibility graph with enlarged obstacles is not guaranteed to be the shortest path between the corresponding waypoints; however, the path will be very close to the shortest path.

In order to construct a candidate graph using a visibility graph path planner, we use the candidate observation points as the waypoints in the input. We also need to provide a set of obstacles in the input. In our system, the line feature SLAM map contains the robot's best information about the location of objects in the robot's environment. Therefore, in Section 5.2.1 we describe a novel method of taking the line feature SLAM map and producing from it a set of rectangular obstacles. Once the visibility graph path planner has built a visibility graph over the obstacles, the candidates,

and the current robot position, we construct the candidate graph by searching the visibility graph for the shortest path between every pair of candidates and the shortest path between every candidate and the robot position. We then set the cost of each edge in the candidate graph to the length of the corresponding shortest path in the visibility graph, and the candidate graph is complete.

The method for extracting the candidate graph as we have described it is not very efficient, however. Because our implementation performs continuous observation planning, the robot constantly extracts a candidate graph from the most recently updated map. And every time the robot extracts a candidate graph, the robot must extract obstacles from the map, build a visibility graph, and search the visibility graph for the shortest path between every pair of candidates and between every candidate and the robot's current position. Yet often, many of the features in the map are not changed by map updates. As a result, many of the obstacles are the same each time we build the visibility graph, and many of the edges in the visibility graph are the same each time we search the visibility graph. Instead of building and searching the visibility graph from scratch each time we extract the candidate graph, therefore, we would like to only have to update the visibility graph and shortest paths from the last time we extracted the candidate graph to take into account what has changed. Although there are currently no algorithms for incrementally updating visibility graphs, there are algorithms for incrementally updating searches over graphs. Our implementation uses the D* search algorithm to incrementally search the visibility graph [45].

D* is an all-source single-destination shortest path search algorithm. Therefore, the robot must run one D* instance for each candidate observation point. On its first iteration, D* computes these shortest paths from scratch. Then, if a map update causes the visibility graph to change (for example, by changing an edge cost or adding a vertex), D* recalculates only those shortest paths that were affected by the change. Therefore, we expect that using D* will significantly reduce the cost of calculating the all-candidate-pairs shortest paths.

Figure 5.3 shows more precisely how our implementation extracts candidate graphs. The robot keeps track of the last visibility graph it built in order to see what has changed when it constructs a new visibility graph. The robot also maintains a hash map

of D* instances. Each candidate observation point has one D* instance, and a candidate's D* instance keeps track of the shortest path from each vertex in the visibility graph to that candidate. The hash map's key for a given D* instance is the candidate that the D* instance computes shortest paths to (the D* instance's goal). We discuss how the function `Extract_Obstacles()` on line 1 works in Section 5.2.1. The function `Get_Last_Visigraph()` on line 3 returns the visibility graph that was stored on the last call to `Extract_Candidate_Graph()` by the function `Set_Last_Visigraph()` on line 9. The function `Remove_Extra_D*_Graphs()` on line 4 iterates through the hash map of D* instances and removes any D* instance from the hash map that has a goal that is no longer in the set of candidates. Therefore, whenever a candidate is removed from the map, `Remove_Extra_D*_Graphs()` will remove the corresponding D* instances.

```
Extract_Candidate_Graph(partially completed map, candidate set, robot
pose)
returns candidate graph

1.  let obstacle list = Extract_Obstacles(partially completed map)
2.  let current visibility graph = Build_Visibility_Graph(robot pose,
candidate set, obstacle list)
3.  let last visibility graph = Get_Last_Visigraph()
4.  Remove_Extra_D*_Graphs(candidate set)
5.  Update_D*_Graphs(last visibility graph, current visibility graph)
6.  Add_New_D*_Graphs(candidate set, current visibility graph)
7.  Search_All_D*_Graphs()
8.  let graph = Build_Candidate_Graph_From_D*_Graphs()
9.  Set_Last_Visigraph(current visibility graph)
10.  return graph
```

Figure 5.3 Candidate Graph Extraction Pseudo-code

The function `Update_D*_Graphs()` on line 5 compares the last visibility graph to the current visibility graph and makes a list of all of the nodes that were added or removed and all of the edges that changed their costs or were removed. The function then notifies each D* instance in the hash map which nodes and edges were added, removed, or changed. The function `Add_New_D*_Graphs()` iterates through the set of candidates and sees if there are any candidates that do not have a D* instance in the

hash map. The function then creates one new D* instance for each of these candidates, tells each new D* instance about all of the nodes and edges in the current visibility graph, sets the goal of each new D* instance to its corresponding candidate, and adds each new D* instance to the hash map. The function `Search_All_D*_Graphs()` on line 7 has each D* instance in the hash map recalculate those shortest paths that need to be updated. Finally, the function `Build_Candidate_Graph_From_D*_Graphs()` on line 8 iterates through the hash map of D* instances and records the length of the shortest path from each candidate in the graph to the goal candidate of the current D* instance.

One final point to note is that when whatever "solver" our implementation is currently using returns an ordered set of candidates to visit, we can use the hash map of D* instances to fill in a path that visits these candidates in order. Each D* instance keeps track of the most recently calculated shortest path to its goal candidate. Therefore, to find the shortest path from one candidate to a second candidate in the ordered set, we simply need to take the D* instance whose goal is the second candidate and read off the path from the first candidate to the second candidate. Our implementation of the function `Fill_In_Path()` in Figure 3.8 uses this method of filling in a path.

Now that we understand our implementation's overall approach to extracting candidate graphs, we are ready to see how our implementation produces the set of obstacles for the path planner.

5.2.1 Obstacle Extraction

This thesis introduces a new method for extracting obstacles from line feature SLAM maps. Many path planners require as input a list of obstacles that are specified as polygons or, in the most restrictive cases, rectangles. Our method of extracting obstacles builds a separate rectangle around each line in the map. Figure 5.4c shows what these rectangles would look like for the map in Figure 5.4a. The "padding distance" for a line feature is either calculated as a function of the line's uncertainty (covariance) or is set to a fixed distance in order to speed up obstacle extraction.

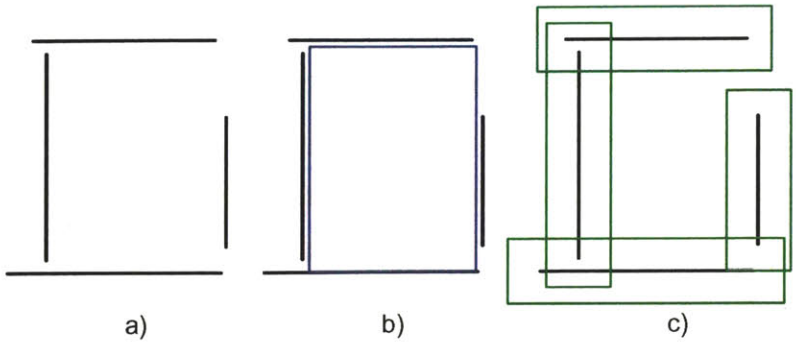


Figure 5.4 Methods of Extracting Obstacles from a Line Map

In order to see why this method of extracting obstacles is correct, it is helpful to consider a second method of extracting obstacles that is obvious but ultimately makes less sense. This second method tries to fit polygons to the line features in the map in order to extract obstacles. Figure 5.4b shows the obstacle that this method produces for the map in Figure 5.4a. The motivation for this approach is that the positions of the lines in the map are uncertain, and the robot may not have seen the full extent of these lines yet. As a result, the robot should try to use some a priori notion of what typical objects in the environment look like to guess what lines come together to form impenetrable polygons, in order to avoid planning a path through the center of an obstacle.

The reason why this second method is worse than the method we use in our implementation is that this second method reads extra information into the map. The robot's environment might look exactly like the map in Figure 5.4a. In fact, the map in Figure 5.4a is what the SLAM algorithm has determined is the most likely layout of the environment. And the SLAM algorithm should be responsible for estimating what the environment looks like. If we want to take into account a priori knowledge about what a typical environment looks like, this knowledge should be placed in the SLAM algorithm, not the obstacle extraction method. While it is true that the SLAM algorithm is not certain of the positions of the map lines, the algorithm keeps track of its uncertainty in the covariance matrix. Therefore in the obstacle extraction method advocated by this thesis, we scale the size of each line's rectangle so that it bounds that line's uncertainty. As a result, any lines that intersect in the real world and form an impenetrable corner should have intersecting rectangles. The path planner then should not consider any paths that pass through spaces that are corners in the real world, because the rectangular obstacles

should overlap at these corners. For example, in Figure 5.4c, all of the gaps between the map lines have been closed by the rectangles except for the gap in the upper right. This gap might correspond to a gap in the real world, such as a doorway.

In the end, therefore, the method of extracting obstacles that is most faithful to the map's estimate of the location of line features is to build a rectangle around each line feature. This method is the method our implementation uses. However, a more fundamental issue with extracting obstacles from a feature map is that, as we mentioned in Chapter 2, such maps do not exhaustively keep track of every point a robot's sensors have seen. For example, if a robot is building a line feature map and it senses a curved obstacle, the robot may not extract any line features where the curved obstacle is. In this case, the curved obstacle would not show up in the map; therefore, the path planner might end up choosing a path that goes through the obstacle. As we discuss in Section 5.3, our implementation prevents the robot from colliding with obstacles by running a low level obstacle avoidance process while the robot is moving. Yet even though the robot does not run into obstacles, the path planner is still not guaranteed to plan shortest paths through the environment that avoid all obstacles using a feature map. In our experiments, most of the environments had flat walls as obstacles; therefore, the SLAM algorithm mapped the majority of the obstacles. However, in general when path planning, it may make more sense to use a map representation that guarantees that any objects that a robot senses will appear in the map. Some examples of these types of map representations are occupancy grid maps and scan-matched maps. A number of exploration methods build occupancy grid maps and feature maps simultaneously [2] [9]. The robot uses the occupancy grid map for path planning and the feature map for localization. Using an occupancy grid map for path planning in this way might be a better solution than extracting rectangular obstacles from a feature map. Unfortunately, we did not have time in this thesis to try building both a feature map and an occupancy grid map simultaneously.

5.3 Executing a Path

The final component of our implementation left to explain is the component that commands the robot to follow the path that the exploration path planner computes. This

component continuously takes as input the robot's current position and the path outputted by the exploration path planner, and it continuously outputs commands that will make the robot move. In Figure 5.1, the component is the module labeled "Controller." In the pseudo-code in Figure 5.2, the component implements the function `Execute_Segment_of_Path()`.

We implement this component as a waypoint controller. In other words, the path that the exploration path planner outputs is an ordered set of waypoints. If the robot knew about all of the obstacles in its vicinity when it planned this path, then the robot should be able to drive in a straight line between any two consecutive waypoints in this path without running into any objects. A waypoint controller constantly takes the robot's current position and the next waypoint in the ordered set of waypoints and outputs commands that will cause the robot to move towards this next waypoint. Once the robot arrives in the vicinity of the waypoint, the waypoint controller removes the waypoint from the ordered set and drives towards the next waypoint in the ordered set. The waypoint controller that our implementation uses outputs commands in the form of desired translational and rotational velocities for the robot. The low level software on the robot then causes the robot to move at these desired velocities. The waypoint controller in our implementation computes these velocities so that when the robot turns, the robot follows a smooth curve. Turning by following a smooth curve is desirable because doing so cuts down on the robot's odometry error. The waypoint controller in our implementation also constantly reads the robot's sensor data and steers the robot away from any obstacles that the sensors detect. Having the waypoint controller perform obstacle avoidance is important, for as we discussed in Section 5.2.1, the exploration path planner in our implementation is not guaranteed to plan paths that do not run into obstacles.

Although our implementation has a waypoint controller, we drove the robot by hand in the experiments whose results we present in Chapter 6. The main reason for driving the robot by hand instead of using the waypoint controller was that the visibility graph path planner plans paths that are composed of straight lines. And in order to test how efficient the paths that various observation planning methods compute are, we need the robot to follow these paths exactly. However, the waypoint controller in our

implementation has the robot turn by following smooth curves. Because we did not have time to change this feature of the waypoint controller, we drove the robot by hand in our experiments in order to make the robot follow straight line paths.

5.4 Example

Now that we have explained how all of the modules of our experimental system work, we walk through a simple example that shows all of the modules working together. In this example, we focus on what the inputs and outputs of each module look like. We also give a rough sketch of the calculations the candidate identification and scoring module and the candidate graph extraction module perform. We do not focus on the internal workings of the SLAM module or the "solver" module because the calculations these two modules perform are somewhat involved. In Chapter 4 we went over examples that illustrate the internal workings of the S-TSP solver our implementation uses.

In this example, we examine two moments in the robot's exploration of an environment using the finite horizon continuous observation planning approach with a six meter receding horizon[13]. Figure 5.5 depicts the map the SLAM algorithm outputs at these two moments. In the figure, the triangle represents the map's estimate of the robot's position and heading. The lines are line features in the map. In Figure 5.5a, the robot is just about to move around a wall and see what is on the other side. In Figure 5.5b, the robot has gotten around this wall and has mapped the doorway of an office to the left. We first examine how the experimental system computes an exploration path from scratch for the situation in Figure 5.5a. We then examine how the experimental system computes an exploration path for the situation in Figure 5.5b by making use of results from the previous path computation.

[13] The maps in this example are simplified versions of maps the robot built while exploring the NE43Floor8 environment in simulation. Appendix A shows what this environment really looks like.

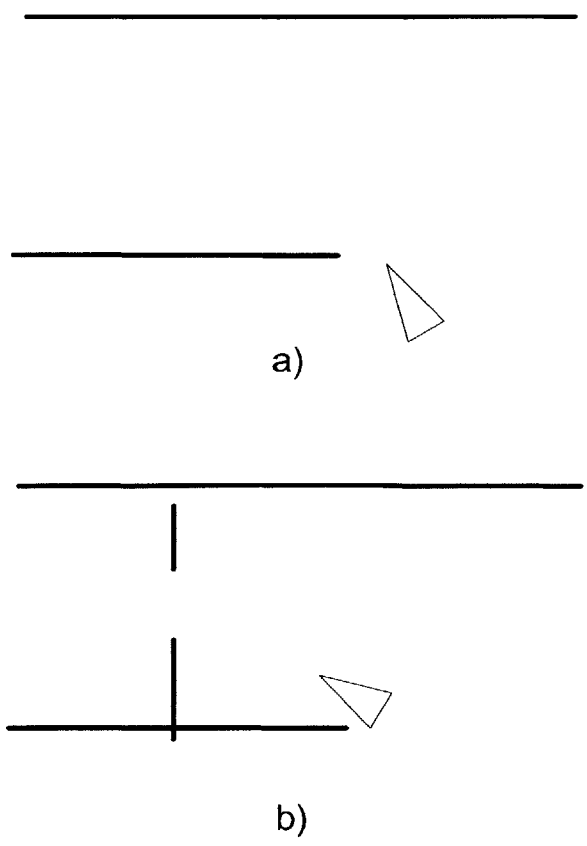


Figure 5.5 Maps from the First Moment (a) and Second Moment (b)

Once the experimental system has computed the map in Figure 5.5a, the first step the system takes in computing an exploration path from scratch is to identify and score candidate observation points in the map. Recall from the description of the Newman, Bosse, and Leonard exploration method in Chapter 2 that each line feature produces a candidate at both of its ends. Figure 5.6 depicts the candidates for the map in Figure 5.5a. The candidates are labeled with "cN," where N is a number. Because candidates c3 and c4 have a score of zero, we do not draw these two candidates in subsequent figures.

Figure 5.6 depicts how the candidate identification and scoring module scores candidate c2. The eight small unlabeled circles in the figure are candidate c2's sample points. The final score of a candidate is the number of the candidate's sample points that are valid divided by the total number of the candidate's sample points. A sample point is invalid if line features block the line of sight between the sample point and the candidate. The three sample points at the top of the figure are invalid for this reason. A sample

point is also invalid if there is a pebble within a given threshold distance of the sample point that has a clear line of sight to the sample point. The candidate identification and scoring algorithm keeps track of pebbles to mark where the robot has been. Figure 5.6 represents pebbles as pentagons. In the figure, the two sample points that are connected to pebbles by solid lines are invalid for this reason. The three sample points that are connected to pebbles by dashed lines are beyond the threshold distance of all pebbles; therefore, these three sample points are still valid. Because there are three valid sample points and eight total sample points for candidate c2, the utility of candidate c2 is 3/8. The utility of each candidate is shown to the upper left of the candidate in Figure 5.6.

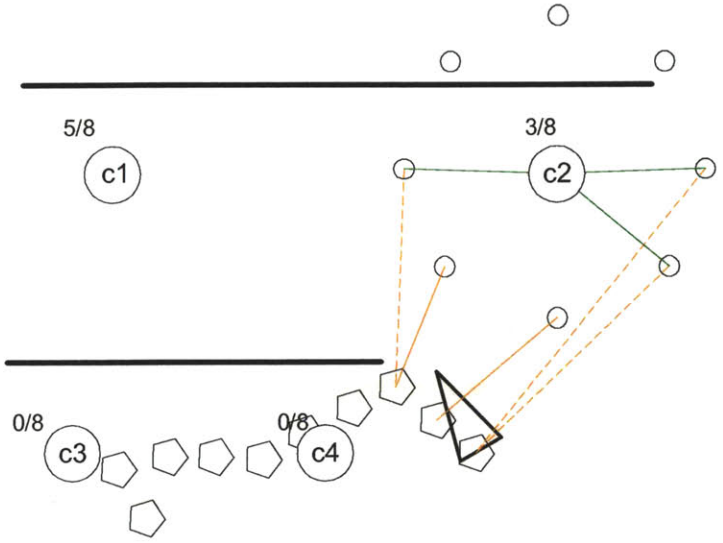


Figure 5.6 Candidate Identification and Scoring for the First Moment

Once the experimental system has identified and scored the candidates, the system passes the name, position, and utility of each candidate to the module that extracts the candidate graph. In addition, the system passes this module the map depicted in Figure 5.5a. The first thing the candidate graph extraction module does is extract obstacles from this map (line 1 of the pseudo-code in Figure 5.3). Figure 5.7a shows the two obstacles that the module extracts from the map. The module then builds a visibility graph over these two obstacles, the candidate positions, and the current robot position (line 2 in Figure 5.3). Figure 5.7b shows the visibility graph laid on top of the obstacles, the candidates, and the robot's current position. Note that because there is an edge between

every pair of vertices in the visibility graph as long as the edge does not pass through an obstacle, the edges of each obstacle are included as edges in the visibility graph.

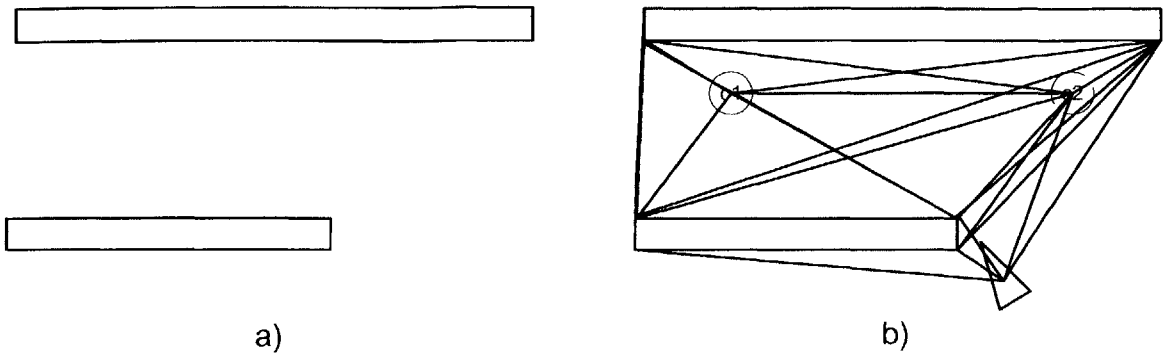


Figure 5.7 Obstacles (a) and the Visibility Graph (b) for the First Moment

Because we are assuming that we are computing the exploration path from scratch, we do not have any visibility graphs from prior path computations. As a result, line 3 of the pseudo-code in Figure 5.3 returns an empty graph. In addition, there are no D* instances in the hash map; therefore, lines 4 and 5 of the pseudo-code in Figure 5.3 do nothing. Therefore, the next thing the module does is create two instances of the D* search for the visibility graph (line 6 of Figure 5.3). One D* instance has candidate c1 as its search goal, and the other D* instance has candidate c2 as its search goal. The module then runs both D* instances (line 7 in Figure 5.3). Each D* instance searches for the shortest path from each vertex in the visibility graph to that instance's goal. Figure 5.8a shows these shortest paths for the D* instance whose goal is candidate c1, and Figure 5.8b shows the shortest paths for the D* instance whose goal is candidate c2. These shortest paths are the bold lines with the arrows at the ends in the figure; the dotted lines are the edges of the visibility graph.

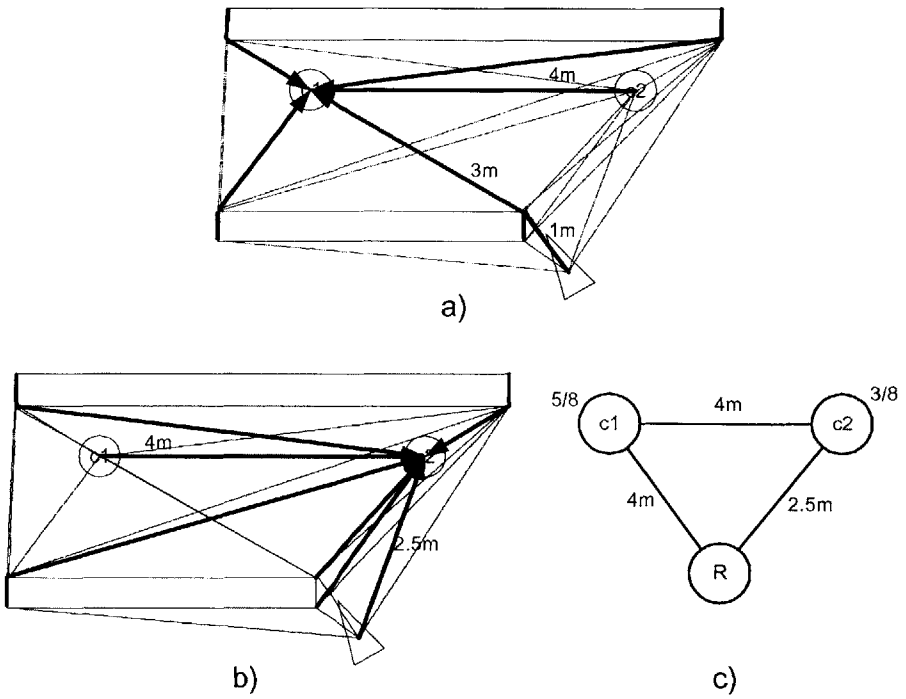


Figure 5.8 Extracting the Candidate Graph from D* Instances

Next the module builds the candidate graph using the shortest paths computed by the two D* instances (line 8 of Figure 5.3). Figure 5.8c shows this candidate graph. The node labeled "R" in Figure 5.8c represents the robot. Notice that the utility of each candidate also appears in the graph. In order to find the cost for the edge from R to c1 in the candidate graph, the module picks out the D* instance that has c1 as its goal from the hash map. This D* instance has already calculated shortest path from the robot to candidate c1; therefore, the module can just read off the length of this path and use it as the cost of the edge. In Figure 5.8a, we see that the shortest path from the robot to candidate c1 is made up of a one meter leg and a three meter leg. Therefore, the edge from R to c1 in the candidate graph is four meters. Similarly, because the shortest path from candidate c2 to candidate c1 in Figure 5.8a is four meters, the edge from c1 to c2 in the candidate graph is also four meters (we do not show the lengths of any other paths in Figure 5.8a because we do not need those lengths to build the candidate graph; the D* instance knows those lengths, however). In order to find the cost of the edge in the candidate graph from R to c2, the module must examine the other D* instance. Looking

at Figure 5.8b, we see that the shortest path from the robot to candidate c2 is 2.5 meters. Therefore, the cost of the corresponding edge in the candidate graph is 2.5 meters.

Once the candidate graph extraction module has built the candidate graph, the module stores the visibility graph (line 9 in Figure 5.3) and passes the candidate graph to the "solver" module[14]. In this case the "solver" module is an S-TSP solver, because the robot is using finite horizon continuous observation planning with a six meter receding horizon. The "solver" module then runs the S-TSP solver on the candidate graph and passes the resulting path back to the candidate graph extraction module. For this example, the S-TSP solver returns the path <c1>. The finite horizon path has the robot visit c1 because within six meters, the robot can only make it to one candidate and candidate c1 has the higher utility of the two candidates.

When the candidate graph extraction module receives this path, it uses the D* instances to fill in the details between consecutive candidates in the path (the function `Fill_In_Path()` on line 3 of the pseudo-code in Figure 3.8). To fill in the details of the path <c1>, the module must fill in the details of the shortest path between the current robot position and c1. The module therefore finds the D* instance that has candidate c1 as its goal and examines the saved shortest path from the robot to candidate c1. Figure 5.9 shows this shortest path by itself. As we can see in the figure, the shortest path goes from the robot to a corner that we label v1 of the lower obstacle, and then to candidate c1. The filled in path is therefore <v1, c1>. The candidate graph extraction module then passes this filled in path to the controller module. In other words, the filled in path is the exploration path that our system computes for the map in Figure 5.5.

[14] When it is using an S-TSP or TSP solver, the "solver" module must first convert the undirected candidate graph to a directed graph in order to find a path that does not have the robot return to its start point. We described how to perform this conversion in Chapter 2.

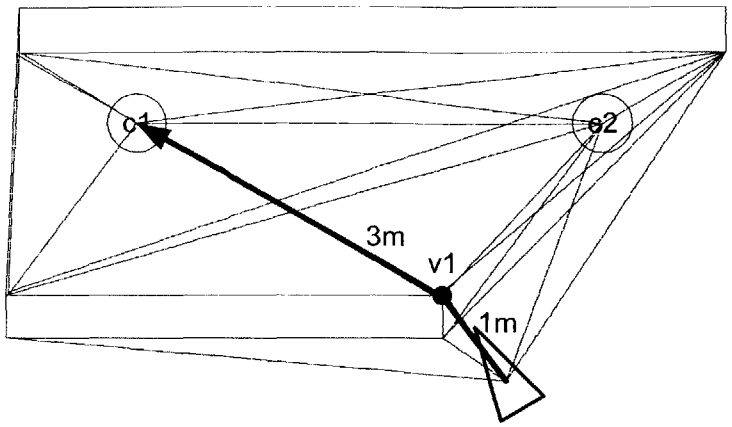


Figure 5.9 Exploration Path for First Moment

We do not go over how the controller computes commands to drive the robot along the exploration path here because the calculations are somewhat involved, and in our experiments we drove the robot by hand. However, we will assume that the robot drives straight towards v1, turns at v1, and then drives straight towards c1. Figure 5.5b shows the robot's map right after the robot makes its turn. We now describe how the robot calculates the exploration path for the situation that Figure 5.5b depicts.

The SLAM module passes the map in Figure 5.5b to the candidate identification scoring module. Figure 5.10 shows the candidates and utilities that this module generates for the map. As we mentioned earlier, the figure does not show the two candidates for the lower horizontal line because they both have a utility of zero. In addition, the figure only shows one of the lower vertical line's candidates, candidate c5, because the other candidate has a utility of zero. Figure 5.10 also shows how the module scores candidate c2 for this situation. Candidate scoring proceeds the same as before. The figure displays the utility the module calculates for each candidate to the upper left of the candidate.

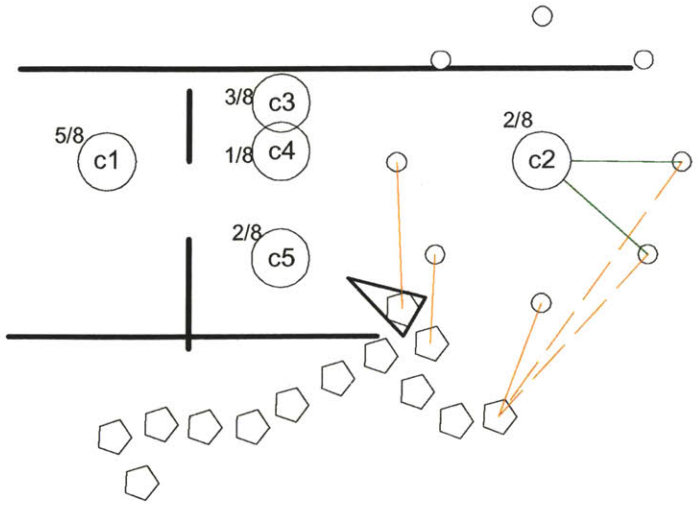


Figure 5.10 Candidate Identification and Scoring for the Second Moment

The candidate graph extraction module takes the names, positions, and utilities of these candidates as input along with the map in Figure 5.5b. The module then extracts obstacles from this map and builds the visibility graph depicted in Figure 5.11. Note that module builds this visibility graph from scratch once again, since there are currently no incremental approaches to building visibility graphs. No candidates were removed since the last time the exploration path was calculated; therefore, the module does not need to remove any D* instances from the hash map. The next thing the module does then is update the two existing D* instances.

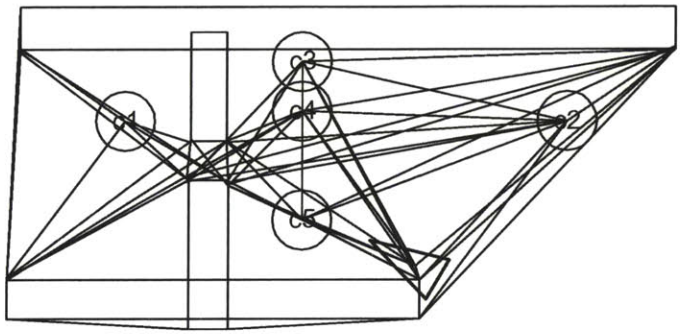


Figure 5.11 Visibility Graph for the Second Moment

In order to update the existing D* instances, the module compares the last visibility graph with the current visibility graph. The two D* instances are told about every edge that has been added, removed, or has changed its cost and every vertex that

has been added or removed from the last visibility graph. The module is then finished updating the existing D* instances. Next, because there are no D* instances with candidate c3, c4, or c5 as its goal, the module creates three new D* instances to search the most recently built visibility graph. The module sets each of these three D* instances to have as a goal candidates c3, c4, and c5 respectively. Then, the module runs all of the D* instances. The three new D* instances compute the shortest paths to their goal from scratch. The two existing D* instances, however, use the information about what has changed in the visibility graph to detect which of their previously computed shortest paths might possibly have been affected. The two existing D* instances then only recalculate these paths. Figure 5.12a once again shows the shortest paths to candidate c1 that the corresponding D* instance found the first time the robot calculated the exploration path. Figure 5.12b shows the new shortest paths to candidate c1 that the D* instance finds this time. Figure 5.12c shows which shortest paths are the same between Figures 5.12a and 5.12b. Although we do not cover the details of what the D* instance must and must not recalculate in order to produce the paths in Figure 5.12b, we can intuit from the fact that almost half of the paths do not change between Figure 5.12a and Figure 5.12b that it requires much less work to incrementally update the paths in Figure 5.12a than it does to find the shortest paths in Figure 5.12b from scratch.

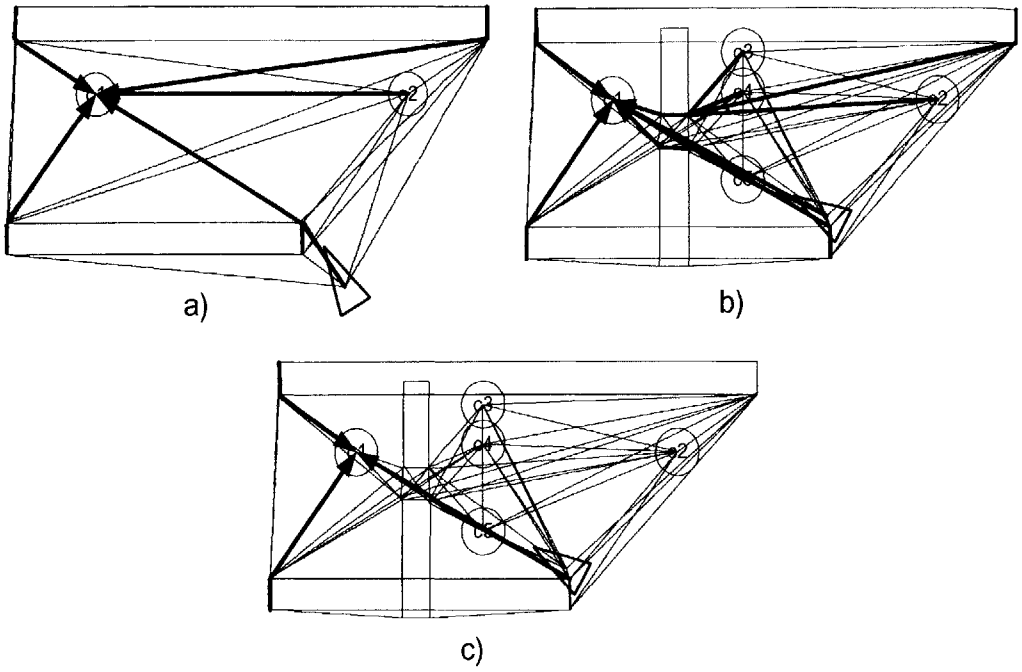


Figure 5.12 Searching the Visibility Graph Incrementally

The module next builds the candidate graph shown in Figure 5.13. In order to find the cost of an edge from ci to cj in the candidate graph, the module finds the D* instance that has cj as its goal. The module then gets the shortest path from ci to cj from the D* instance and uses the length of this path as the edge in the candidate graph, just as before. In Figure 5.13 we only show the costs of the edges that were in the previous candidate graph because the graph would be unreadable if we included every edge cost. Once the module computes this candidate graph, it saves the current visibility graph and passes the candidate graph to the "solver" module. Because the experimental system is using the receding horizon continuous observation planning method, the "solver" module once again runs the S-TSP solver on the candidate graph with a six meter horizon. The path visiting candidates with the highest total utility that is under six meters is <c5, c4, c3, c1>; therefore, this path is the output of the S-TSP solver. The "solver" module then passes this path to the candidate graph extraction module.

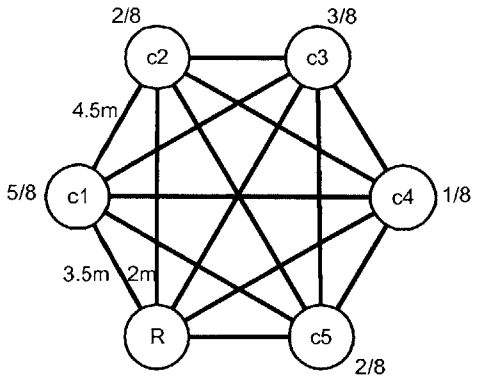


Figure 5.13 Candidate Graph for the Second Moment

The candidate graph extraction module fills in the path <c5, c4, c3, c1> by first finding the D* instance that has c5 as its goal. The module then examines the shortest path between the robot's current position and c5. As we can see from the visibility graph in Figure 5.11, the shortest path from the robot's current position to c5 goes directly from the robot to c5. Therefore, we do not add any thing to the front of the finite horizon path. The module next needs to fill in the finite horizon path between c5 and c4. Taking the D* instance that has c4 as its goal, the module finds that the shortest path from c5 to c4 goes directly to c4. Therefore, the module once again does nothing to the finite horizon path. The module next fills in the finite horizon path between c4 and c3 and finds that the shortest path from c4 to c3 goes directly to c3. Finally, the module fills in the finite horizon path from c3 to c1. Looking at the D* instance that has c1 as its goal, the module finds that the shortest path between c3 and c1 first visits two vertices of an obstacle. In Figure 5.13, we label these vertices v2 and v3. The final exploration path therefore is <c5, c4, c3, v2, v3, c1>. Figure 5.14 shows this path. The candidate graph extraction module then passes this exploration path to the controller module, and the robot begins to execute the path.

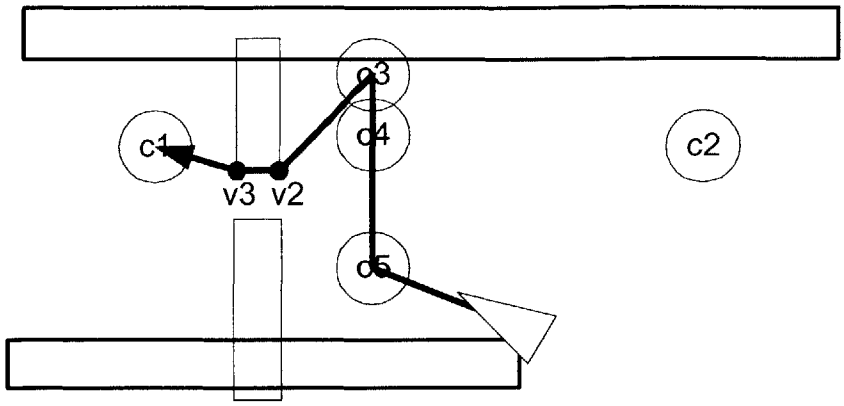


Figure 5.14 Final Exploration Path for the Second Moment

We have now seen how the modules in the experimental system work together to calculate an exploration path from a SLAM map for two moments in a robot's exploration mission. However, it remains to be seen how well these exploration paths guide a robot to build a map of its environment. In Chapter 6, we present the results of the experiments we performed to test the system that this chapter has described.

6 Testing and Evaluation

The main claim of this thesis is that for many different exploration frameworks (for example, the Gonzalez-Banos and Latombe framework, frontier-based frameworks, and the Leonard, Bosse, Newman framework, all of which were explained in Chapter 2) and in many different environments, using the finite horizon approach to perform continuous observation planning will significantly improve the efficiency of autonomous exploration. Finite horizon observation planning methods should especially improve the efficiency of exploration when the robot knows the large scale structure of the environment early on in the exploration.

Our analysis in Chapter 3 of how well the finite horizon approach should perform for all exploration frameworks in general provides some support for this claim. However, in this analysis we found that the exploration frameworks studied are not similar enough for us to come to definite conclusions that are applicable to all frameworks, by reasoning about their common properties. Therefore, the only way for us to convincingly support the main claim of this thesis is to perform experiments and see how well the finite horizon approach works for each individual exploration framework.

In this chapter we present the results of experiments that tested how well the finite horizon observation planning approach performs with the Newman, Bosse, and Leonard feature-based exploration method, described in Chapter 2. In addition, we analyze these results and show that they support our above claim.

We only had time in this thesis to implement and test the finite horizon approach with this particular exploration method. Unfortunately, it is probably not possible to generalize the results of experiments that use a particular exploration method to all exploration methods. From the general analysis of the finite horizon approach that we performed in Chapter 3 we know that, how well the finite horizon approach performs in a particular situation, depends on how often the robot gets caught making large unrewarded sacrifices, versus how often the robot is able to execute its finite horizon path far enough to do something more efficient than a greedy method would. How well the robot is able

to execute its planned path depends on how the candidates are arranged and how they change as the robot explores. Furthermore, the candidate dynamics can be substantially different for different exploration frameworks and environments. Therefore, while these experiments support our claim for the particular case of the Newman, Bosse, and Leonard exploration method, we can only make a few speculations as to how the results might apply to other exploration methods. In the future, more experiments should be performed to verify that the finite horizon approach performs well with other exploration methods.

We explain what exactly we mean by the efficiency of exploration in Section 6.2. Then, in Section 6.3 we describe the details of the experimental method. In Section 6.4 we present the results of the experiments and discuss what we can learn from these results. Finally, in Section 6.5, we briefly summarize our conclusions from these experiments.

6.1 Overview of Experiments

In these experiments, we had a robot explore and map one environment in the real world and six environments in simulation. These environments were chosen to be representative of areas we are likely to explore. In each environment we compared using the receding horizon, fixed horizon, full horizon, and greedy methods to select waypoints for the robot. For the finite horizon trials, we also experimented with using two or three different horizon lengths.

These tests suggest that in the Newman, Bosse, and Leonard strategy, the candidates change so much that using the finite horizon approach with a long horizon or the full horizon approach to select waypoints performs similar to using the greedy method in many environments. However, if the robot starts off knowing the large-scale structure of its environment, then the robot can greatly improve the efficiency of the paths it executes by using the finite horizon approach with a long horizon. This improvement occurs because, if the robot knows where all of the large obstacles and interesting regions to explore are, then the robot can plan a rough path that is globally efficient. While the small-scale details of this path will change as the robot fills in the holes in its map, the overall shape (and therefore the efficiency) of the path will remain the same. In addition, these experiments suggest that if we were to make some changes

to the Newman, Bosse, and Leonard strategy, then using the finite horizon approach with a *short* horizon may allow the robot to explore its environment much more thoroughly than with other observation planning methods. These changes would need to make the way the Newman, Bosse, and Leonard method places candidates less conservative by not placing candidates in areas that the robot is almost certain have been explored.

6.2 Metrics for Evaluating the Quality of Exploration

In order to compare how well greedy, full horizon, and finite horizon methods selected candidates in our experiments, we must develop somewhat objective measures of how well the robot explored its environment. In our general analysis of the strengths and weaknesses of various observation planning methods in Chapter 3, we equate how well the robot explores to how efficiently it explores. In particular, we measure how efficiently a robot explores by calculating the total utility of the candidates that the robot visits within a given distance. Yet ultimately we do not care about the utilities of the candidates that the robot visited; we care only about how well the robot added information to its map during exploration. The only reasons to look at the candidates and their utilities at all is to understand why the robot executed the path it did and to try to generalize the results to other exploration frameworks. Later in this section, however, we see that it is unclear what we even mean by the total utility of the visited candidates. Instead, it makes sense to establish metrics that more directly measure how well the robot added information to its map.

The goals of the exploration framework we used for testing determine what it means to explore an environment "well." In Chapter 2 we describe this exploration framework, and our explanation of the reasoning behind the Newman, Bosse, and Leonard candidate identification and scoring strategy tells us what these goals are. The first and most explicit goal of our exploration framework is to extract as many new features as possible and thereby expand the coverage of the map. A second and less apparent goal of the framework is to try to ensure that all features that could have been extracted in the area explored were extracted, and extracted fully. We can see that extracting features fully is an objective of the Newman, Bosse, and Leonard strategy in the fact that each feature produces its own candidate observation points in places that

enable the robot to observe the full extent of the feature. We can see that trying to extract every feature that can be extracted in regions the robot has visited is a goal in the fact that the strategy keeps track of how well the robot has explored the region near a feature, by measuring the density of surrounding features and how close the robot has been to the feature before.

The final goal of the framework is to make sure that the robot stays well localized by preventing the robot from leaving sight of features it has seen before. The fact that known features produce candidates near themselves provides evidence for this final goal, in addition to the second goal. Finally, the additional goal of this thesis is to perform this exploration efficiently. Recall that because we can use any quantity as the cost of a path between two candidates, an efficient path might be a path that is short in distance or time, or a path that requires little energy to follow. In this exploration framework, we measure efficiency by the distance that the robot travels.

We would like to measure objectively how well the path that the robot executes during exploration fulfills each of these goals. In order to score how well the robot achieves the first goal of extracting as many features as possible, we measure the combined length of all the line features the robot extracts. We do not simply count the number of line features that the robot extracts, for extracting a long line feature expands the map's coverage more than extracting a short line.

In order to measure how well the robot achieves the second goal, for each line feature in the robot's final map we calculate the ratio of the length of the line in the map to the actual length of the line in the environment. If a line is longer in the map than it really is in the environment, then the line is treated as having been mapped fully and the ratio is one. If part of the line was outside of the area that the robot explored, then we consider the actual length of the line to be the length of the line that was within the area the robot explored. We roughly define the area the robot explored to be the part of the environment that the robot's sensors have scanned well enough to extract features. In practice we delineate the region the robot has explored by forming a closed region that surrounds all of the features in the map and that is created by connecting the endpoints of line features. When faced with a choice on how to connect these endpoints, we attempt to connect the endpoints so that regions that the robot's sensor should have seen are

included, and regions that the robot's sensor should not have seen are excluded. The dotted lines in Figure 6.1 show this region for the final map of the first greedy trial in NE43Floor8. Additionally, we keep track of all lines in the area that the robot explored that were not extracted. These lines have a ratio of zero. For each map we then compute a histogram for these line feature ratios. Appendix B contains these histograms.

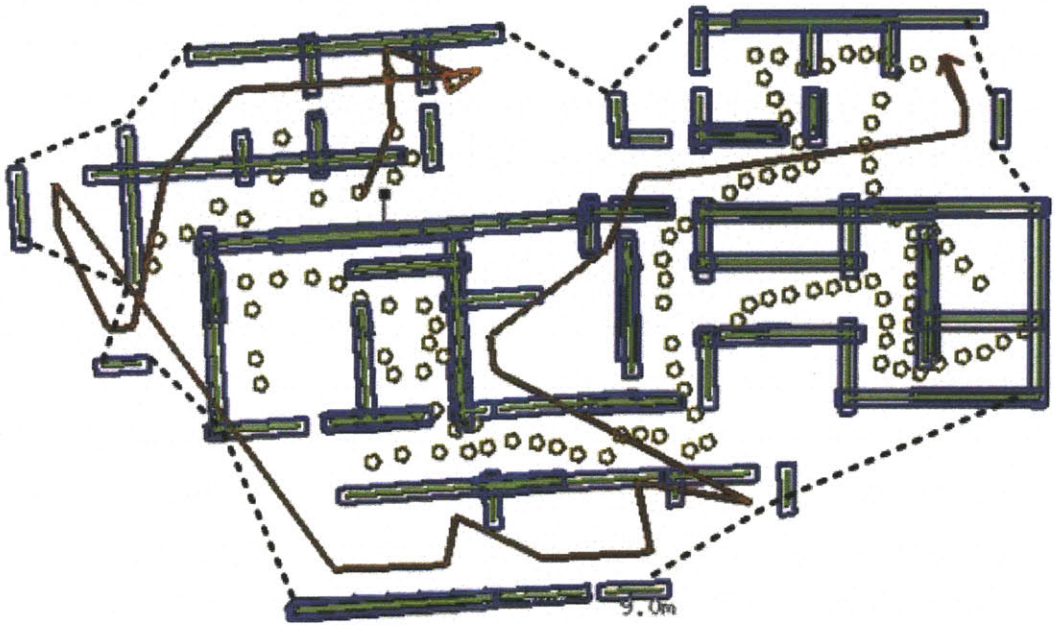


Figure 6.1 Explored Region for the Final Map of the First Greedy Trial in NE43Floor8

To measure the third goal of the exploration framework, we count the number of times the robot strays out of sight of all line features it knows about in a given exploration run. However, since most of the environments we used in our experiments were indoors and had a relatively dense distribution of features, the robot almost never left sight of known line features during these trials. This measure is therefore zero or close to zero in all of the trials, and we do not use it in our analysis.

Finally, we measure the efficiency of the robot's path indirectly by running each trial in a given environment until the robot has traveled some set distance. Then, by calculating any of the three measures just described for a set of trials, we can compare how efficient the robot was at achieving the corresponding goal in each trial in the set.

We now have three different dimensions along which to measure how well the robot achieved its goals for exploration. Yet at times, we might want to combine these

three separate values into an overall score of how effective the robot's exploration was in a particular trial. However, there is no obvious way to combine our three measures into a single measure. One option is to note that the way the Newman, Bosse, and Leonard strategy identifies and scores candidates takes into account all of the goals of the exploration framework. Therefore, one additional reason to look at the total utility of the candidates that the robot visits in a trial is to use this total as an overall score of how well the robot achieved the goals of exploration. Obviously this is a very rough overall score, however; therefore, it should not be taken too seriously.

A big problem with using the total utility of the visited candidates for an overall measure is that, as we will see shortly, it is difficult to define what we really mean by the total utility. Therefore, instead of using the total utility of the visited candidates, in environments where it is feasible we explore until there are no more candidates left in the map. We then use the distance the robot traveled as the overall score. This score is still based on the fact that the candidates identify places that need to be explored in order to fulfill the goals of the exploration framework. However, this score does not require us to use the concept of the total utility of the visited candidates. Once again, however, we do not give this measure too much weight. We only use the score in order to get a rough overall idea of how well a waypoint selection strategy worked in a given trial.

It is tempting to look only at the candidate observation points and their utilities when evaluating how well the robot selected waypoints during exploration, yet it makes more sense to focus on methods that explicitly measure how much information the robot added to its map. The temptation is that by looking solely at the sequence of candidates the robot decides to visit in its path and how many of those candidates the robot actually gets to, we can abstract away from our evaluation messy details such as whether or not the SLAM algorithm was able to extract all of the features it should have at those candidate points, or whether or not the candidate identification and scoring method placed the candidates in the most informative places. We would therefore be looking at the observation planning method in isolation, and we could then generalize the results of our experiments to say how well finite horizon methods work for any SLAM algorithm and candidate identification and scoring method.

Yet is not easy to isolate the performance of the observation planning method from the details of the exploration framework used for testing. The performance of the observation planning methods depends solely on how the candidate observation points appear, disappear, move, and change score during exploration. Yet for a given environment, these candidate dynamics vary greatly given different SLAM algorithms and candidate identification and scoring strategies. For example, in the Gonzalez-Banos and Latombe method for identifying and scoring candidates [18] described in Chapter 2, it is not possible for the robot to plan a path and then later find out that an obstacle blocks part of that path. In the Newman, Bosse, and Leonard method this situation happens frequently, however. In addition, while the Newman, Bosse, and Leonard method places candidates only near known features, most grid-based methods [10] [53] place candidates at the edge of the area that the robot's sensors have seen, regardless of whether or not there are any obstacles nearby. Perhaps the only major similarity between candidate identification and scoring methods is that when the robot sees a new room, all methods create new candidates somewhere in the vicinity of this new room. And even if we made small adjustments to the Newman, Bosse, and Leonard method, we probably could not apply our results from the original method to the adjusted version, because small alterations in how the candidates change eventually cause large differences in the paths taken.

Another problem is that, as we have noted, it is unclear what we really mean by the total utility of the visited candidates. Exploration does not quite proceed in the abstracted manner that we assumed in the general analysis in Chapter 3. An important difference is that the utility of a candidate often drops to or near zero as the robot approaches that candidate. The reason that the utility drops is that the robot makes observations all the time, not just at the candidate observation points. Therefore, the robot sees most of the area it is supposed to see from a candidate before it ever gets there. This drop in utility means that most of the candidates that the robot ever reaches will have a low utility. In addition, the robot will not get to some candidates at all because if a candidate's utility drops as the robot heads towards the candidate, then the next time the robot computes its path it will often find that it is no longer worth it to visit that candidate.

Therefore, simply keeping track of the utility of the candidates that the robot actually visited does not provide much information about how well the observation planning method is working. The problem is that the robot's choosing and driving towards a candidate caused the candidate's utility to drop and therefore added information to the map. However, the total utility of the candidates the robot has visited does not reflect this success of the observation planning method. Instead of adding the utility of the candidate when the robot reaches the candidate to the total utility, we could add the utility of the candidate as soon as the robot starts heading towards it. Yet it is also possible for the utility of a candidate to change for reasons that have little to do with the robot heading towards it. For example, even in the absence of new observations the SLAM algorithm might perform additional processing and update the map. This change in the position of features in the map might then change the utility of the candidate.

If we are trying to gauge how much information heading towards a candidate added to the map, we would be better off looking at direct measures of the information that exploration adds to the map. Therefore, our primary metrics for evaluating an exploration run are the three measures we introduced initially.

6.3 Methods

The experiments we performed tested how well various observation planning strategies performed in one real world environment and six simulated environments. Specifically we tested the receding horizon, fixed horizon, full horizon, and greedy strategies. The greedy strategy simply drove to the closest candidate to the robot. The strategy therefore did not take into account candidate utilities.

In the experiments we performed in the real world, we used an iRobot B21 robot equipped with a SICK LMS laser scanner to explore the environment. We did not use the B21's sonar array in the experiments at all. We controlled and monitored the robot with a laptop over a wireless Ethernet connection.

In the trials we performed in simulation, the only code that was different than what we used in the real world trials was the code that handles sending commands to the robot and receiving sensor readings. The SLAM code, candidate identification and scoring code, observation planning code, and user interface code were all the same as

what we used in the real environment. The code that handles robot input and output simulates a robot equipped with a laser scanner and an odometer driving on land. The error in the laser range values simulates the error of a typical SICK laser scanner. The odometry error simulates that of the B21 robot.

An important detail is that in all trials we drove the robot by hand; we did not use a waypoint controller to automatically drive the robot. The main reason for this decision was that sometimes it took awhile to compute a path by solving the S-TSP or TSP. In these cases, when the path was finally output, it was no longer up to date with the map or the set of candidates. Therefore, we would have to stop the robot and let the observation planner catch up with the current state of exploration. In the future we hope that we will have algorithms for solving the S-TSP and TSP that a waypoint controller can use to control a robot in real time. However, finding such algorithms is outside the scope of this thesis.

When driving the robot, we tried to follow the path output by the path planner as best as possible. Our simple path planner used a visibility graph [51]; therefore, these paths always consisted of straight lines connecting candidates and the corners of obstacles. Our driving procedure was to turn the robot at its current waypoint to face its next waypoint and then drive the robot in a straight line to that next waypoint. Whenever the path changed while the robot was driving, we would stop the robot, turn the robot to face its new waypoint, and continue driving in a straight line. After selecting waypoints and computing a path to visit those waypoints, the robot waited four seconds before selecting waypoints again. This pause allowed the program performing the observation planning to bring its internal map and set of candidates up to date. The pause also allowed us to drive the robot a short distance along its path.

Unfortunately, turning in place results in more odometry error than turning in an arced path. Worse odometry error results in worse localization and mapping, and as a result, our driving method affected the quality of the maps we built. Yet there is nothing that prevents these observation planning strategies from using more sophisticated path planners that plan paths that keep odometry error to a minimum [28] [33]. We simply did not have such path planners available. As a result, it is possible that some of the apparent failures of various observation planning strategies in these experiments were not due to a

weakness of the strategy but the quality of the planned path. However, it is likely that the path planner affected all observation planning methods roughly equally and, therefore, does not skew our evaluation of the relative performance of these methods.

Appendix A shows the floor plan of the building that we explored in our real world experiments and the six environments that we explored in simulation. The grid cells in the images of the simulated environments are one meter by one meter. We tried to make these environments typical of the types of environment a robot is likely to explore. The environment that we explored in the real world experiments is the fourth floor of two connected buildings, Building 34 and Building 36, on MIT campus. The part of these buildings that we explored consists of the elevator lobby and the hallway outside of a conference room. The simulated environments NE43Floor8 and Building10Floor1 are based on the floor plans of other real buildings on MIT campus. NE43Floor8 is a floor of a typical office building, and Building10Floor1 is a floor of a large building with big hallways and lobbies. The simulated environments 15by15Room and 25by45Room are fictional rooms. They were created to test the theory that finite horizon methods should perform better in buildings that have large rooms from which it is easy to see the other rooms. RandomRocks and StructuredRocks are fictional outdoor environments. The one by one meter squares are meant to represent rocks. We used a random number generator to place the rocks in RandomRocks, but not in StructuredRocks.

In each environment we usually tried two or three different horizon distances for the finite horizon observation planning methods. The first horizon distance we tried was always the distance we thought would perform best in the environment. We determined this distance by estimating the horizon length the robot could plan a path for such that on average this path would not go outside of the room the robot was currently in. We then would try double this horizon distance and half this horizon distance.

In each environment we also chose some set distance for the robot to explore, before we stopped the robot and tallied the three measures of performance described in Section 6.2. We chose this distance by having a robot explore the environment use the receding horizon approach with a horizon length that we thought would perform well. The distance it takes the robot to explore roughly two-thirds of the environments is the

distance that we used. We use the SLAM estimate of the robot's pose in order to keep track of how far the robot has traveled.

In calculating the first two measures of exploration performance, we measure the length of line features in the robot's final map by hand. If the SLAM algorithm has extracted any line features that are longer than the corresponding line in the actual environment, we use the length of the actual line for the length of the line in the map. In addition, when the SLAM algorithm creates multiple lines that correspond to a single line in the actual environment, we only measure the longest of these redundant lines.

Finally, out of interest we also keep track of the number of candidates that the robot actually reaches and the total utility of the candidates the robot "visits." We calculate this total utility by adding the utility of the candidate that the robot is currently headed towards to the total when the robot is within a meter of the candidate.

6.4 Results and Analysis

The following tables quantify how well the four observation planning methods performed in our simulated environments[15]. In addition, Appendix B contains histograms showing the distribution of what fraction of each line feature was extracted in each trial.

[15] There are no measures of performance for the StructuredRocks environment because we were unable to explore the environment using the finite horizon or full horizon methods. The problem is that the TSP solver gets stuck after a short period of time in the environment for some reason. We therefore are able to plan an initial path using the finite horizon or the full horizon methods, but not much else. Nevertheless, these initial paths provide important evidence for the claim that if the robot knows the structure of its initial environment, then it will be possible to plan a much more efficient path using the finite horizon or full horizon methods than using the greedy method. We will elaborate on this issue later in this section.

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
RH	15m	113m				
Greedy		111.835m	140.59m	0.6643	84	≈14.3
RH	30m	110.455m	106.13m	0.8527	32	7.4444370
RH	7.5m	112.546m	122.83m	0.9016	38	8.4444390
RH	15m (trial 2)	105.373	147.04m	0.7929	42	9.1111050
FH	15m	110.203m	127.77m	0.7601	58	9.7777700
FH	8m	110.526m	122.4m	0.7092	80	13.9999910
Greedy (trial 2)		110.267m	138.33m	0.8618	79	17.4444

RH = Receding Horizon, FH = Fixed Horizon

Table 6.1 Performance of Observation Planning Methods in NE43Floor8

We were unable to test using TSP methods in NE43Floor8 because for some reason the TSP solver crashed midway through the exploration every time.

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
RH	15m	162.502m	144.66m	0.6783	34	11.1111060
Greedy		163.215m	147.72m	0.8047	52	8.1111
Full Horizon		165.828m	154.77m	0.8431	46	9.33333
FH	15m	163.022m	134.697m	0.6710	63	12.3333220
RH	30m	164.627m	174.6m	0.7373	63	13.9999910

RH = Receding Horizon, FH = Fixed Horizon

Table 6.2 Performance of Observation Planning Methods in Building10Floor1

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
Greedy		32.6346m	71.92m	0.8580		
RH	15m	32.4045m	73.24m	0.8810		
FH	15m	33.9783m	71.38m	0.8900		
FH	8m	32.5902m	66.69m	0.8460	10	1.7777760

RH = Receding Horizon, FH = Fixed Horizon

Table 6.3 Performance of Observation Planning Methods in 15by15Room

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
RH	15m	91.3566m				
Greedy		97.4837m				
FH	15m	92.7731m	143.76m	0.8654		
RH	15m (trial 2)	92.7213m	117.54m	0.6403	30	10.5555530
Greedy (trial 2)		96.5831m	135.7m	0.7736	38	6.22222
FH	15m (trial 2)	93.1311m	144.93m	0.6297		
Full Horizon		92.6588m	102.68m	0.8425	37	5.11111

RH = Receding Horizon, FH = Fixed Horizon

Table 6.4 Performance of Observation Planning Methods in 25by45Room

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
RH	20m	50.3775m	33.361m	0.9019		
Greedy		52.4176m	35.54m	0.9353		
FH	8m	51.7187m	33.265m	0.9784	71	27.1111050

RH = Receding Horizon, FH = Fixed Horizon

Table 6.5 Performance of Observation Planning Methods in RandomRocks

Method	Horizon	Distance Traveled	Total Length of Line Features Mapped	Average Fraction of Line Extracted	# Waypoints Reached	Total Utility Collected
Greedy		56.5687m	42.91m	0.9524		
RH	15m	45.6891m	17.51m	0.9714		
RH	30m	50.7897m	26.03m	0.9408		

RH = Receding Horizon, FH = Fixed Horizon

Table 6.6 Performance of Observation Planning Methods in Real Buildings 34 and 36

Our conclusion from these experiments is that in many environments, the set of candidate observation points changes too much with our particular exploration framework for the robot to plan ahead effectively using finite horizon or full horizon methods. Yet if the environment is open enough that the robot can quickly discern the overall structure of its surroundings (or if the robot starts out knowing this overall structure somehow), then finite horizon methods perform much more efficiently than greedy methods at filling in the remaining holes in the map. In addition, in all environments it may be possible to plan ahead using the finite horizon approach with a very short horizon in order to improve the thoroughness of the exploration. In order to achieve this improvement, however, we would also need to make a few improvements to the Newman, Bosse, and Leonard strategy for identifying and scoring candidates.

Section 6.4.1 describes how the experiments suggest these conclusions and examines the details of some important trials. The remaining sections are devoted to the individual environments, and discuss interesting or unusual trials in these environments.

6.4.1 Overall Analysis of Experiments

Examining these results, we find that no observation planning methods stand out as being consistently dominant in any measure of performance for all environments. Furthermore, the greedy method performs reasonably well in all trials. It appears, then, that we cannot conclude from these experiments that finite horizon methods for planning observations offer large improvements over greedy methods in all environments. In order to understand why it is not always useful for the robot to plan ahead and in what situations the finite horizon approach should perform efficiently, we must analyze the set

of candidates that determine the robot's path and how the candidates change during exploration.

The most obvious reason that none of the observation planning methods were able to consistently outperform the greedy approach is that often the candidates changed too much during exploration to allow planning ahead to be useful. In all environments, it was rare for the robot to reach more than one observation point in its plan before the set of candidates changed enough to cause the plan to change. As a result, the robot usually ended up executing a path that was the concatenation of the first steps of paths planned to be efficient. This executed path itself is not guaranteed to be efficient at all, however. Because the robot never did what it was planning on doing, even if the executed path had ended up being extremely good, this success would have been largely due to luck. And on average, most executed paths ended up performing about as well as the greedy path.

That the robot's path changes frequently was somewhat expected. In Chapter 3 we predicted that visiting previously unexplored areas, candidate interactions, and SLAM updates to the estimated state of the world will cause the set of candidates to change as the robot explores. In particular, we predict that visiting previously unexplored areas might cause the set of candidates to change enough to cause the robot's planned path to change every time the robot reaches a candidate. Therefore, we worry most about the effects of visiting previously unexplored areas.

Indeed, the set of candidates does change significantly each time the robot reaches a candidate. What is surprising is that in the Newman, Bosse, and Leonard exploration method, SLAM updates and candidate interactions tend to cause changes to the set of candidates that is just as drastic as visiting unexplored areas. Because SLAM updates and candidate interactions can occur even more frequently than every time the robot reaches a candidate, the robot's planned path often changes even before the robot reaches the next candidate in its path. Some specific reasons that the set of candidates changes in the Newman, Bosse, and Leonard method are:

1. New lines are extracted. These new lines produce new candidates.
2. Existing lines lengthen or new lines appear at a position which blocks the line of sight between two candidates. Such a blockage drastically increases the

length of the shortest path between the two candidates and is basically equivalent to the candidates suddenly jumping apart.

3. The map changes its estimate of the location of lines, causing candidates to disappear, reappear, or change score. Because the SLAM algorithm continuously updates its position estimates of all line features, this type of change causes the most frequent fluctuation. Figure 6.2 gives two examples of such situations. In the figure, large circles are candidates, small circles are sample points, and pentagons are pebbles which mark where the robot has been. Recall from Chapter 2 that the utility of a candidate is the fraction of the sample points that have a clear line of sight to the goal and that are not within a set range of a pebble with a clear line of sight. Figure 6.2b depicts a common situation in which two lines come together to form a corner, and their corresponding candidates overlap. The Newman, Bosse, and Leonard method considers these candidates redundant and removes one of them. If the lines ever separate slightly, however, their respective candidates no longer overlap and what was one candidate becomes two.

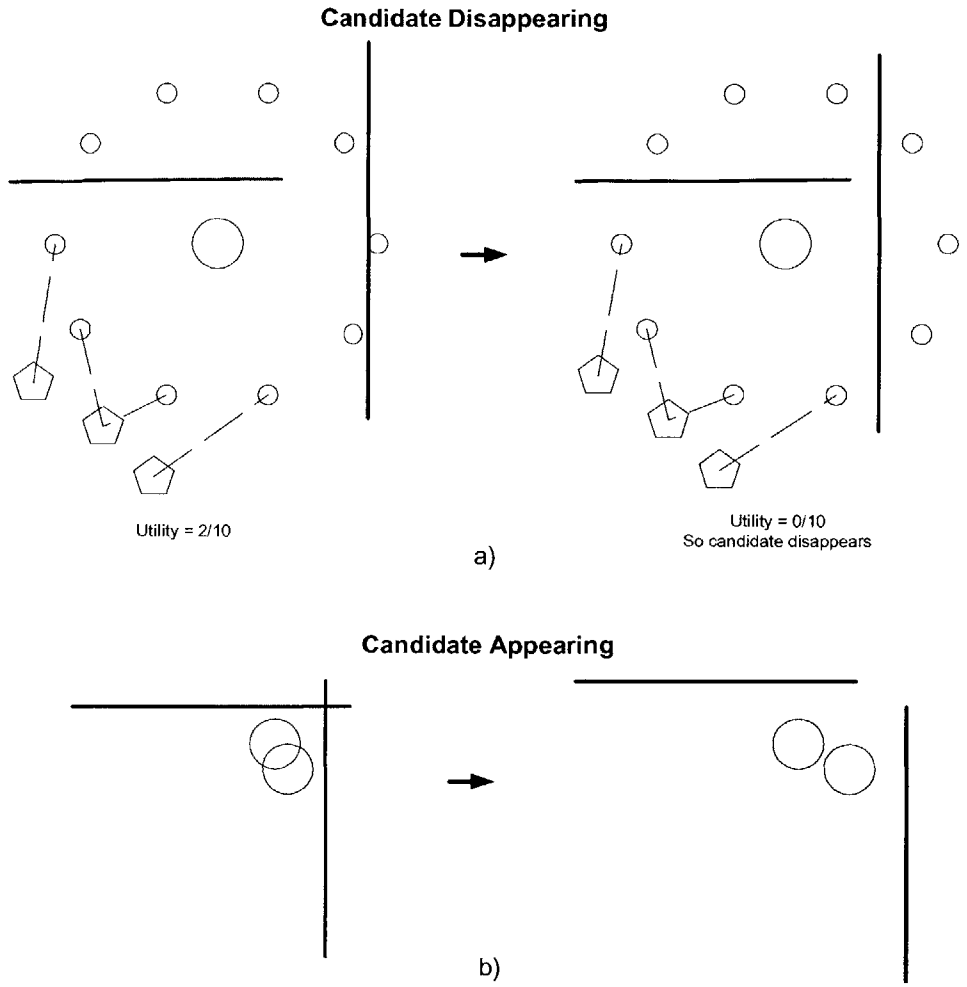


Figure 6.2 Candidates Disappearing and Appearing Because of Line Movement

4. As the robot approaches a candidate, the candidate's utility drops. As we mentioned in Section 6.2, this utility drop happens almost every time the robot heads towards a candidate. As the robot approaches a candidate, the robot sees more features in the surrounding area and places pebbles near the candidate. Both of these actions can cause the candidate's utility to decrease.
5. Visiting one candidate can cause another candidate's utility to decrease if the candidates are close together or if the robot's path passes by the other candidate. This type of candidate interaction also happens frequently.
6. Even if the robot is stationary, the SLAM algorithm can update the robot's estimated location and, therefore, make it look like the robot has moved. Although this motion does not cause the candidates to change, it can cause the

robot's plan to change. If the robot makes motions that are not along the path it has planned, then even if nothing else in the map changes, the best path might still change[16].

The screenshots in Figure 6.3 and Figure 6.4 show how the set of candidates typically changes during exploration. The robot is in environment NE43Floor8 and is selecting its waypoints using the receding horizon approach with a 15m horizon. In the screenshots, the robot is the triangle, the green lines are the line features in the robot's current map, and the blue boxes around the green lines are the obstacles that the path planner uses. The candidates are the circles, and each candidate's utility is displayed to the upper-right of the candidate. The candidates in the robot's current path are connected in order by a red line. In addition, the waypoint that the robot is currently headed towards is marked by a very small black square with a thin line sticking out of it (the direction of the line gives the desired heading at the waypoint). The current waypoint is not necessarily a candidate, for in addition to candidates, the robot's path contains corners of obstacles that the robot needs to drive to in order to avoid the obstacles. Finally, the small yellow pentagons are the pebbles that mark where the robot has been.

[16] In fact, as Figure 3.6 shows, the best path can change even if the robot follows its plan exactly if it is using the S-TSP with a receding horizon to plan.

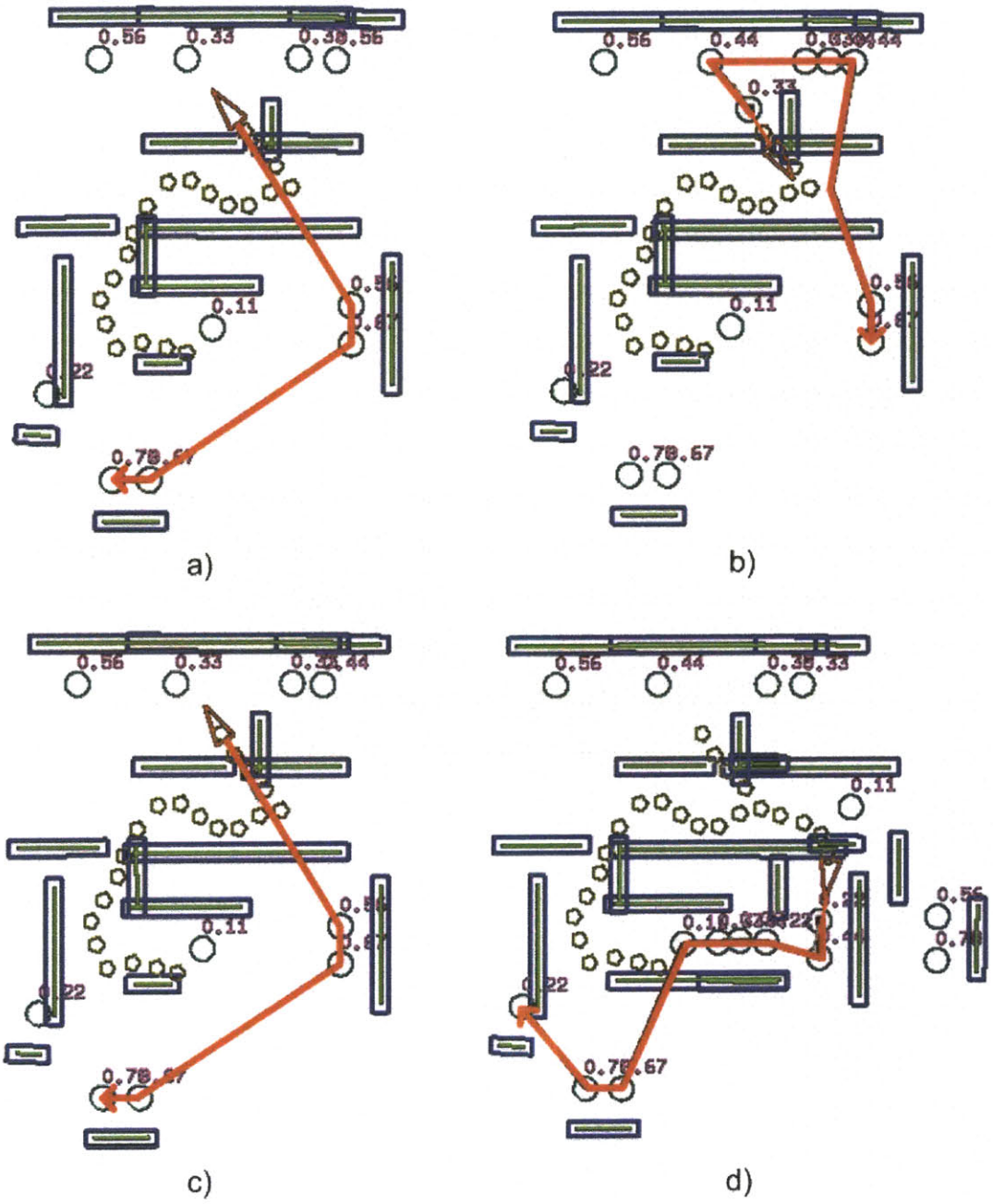


Figure 6.3 Beginning of the Second NE43Floor8 15m Receding Horizon Trial

In Figure 6.3a, the best path of 15m takes the robot out of the office it is currently in and into the elevator lobby below. Note that the horizontal line at the top of the map is actually four separate line features, as a result of the SLAM algorithm. Therefore there are four candidates along this line. The line on the far left produces only one candidate because it is too short to require two candidates. The long line in the middle produces the

middle two candidates. The line which ends immediately to the right of this middle line does not produce any candidates, for these candidates would be redundant with the middle line's candidates. Finally, the line furthest to the right produces one candidate.

In Figure 6.3b, the robot has turned clockwise to head down into the elevator lobby. As the robot turned, however, the line whose candidates were redundant with the long line grew enough to make one of its candidates no longer redundant. Therefore, in this screenshot, the highest utility 15m path now takes the robot back into the offices. Yet Figure 6.3c shows that after the robot gets to the first candidate in this path, the next candidate in the path drops in utility, because the robot has placed a pebble near this next candidate. In addition, the long line above the robot has lengthened slightly. As a result, the candidate that appeared in the last figure has once again become redundant. These changes make the best path switch back to being down in the elevator lobby.

Figure 6.3d shows the robot entering the elevator lobby. The robot sees a new horizontal line in front of it; therefore, the path changes slightly to observe this line. In Figure 6.4a the robot has reached the candidate it was heading to in Figure 6.3d and has seen a line at the bottom right of the map. Therefore, the best path changes once again to visit this line and also an isolated line on the right side of the map that was not previously in the path. The robot makes it to this point in Figure 6.4b, and since it plans for the best 15m path from this location, the robot changes its path slightly to include three new candidates that it can now reach. The robot then finally gets to visit three waypoints without changing its path, and in Figure 6.4c has planned a new 15m path. Upon getting to the first point in this new path, however, the robot finds itself out in the hallway again. Therefore in Figure 6.4d, the robot has planned a new 15m path, because it can once again reach the points in the offices it had passed up earlier.

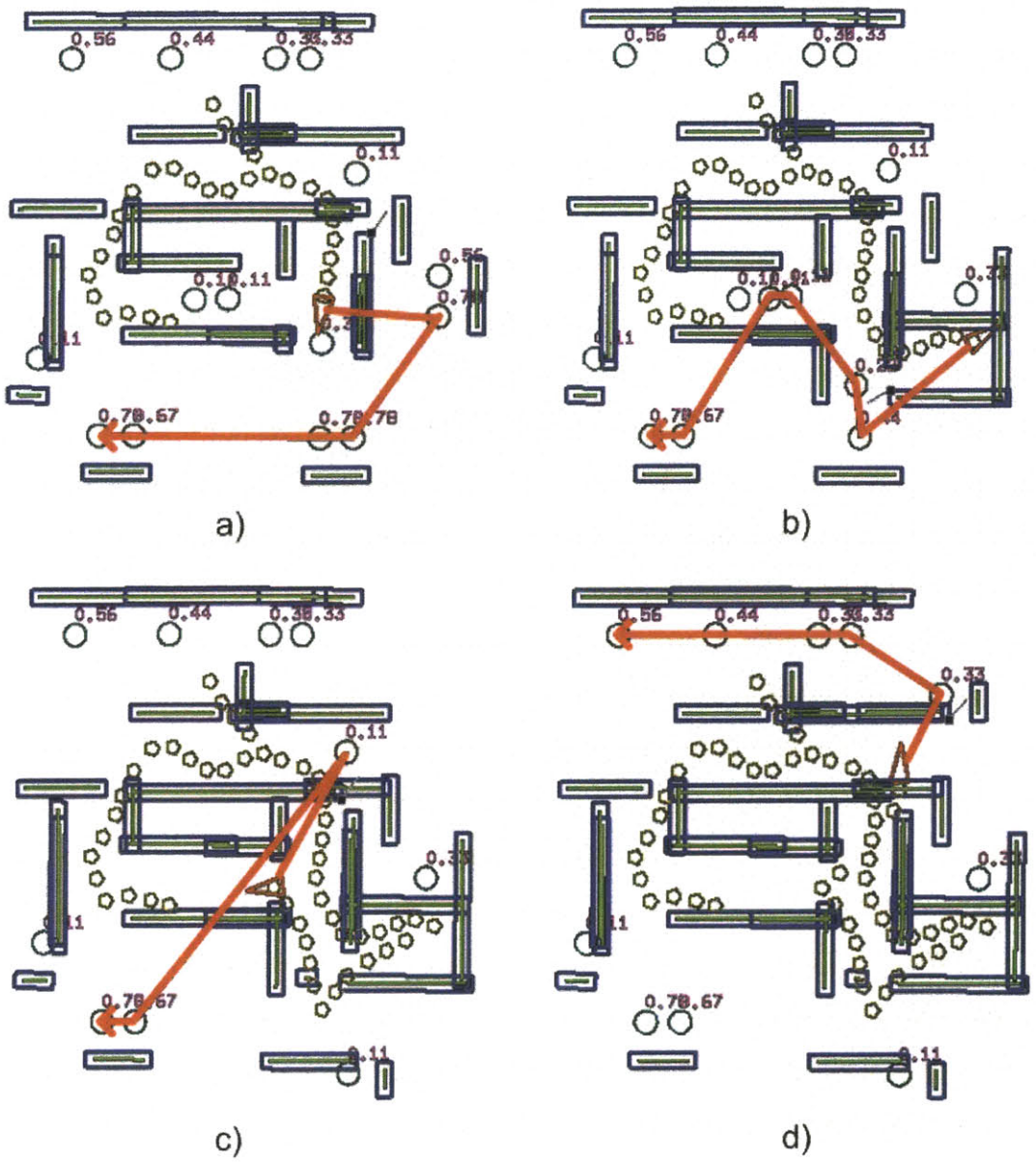


Figure 6.4 Beginning of the Second NE43Floor8 15m Receding Horizon Trial Continued

In this fragment of a trial, only once does the robot get to more than one candidate in its path without the path changing. This example is fairly representative of how exploration proceeded in most trials. In actuality, this example is a relatively good case because most of the path changes do not drastically alter the overall direction of the path.

Planning multiple candidates ahead is not entirely hopeless in our exploration framework, however. Two possible methods for combating the effects of the set of candidates changing frequently are to plan over a very short horizon using the finite

horizon approach, and to know the overall structure of the environment at the beginning of exploration. By using the finite horizon approach with a very short horizon, the robot's plans will not often extend beyond the room that the robot is in. As a result, the robot's path will not change as often from the robot entering an unexplored room and extracting a lot of new line features or from the robot finding out a wall blocks part of its path. By knowing the large-scale structure of the environment, the robot can plan paths that are much less likely to change drastically in overall shape.

We first examine how well planning over a short horizon performs. In indoor environments or environments with a high density of features, a robot should be able to execute many of its paths to completion, if it plans these paths using a short horizon. By "short" we mean on the order of the length of a room. The reason a robot should be able to execute more of these short-horizon paths is that in indoor environments, the walls of the room that the robot is in prevent the robot from seeing too much unexplored territory. Therefore, if a robot's planned path does not go outside of the room that the robot is in, then the robot will not extract new far away features that will drastically alter the robot's path unless the robot sees another room through a doorway. Setting the horizon at about the average room length makes it less likely that the robot's path will go outside of the current room. Using the fixed horizon method makes it even less likely that the robot will re-plan its path to be outside of the current room; therefore, paths calculated with short fixed horizons should be executed to completion even more often. In addition, planning paths on the order of a room length makes it less likely that these paths will be unexpectedly blocked by unmapped walls. Therefore, the first two reasons listed above for the set of candidates changing are less likely to occur when using a short horizon.

These short horizon paths that the robot executes should lead the robot to explore its local environment thoroughly before moving on to a new area. In fact, we noted in Chapter 3 that it is a general feature of using the finite horizon approach with any length horizon that the planned paths will tend to visit regions that have already been somewhat explored over visiting isolated and largely unexplored regions. This bias of finite horizon paths is bad for the goal of mapping as many new features as possible, but is good for the goals of mapping regions thoroughly and staying near known features. The reason for this bias is that areas that have not been explored much do not contain many features and,

therefore, candidates. On the other hand, areas that have been explored a moderate amount should be filled with features, and these features should produce many candidates. Therefore, even though the candidates in unexplored regions will probably have a higher utility then the candidates in explored regions, the sheer number of candidates in explored regions will make it more profitable for the robot to go to the explored region. Figure 6.5 is a screenshot of such a situation occurring, while the robot explores NE43Floor8,using the fixed horizon method with a 15m horizon. Instead of going to the less explored features on the right side of the map, the robot's path goes inside the more explored office above, because this area has more candidates that the robot can visit.

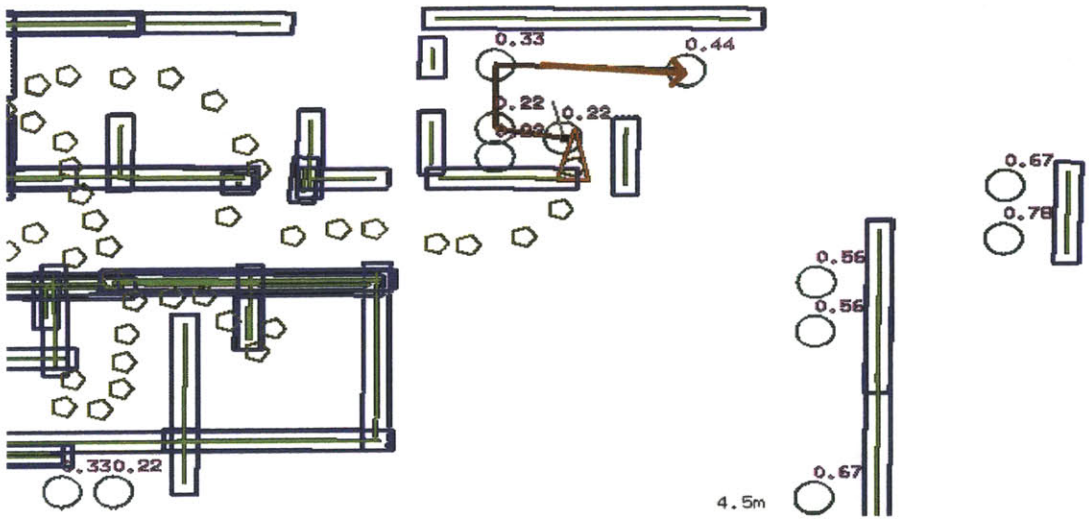


Figure 6.5 Visiting Explored Regions Over Unexplored Regions

When the robot plans a path over a short horizon, this bias ensures that the robot will finish exploring a local, partially explored area before moving on to a less explored area. The bias will cause the robot to choose a path that stays in an already explored region, and since the horizon is short, the robot will probably execute this path to completion. Therefore, even if the robot sees a new room through a doorway, the robot will stick to exploring the room it is in until most of the candidates in the room are gone. The result should be that the robot performs well in terms of the goal of thorough exploration.

The series of screenshots in Figure 6.6 provide an example of the robot being able to execute paths planned with a short horizon. The robot is exploring the environment Building10Floor1 using the fixed horizon method with a 15m horizon. In Figure 6.6a, the robot plans a path that actually visits two different rooms. The first room is very small and has been somewhat well explored. Therefore, it is not surprising that the robot's plan explores an additional room. In Figure 6.6b the robot has executed this path successfully and has reached the end of its 15m fixed horizon. In Figure 6.6c, the new 15m horizon has started and the robot's new path takes it into a room it has already explored a decent amount, as opposed to the less explored area on the right side of the map. All of the candidates in this new path are in the same room, and, therefore, the robot is able to execute this path, as shown in Figure 6.6d. Figure 6.6e shows the robot's path for the next horizon. The robot has chosen to explore the next most explored area in the map. All of the candidates in this path also are in the same room. In Figure 6.6f, the robot has reached most of these candidates. The robot took a slightly less efficient path than it initially planned to and was, therefore, unable to make it to the last candidate before reaching the end of the horizon.

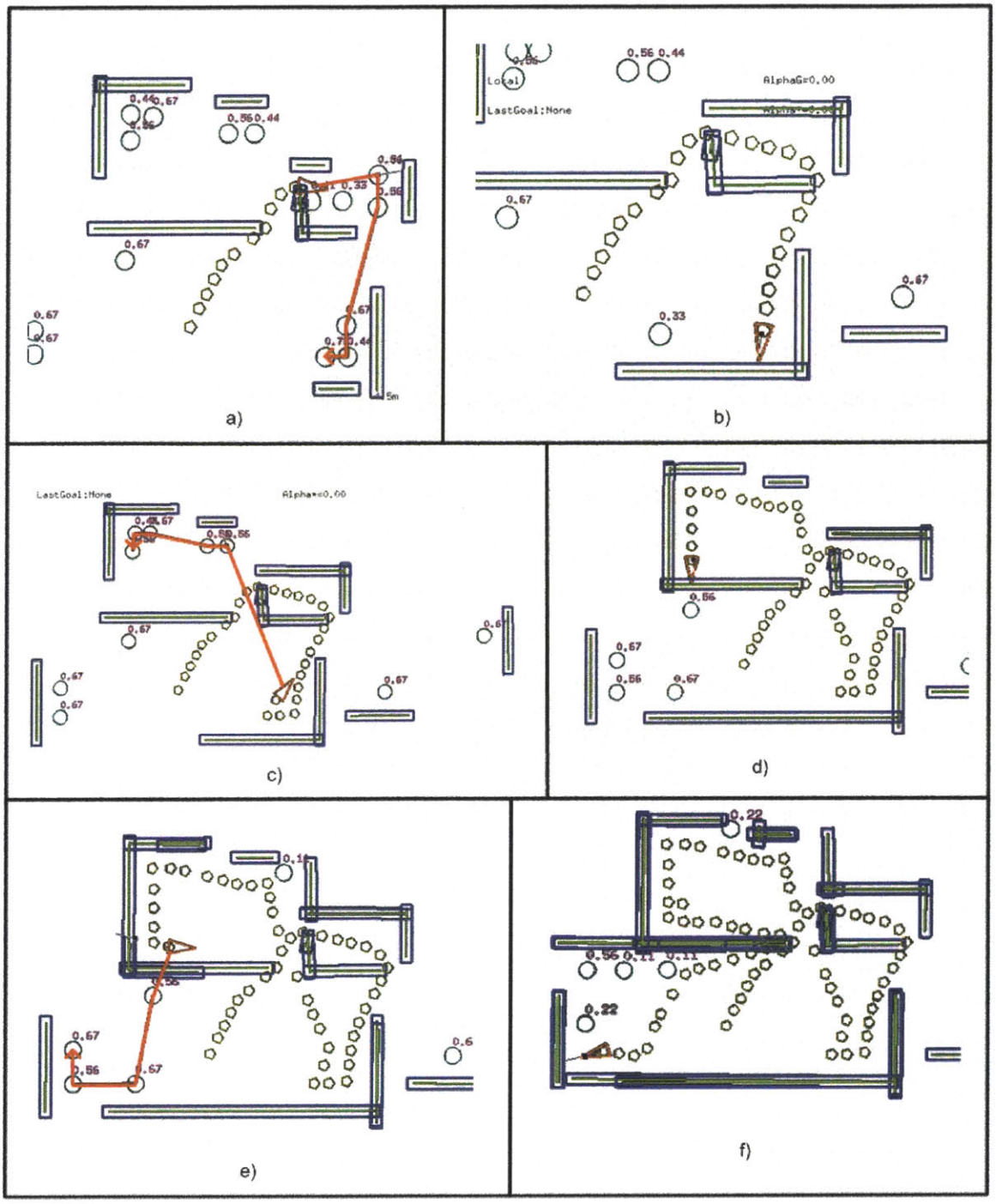


Figure 6.6 Screenshots of Exploration of Building10Floor1 with a 15m Fixed Horizon

Even though the robot executed its plans to completion and visited most of the waypoints in the rooms it explored as we expected, the results do not show this trial outperforming the other trials in Building10Floor1 in terms of thoroughness. In fact, the average fraction of line extracted in this trial is lowest of all Building10Floor1 trials. In

addition, in environment NE43Floor8, the trial using the fixed horizon method with a short horizon of 8m is the second worst in terms of fraction of line extracted (although the trial using a receding horizon of 7.5m in this environment performs the best in terms of fraction of line extracted). In actuality, the robot's exploration of its environment was fairly thoroughly in these two trials; it just did not get to visit one partially seen area. Figure 6.7a shows the map after the robot has finished exploring Building10Floor1 using the fixed horizon method with a 15m horizon. The robot filled in the regions it visited very well, but never got to explore the hallway at the bottom right of the map. The final map in Figure 6.7b shows a similar situation from the robot's exploration of NE43Floor8 using the S-TSP with a fixed horizon of 8m. The robot explored the places it got to thoroughly, but never visited the area in the bottom left corner of the map. Therefore it is misleading to say that the robot explored its environment less thoroughly in these two trials than the other trials in these environments.

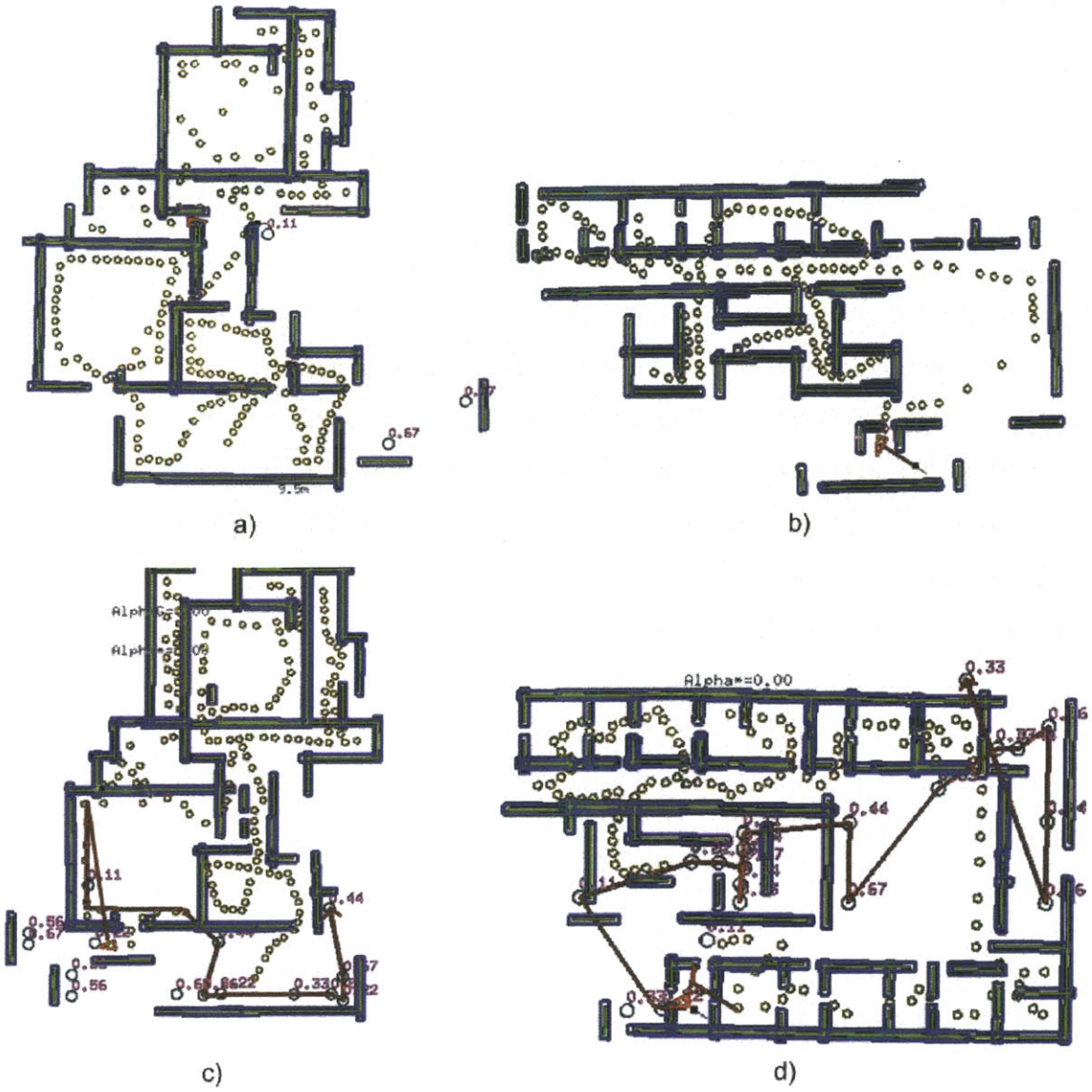


Figure 6.7 Final Maps of Short Horizon and Greedy Trials in Building10Floor1 and NE43Floor8

Nevertheless, even if we disregard the areas that the robot did not get to in these two trials, the robot did not explore its environment more thoroughly using a short horizon than it did using other methods in the same environments. For example, Figure 6.7c shows the final map after Building10Floor1 greedy trial, and Figure 6.7d shows the final map after the second NE43Floor8 greedy trial. The areas that the robot visited in these two trials are just as thoroughly explored as the areas the robot visited in the trials depicted in Figure 6.7a and Figure 6.7b. Therefore, it seems that a robot planning over a

short horizon does not explore its environment more thoroughly than a robot planning with other methods.

The main problem is that a robot often does not need to explore all of the candidates in its local area in order to explore that area thoroughly. In other words, even though planning over a short horizon produces paths that go to small clusters of candidates and visits every candidate in the cluster, visiting every candidate in a cluster is not necessary for the robot to explore the cluster's area thoroughly. Often all a robot needs to do to see every line feature in a room is to drive to some point inside the room and turn in a few different directions. For example, Figure 6.8 shows the robot traveling from one room into another room in NE43Floor8 during the second greedy trial. In Figure 6.8b the robot has just entered the new room and has already fully extracted each line in that room, with the exception of the line right above the robot. Nevertheless, there are still four candidates in the room for the robot to visit.

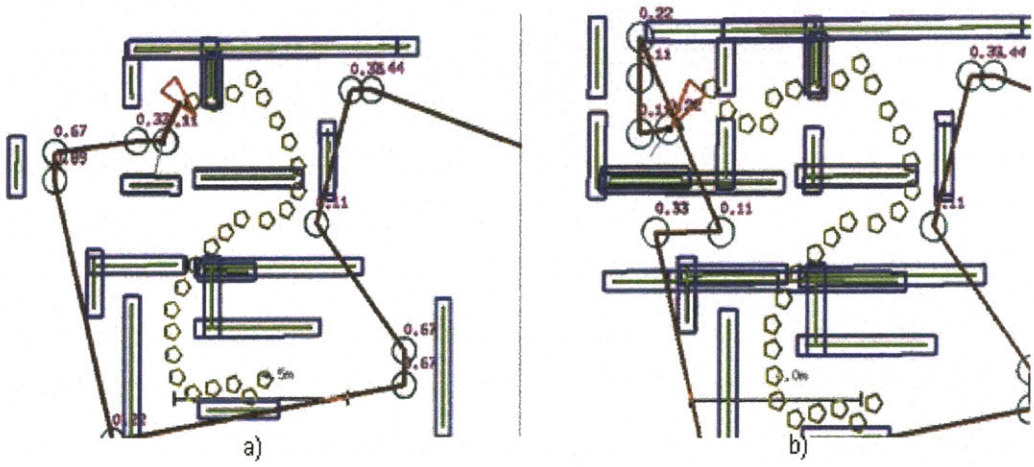


Figure 6.8 Candidates in a New Room While Exploring NE43Floor8

The reason that the robot often does not need to visit every observation point in an area is that the way the Newman, Bosse, and Leonard candidate identification and scoring method produces candidates is very conservative. The method places candidates right next to the ends of map lines so that it is almost guaranteed that the robot will see more of the line if the line extends further in reality. Yet in most cases the robot can see the full extent of a line from points much further away than these candidates. The only times the robot should need to go right up to the end of a map line is if the view of the

real world line is blocked from most angles by obstacles, or if the robot is in a narrow hallway. In a narrow hallway, the laser cannot extract the walls of the hallway beyond a certain distance in front of the robot, because the angle of incidence of the laser beam on the wall is too large. Therefore, the robot must move further along the map line in order to see parts of the corresponding wall beyond this distance.

In order to make it useful to plan over a short horizon and visit every candidate in a local region, we would need to make the candidate identification and scoring method less conservative. One way to make the method less conservative is to somehow keep track of what areas the robot's laser has and has not seen. The robot then would know whether or not an obstacle was blocking its view of the rest of the part of a line feature. Making such an improvement to the candidate identification and scoring method is beyond the scope of this thesis, however.

The second method of dealing with the fact that the set of candidates changes frequently is to learn the large scale structure of the environment early on during exploration. If the robot knows where most of the big obstacles and interesting places to map are, then the robot can plan an efficient path to visit all of these interesting places. This path will change slightly as the robot maps small details of the environment. However, the path should not change drastically, for the robot is not likely to find large obstacles blocking legs of its path or new regions of the map that it must divert the path to visit. In an extreme instance of this approach, the robot would have a perfect map of its environment except for a few small holes. The robot could then use the finite horizon or full horizon methods to plan an optimally efficient path through the environment to fill in these holes, and none of the map changes that resulted from visiting these unseen areas would alter the path.

The path that the robot executes when it knows the overall structure of its environment and plans a path with a long horizon should be much more efficient than the path the robot executes when the robot selects candidates to visit greedily. Here what we mean by efficient is mapping a greater length of line feature in a given distance. In fact, the only time we can feel certain that a path planned using the finite horizon or full horizon methods will be more efficient than the greedy path is when the robot knows the overall structure of its environment. The reason is that if the robot does not know the

large scale structure of its environment, then the robot can only take into account local concerns when choosing its path. Therefore, at best the path that the robot executes will be a sequence of locally efficient paths that meander randomly through the overall structure of the environment. Yet the large scale shape of the path will likely have the largest effect on the path's efficiency, and in this situation, the greedy method has just as good of a chance of selecting an efficient large scale path as the finite horizon or full horizon methods have.

The largest problem with this approach is finding a way for the robot to learn the overall structure of its environment. One solution to this problem is to know this structure before the robot begins exploring. The robot's map would be initialized to reflect this a priori knowledge. We might know the large-scale structure of the environment from overhead satellite images or other wide-area, low resolution mapping methods. Or perhaps the robot could explore the environment twice, the first time using an algorithm which tries to quickly build a rough, large-scale map, and the second time using a finite horizon method. In this second scenario, however, in order to evaluate how efficiently the robot explored its environment, we would have to look at the combined path that the robot took for both explorations.

A second way the robot could learn the overall shape of its environment quickly is if the environment is structured properly. The environment is structured properly if it is possible for the robot to see most of the large groups of features in the environment from the vicinity of the robot's initial location. We refer to this type of environment as an open environment. The most open environments are outdoor locations that are not densely packed with obstacles. Using the finite horizon method with a long horizon should cause the robot to explore these environments very efficiently. Indoor environments are usually not as open as outdoor environments, unless the robot is confined to mapping obstacles inside one large room. Yet planning over a long horizon might still perform well indoors if the building has large rooms with a lot of wide doorways. Having large rooms and wide doorways minimizes the density of the walls in indoor environments, and walls block the robot's view.

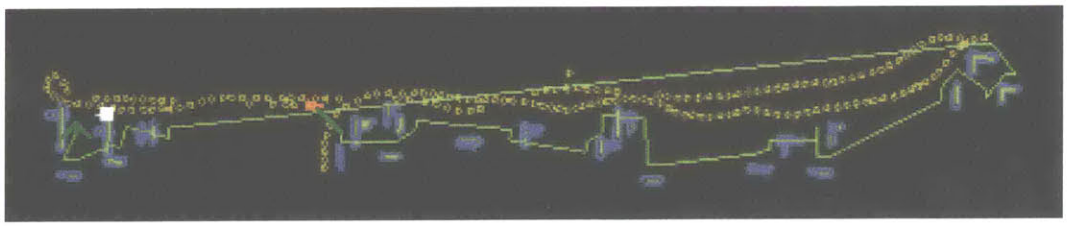
Another issue with these situations in which the robot starts out knowing the structure of its environment is that it is not obvious that finite horizon methods for

planning observations offer any benefit over the full horizon method. One advantage that finite horizon methods have is that they take into account the utility of candidates. Therefore, paths planned using finite horizon methods tend to avoid going to low utility candidates that will not add much to the map. If the robot's mission ends before the robot visits all of the candidates in the environment, maps built using finite horizon methods will probably have mapped a greater length of line than full horizon methods. In addition, often times these low utility candidates disappear when the robot steers near them. In these cases, the robot is better off if it does not explicitly plan to visit these candidates.

The StructuredRocks environment is a fairly open outdoor environment. However, the environment is too large and the rocks are too small for the robot to see the overall structure of the environment without moving. Therefore, before we let the robot explore, we quickly drove the robot through the environment so that it knew the general location of each cluster of rocks. Figure 6.9a shows the robot's map after this initial tour of the environment. The line of small yellow pentagons at the top of the map marks the path of the initial tour. Figure 6.9b shows the path planned using the full horizon method after the initial drive through of the environment. Figure 6.9c shows the greedy path for the exact same situation. On the global scale, the greedy path is much less efficient than the full horizon path. In fact, from these figures it looks as if the greedy path has to be about 25 percent longer than the full horizon path to visit all of the candidates. As the robot begins to explore, small-scale twists and turns of these two paths will change. However, we expect the overall order in which the paths visit the clusters of rocks to remain the same throughout the mission. Therefore, in the StructuredRocks environment, planning over a long horizon should drastically improve the efficiency of exploration. Unfortunately, we were unable to complete the exploration of this environment using the full horizon method because for the TSP solver always crashed midway through the mission for some reason.

a)

b)



c)

Figure 6.9 Initial Exploration Paths through the StructuredRocks Environment

Examining our results for indoor environments, we see that in the trial with the longest planning horizon (30 meters) in NE43Floor1, the robot performed very poorly in terms of total length of line mapped. However, in the trial with the longest planning horizon (30 meters) in Building10Floor1, the robot vastly outperformed all other methods. In keeping with our analysis, the rooms in are larger and the environment is generally more open in Building10Floor1 than in NE43Floor1. In order to see whether or not planning over a long horizon was responsible for the robot's exceptional performance in Building10Floor1, however, we must look at the paths that the robot planned during exploration in this trial.

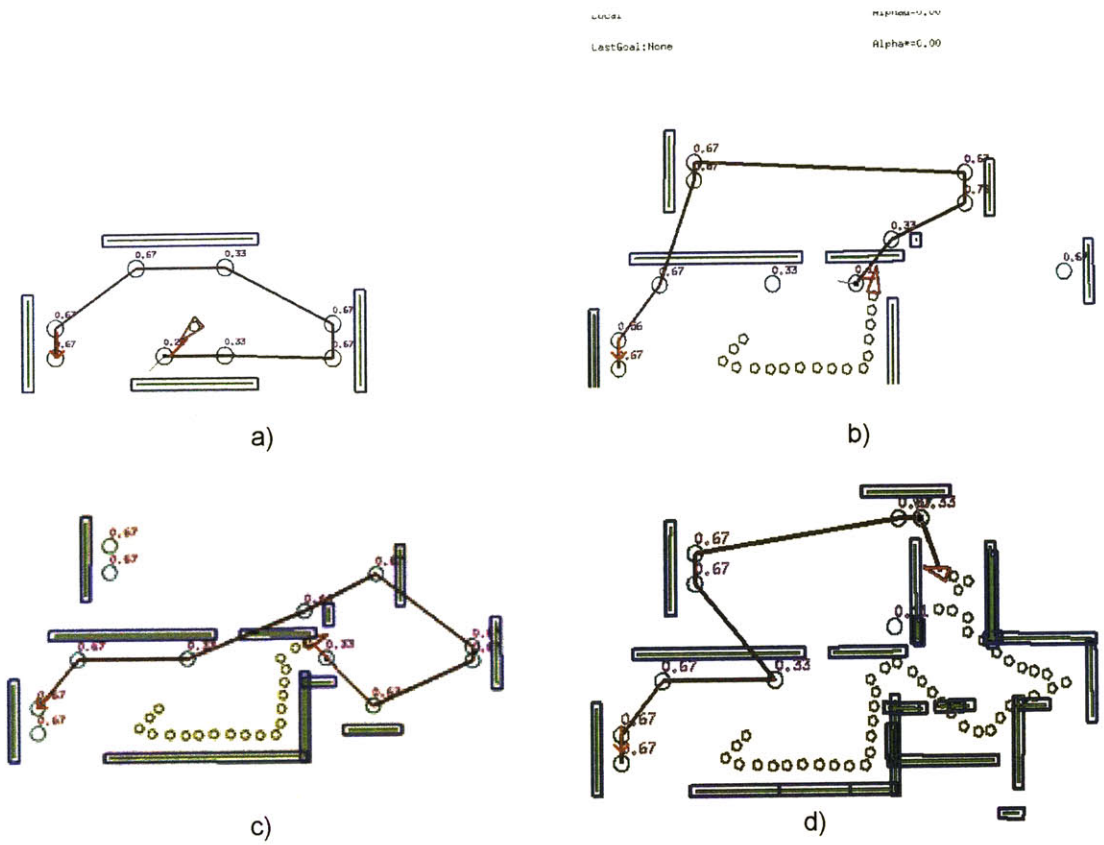


Figure 6.10 First Third of Building10Floor1 30m Receding Horizon Trial

The screenshots in Figure 6.10 show the first third of the Building10Floor1 trial using the receding horizon method with a horizon of 30m. Figure 6.10a shows the plan before the robot had driven anywhere. Note that from its initial position, the robot cannot map the large scale structure of its environment as we would like. In Figure 6.10b, the robot has executed the first segment of its plan, and on the way it has seen into the room above it and the end of the hallway to the right. As a result, the path has changed to visit this new room. The robot still has not seen most of the environment, however, and so this path has not been planned to be globally efficient. After moving just a little bit farther, the robot picks out a line at the bottom right of the map; therefore, the path changes again. As Figure 6.10c shows, this new path heads down the hallway to the right, visits one point above the hallway, and then goes back to the robot's initial room in order to finish exploring there. Figure 6.10d shows that the robot was able to execute the first part of this path. Because the robot is using a receding horizon, the robot can now

reach two more points with its path than it could before. However, these additional points do not change the path's overall direction.

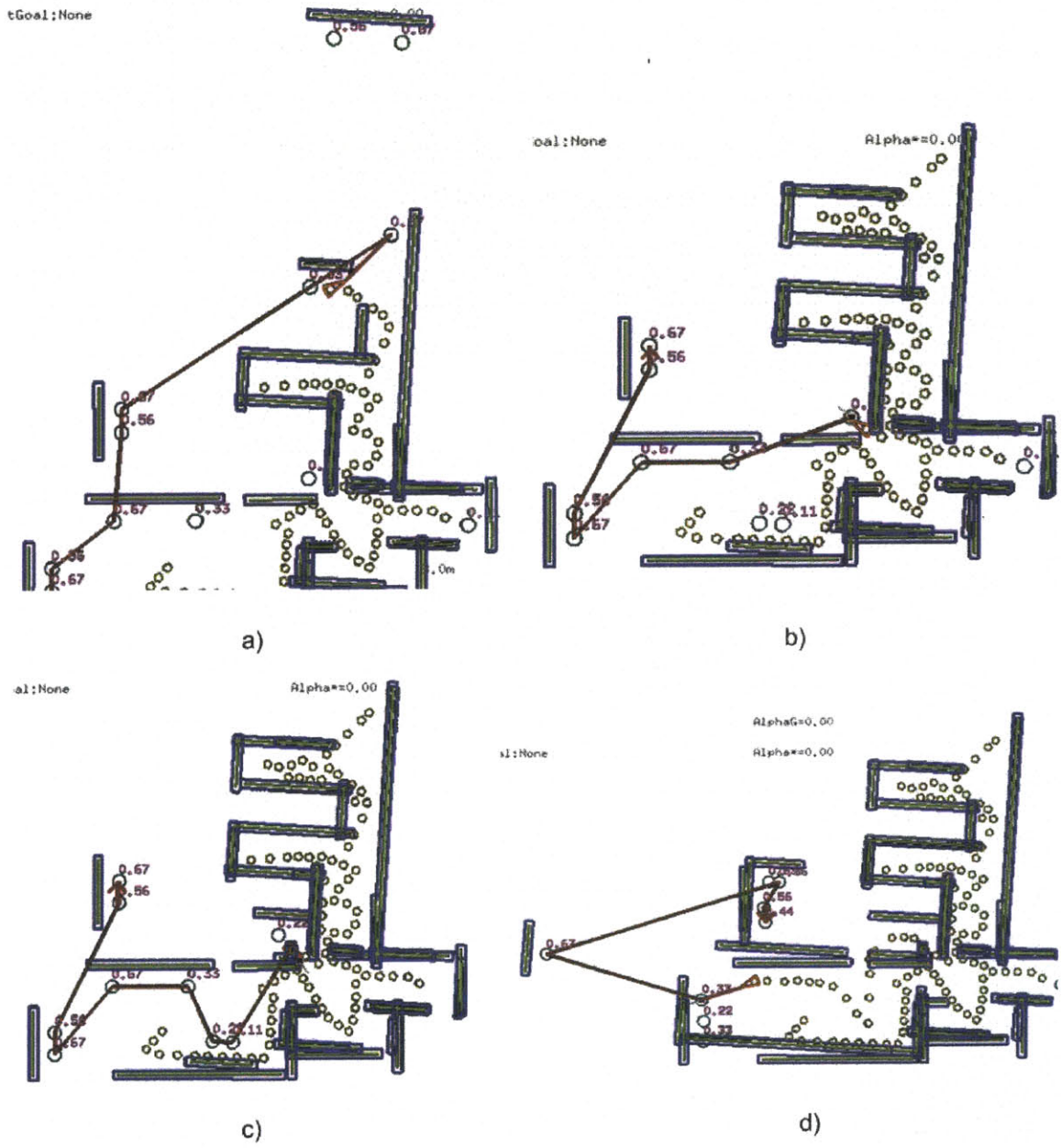


Figure 6.11 Second Third of Building10Floor1 30m Receding Horizon Trial

A very long wall blocks the path in Figure 6.10d, and in Figure 6.11a the robot has journeyed upwards to find a way around it[17]. In Figure 6.11b, the robot decides it is not worth it to continue to go upwards to try to get around the wall, and instead goes around the bottom of the wall. The path has changed slightly in Figure 6.11c to visit two additional candidates, but the overall direction of the path stays the same. The robot finally gets to execute most of this last portion of its plan. Yet in Figure 6.11d the robot has just seen the end of the hallway on the left side of the map; therefore, the robot delays visiting the last room in the path to go visit this hallway.

[17] The screenshots in Figure 6.11 and 6.12 are actually from a different trial than the screenshots in Figure 6.10. Nevertheless, this second trial still used the receding horizon method with a horizon of 30m, and the first part of the trial was almost exactly identical to the steps depicted in Figure 6.10.

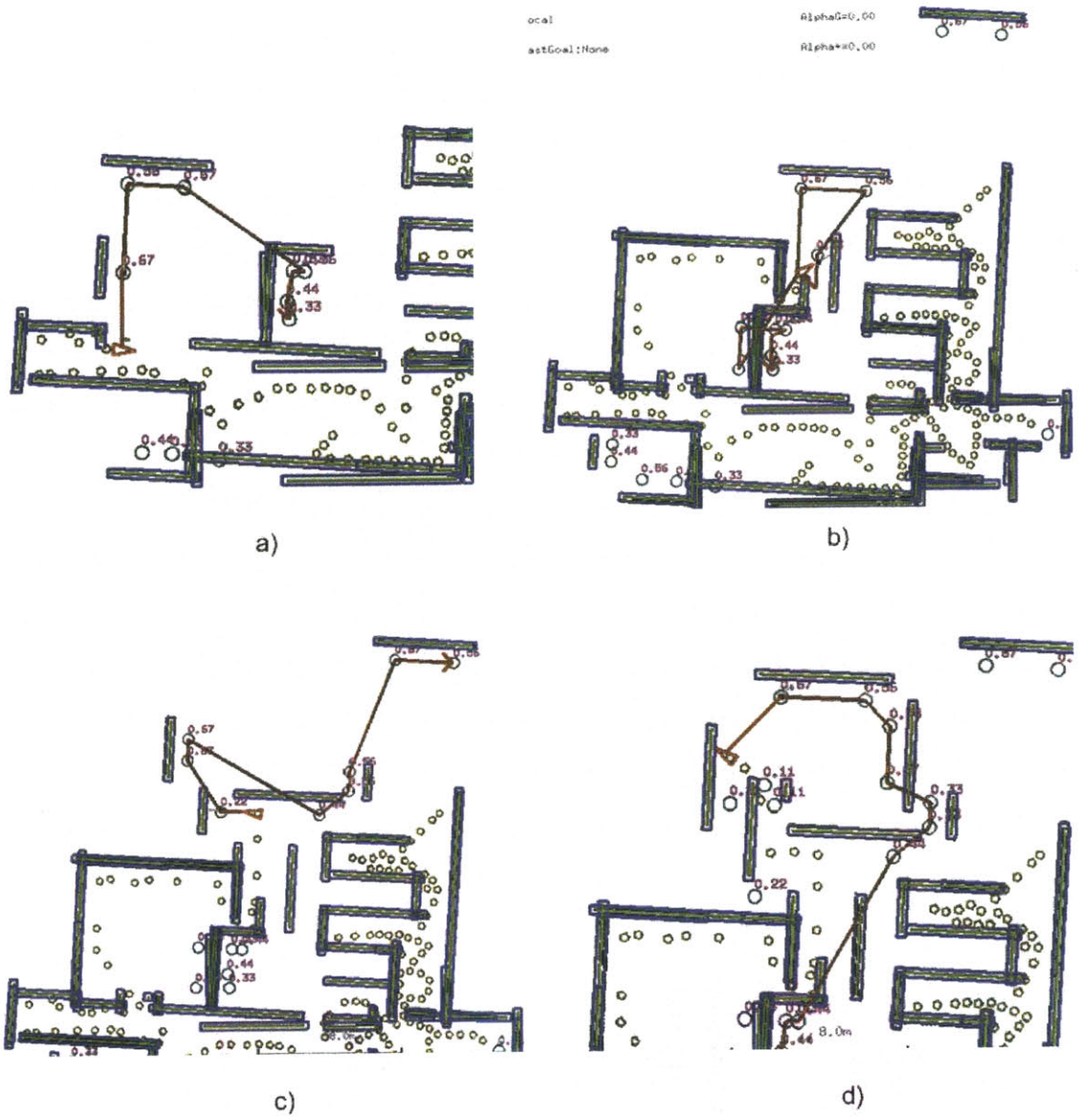


Figure 6.12 Final Third of Building10Floor1 30m Receding Horizon Trial

New rooms and hallways continue to distract the robot from exploring the final room in its path, as Figure 6.12a and Figure 6.12b show. In Figure 6.12c, the robot has found a large unexplored area and gives up on visiting this final room. In Figure 6.12d, however, the robot has explored the new area a bit more and once again plans to visit the old final room. The robot never gets to execute this plan to completion, however, because the mission ends. Figure 6.13b shows the final map.

In this trial, the robot did not even find out about the upper part of the environment until the very end. Therefore, the robot never truly planned globally

efficient paths using knowledge of the overall structure of the environment. It was mostly out of luck that the robot's path in this trial was more efficient than its path in the greedy trial in Building10Floor1 (the greedy trial had the second most efficient path in this environment). Figure 6.13a shows the robot's map after exploring Building10Floor1 using the greedy method. Figure 6.13b shows the final map of the 30m receding horizon trial with the most significant line features that were not mapped during the greedy trial circled. In the greedy trial, the robot never got to map the short hallways on the lower left and lower right of the environment, because it immediately headed up to the top of the map and spent most of its time exploring this part of the map in detail. In the receding horizon of 30m trial, however, the robot could not have actually known that it was better to map these hallways before heading up to the top of the environment, for the robot did not even know there was anything at the top of the environment until the end of its mission. The robot therefore only mapped these hallways out of luck in the 30m receding horizon trial. In addition, the longest line that the robot did not map in the greedy trial was mapped largely by accident when the robot was trying to find some way out of the enclosed corridor that this line forms the right wall of. In other words, much of the 30m receding horizon trial's success was due to luck.

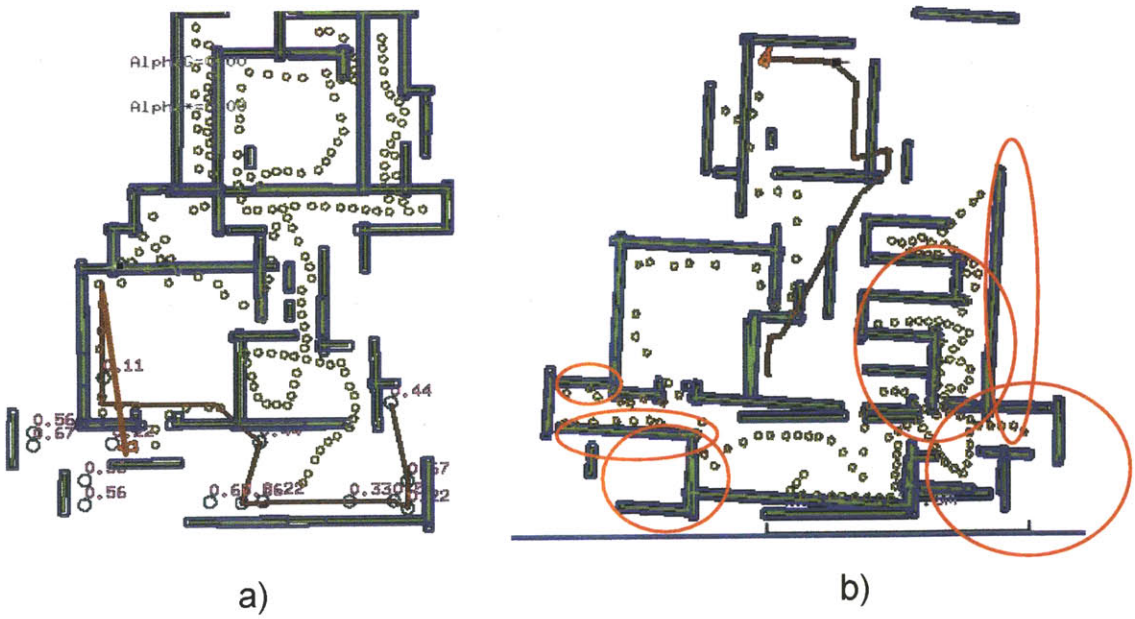
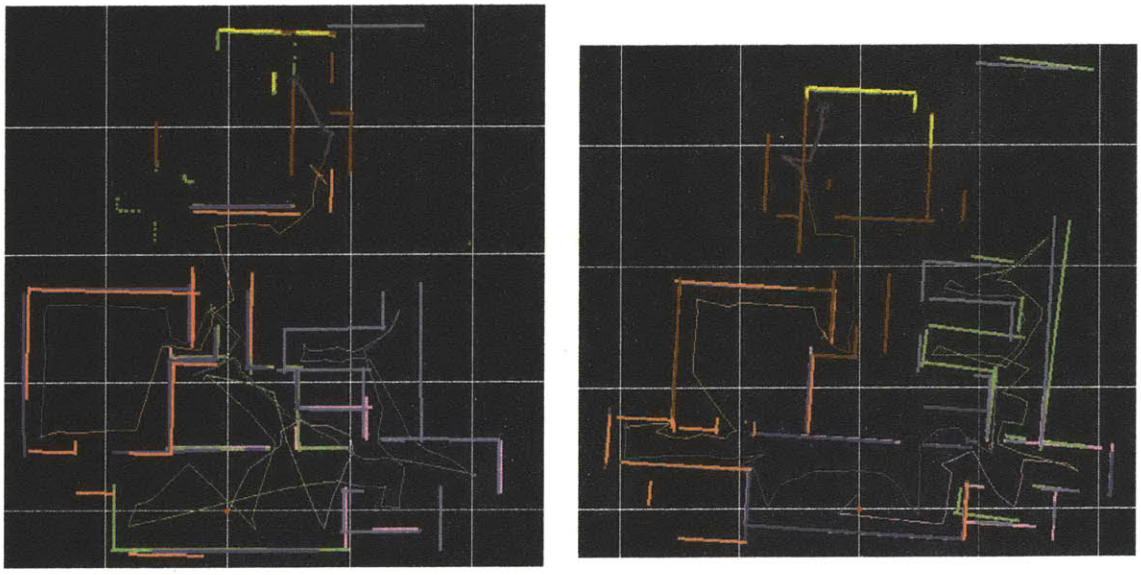


Figure 6.13 Final Maps for the Building10Floor1 Greedy and 30m Receding Horizon Trials

Yet the robot was able to plan paths over an intermediate size distance and execute these plans without them changing entirely. For example, the last segment of the robot's plan stayed constant throughout almost the entire trial. And in the hallway to the right, the big room on the left, and the area at the top of the map, the robot was able to execute most of its plan. This successful intermediate length planning caused the robot to execute a more efficient path than in the other finite horizon trials. For example, in the portion of the 15m fixed horizon trial shown in Figure 6.6, the robot inefficiently bounces back and forth between its initial room and the rooms directly above, because the robot's horizon is too short. In addition, Figure 6.14a shows the final map for the 15m receding horizon trial. The thin multicolored line winding through the map shows the path the robot executed. This path also bounced around between the initial room and the surrounding rooms. Figure 6.14b once again shows the final map for the 30m receding horizon trial. The executed path in this final map does not show these signs of inefficient bouncing, which reaffirms that planning over the longer 30m horizon improved the efficiency of the robot's executed path.



a) b)

Figure 6.14 Final Maps for the Building10Floor1 15m and 30m Receding Horizon Trials

In the end, therefore, the Building10Floor1 environment was not open enough for the robot to be able to plan globally efficient paths. Nevertheless, the environment *was* open enough for the robot to be able to plan over an intermediate length horizon and not have these plans change too drastically. This intermediate length planning improved the efficiency of the robot's executed path. We conclude that as long as the environment is open enough for the robot to discern the overall structure of the environment to some radius, the robot can execute finite horizon paths planned over a horizon scaled to this radius well enough to improve the efficiency of exploration. Furthermore, the longer this horizon is, the more efficient the robot's path will be.

Out of all of our experiments, the trials in Building10Floor1 that we just presented and the StructuredRocks trials provide the only real positive evidence for this conclusion. The trials in the other environments did not contradict the conclusion, however. In addition, there are a number of questions about observation planning that we have not examined. For example, we would like to find out from these experiments how the fixed horizon and receding horizon approaches to continuous observation planning compare to each other. Therefore, in the following sections we examine what interesting things the trials in each environment have to teach us. We also argue that, despite how it sometimes appears, the trials in other environment do not contradict the conclusions drawn in this section.

6.4.2 Real Buildings 34 and 36 Trials

We performed these trials on a real iRobot B21 robot on the fourth floor of two connected buildings. The part of the environment the robot explored in these trials consisted of a large elevator lobby and a hallway that dead ended in closed doors at one end and led to the elevator lobby at the other end. The main point of these experiments was to validate that the results we got from the experiments that we performed in simulation were similar to the results we would have gotten in a real environment. Overall, while there were some differences in these real world experiments, none of these differences made the results drastically different from the results we would have gotten in simulation.

One difference between the real world experiments and what we would have seen in simulation is that in the real world, the robot mapped many small details that we would not have included in a simulated environment. For example, three of the walls in the real environment were filled with windows. These windows were set back very deep into the wall (probably about a foot deep). The robot, therefore, mapped a few window frame edges as short vertical lines sticking out of the wall. There were also a few chairs and door frames that added small details to the robot's map that we would not have modeled in a simulated environment from the floor plans of the building. These small details added a few more candidates for the robot to visit; however, there were not enough of them to significantly alter the robot's exploration path.

Another difference between the real environment and simulated environments was that the robot extracted many redundant lines in these trials. Figure 6.15 shows the final map from each trial in this environment. In the greedy trial and 30 meter receding horizon trial, we see that the robot has mapped many close together lines for each wall. Most likely, these redundant lines were caused by the glass windows and the mirror-like metal of the elevator doors in the environment. The robot's laser scanner does not receive good data from surfaces that are somewhat transparent or that are mirror-like. Therefore, the robot probably had trouble localizing and thought it was seeing new walls when it was actually seeing walls it had mapped already. In our simulations, we would not have experienced this problem of the robot mapping redundant lines, because we would have made all walls act like normal opaque walls.

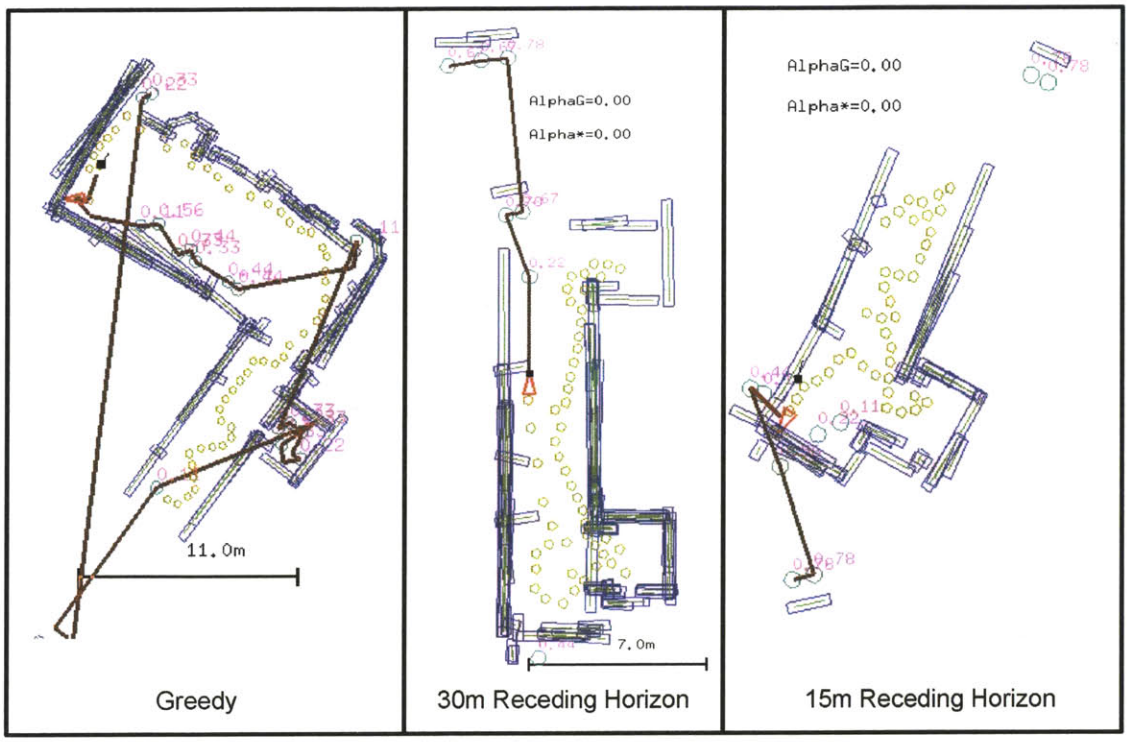


Figure 6.15 Final Maps for Real Buildings 34 and 36 Trials

At times these redundant lines caused there to be more candidates in the map than there should have been. However, none of these trials lasted long enough for the number of extra candidates to get to a size that drastically altered the relative performance of the observation planning method. Although in longer missions these types of environments may be problematic, the problem of extracting redundant lines should be addressed by improving the SLAM algorithms and laser scanners, not the exploration algorithm. A map with many extra lines is not desirable, even if we can find a way to explore well using such a map.

Table 6.6 shows that the all of the robot performed similarly in terms of average fraction of line extracted for all observation planning methods we tested. This environment is very simple in that there are not a lot of blocked off, hard to see features. Therefore, it was very easy for the robot to explore the environment thoroughly. What jumps out most from Table 6.6 is that in the greedy trial, the robot extracted a much larger total length of line than in the finite horizon trials. Looking at Figure 6.15, we see that the greedy trial was the only trial in which the robot mapped the big elevator lobby.

Mapping this lobby is what allowed the robot to extract a much larger total length of line than in the other methods.

The robot made it to the elevator lobby by chance in the greedy trial. In all trials, the exploration began with the robot in the middle of the hallway leading to the elevator lobby. In the two finite horizon trials, the robot never made it into the elevator lobby, because it went down and explored the other end of the hallway first. It was a matter of chance that the robot made it up into the elevator lobby in the greedy trial. Unlike in the finite horizon trials, for some reason the robot did not see much of the end of the hallway away from the elevator lobby in the greedy trial until the robot was close to the lobby. Figure 6.16 shows the beginning of the exploration in each trial. Figure 6.16a shows the robot's path early on in the greedy trial. The end of the hallway at the top of the picture is where the elevator lobby is. Note that the robot has not mapped any of the hallway below the robot's position. Furthermore, the robot will never turn towards this end of the hallway on its current path. Figure 6.16b shows the robot's path early on in the 15 meter receding horizon finite horizon trial, and Figure 6.16c shows the robot's path early on in the 30 meter receding horizon trial. In both of these figures, the robot has mapped some of the hallway below its current position, and its path takes it down to see more of that end of the hallway.

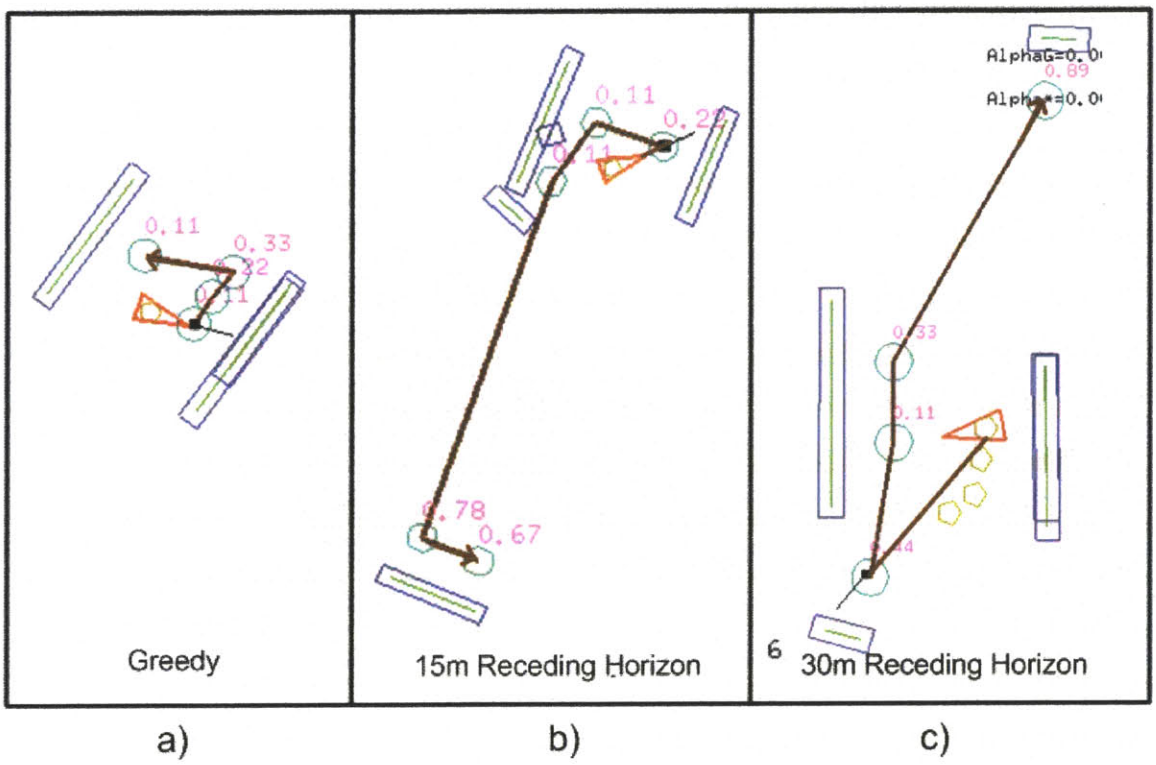


Figure 6.16 Why the Greedy Trial Got to the Elevator Lobby

There were a number of times before the robot made it to the elevator lobby in which if the set of candidates had been slightly different, the robot would have been pulled away from the elevator lobby in the greedy trial. It is important to note, however, that globally it is more efficient for the robot to explore the end of the hallway away from the elevator lobby before exploring the elevator lobby itself. In addition, the robot plans to do exactly that in the 30m receding horizon trial. In Figure 6.17a, the three candidates at the very top of the map are in the elevator lobby. The robot has planned to travel down to the bottom of the map before traveling up to the slightly further away elevator lobby. In Figure 6.17b, we see that the robot has gotten pulled down even farther towards the bottom of the map, but eventually plans on going up to explore the elevator lobby. Unfortunately, the mission ends before the robot ever makes it up to the elevator lobby. However, the path the robot would have executed in this trial probably would have been more efficient than the path the robot would have executed in the greedy trial, had we allowed the robot to explore until the entire environment had been mapped.

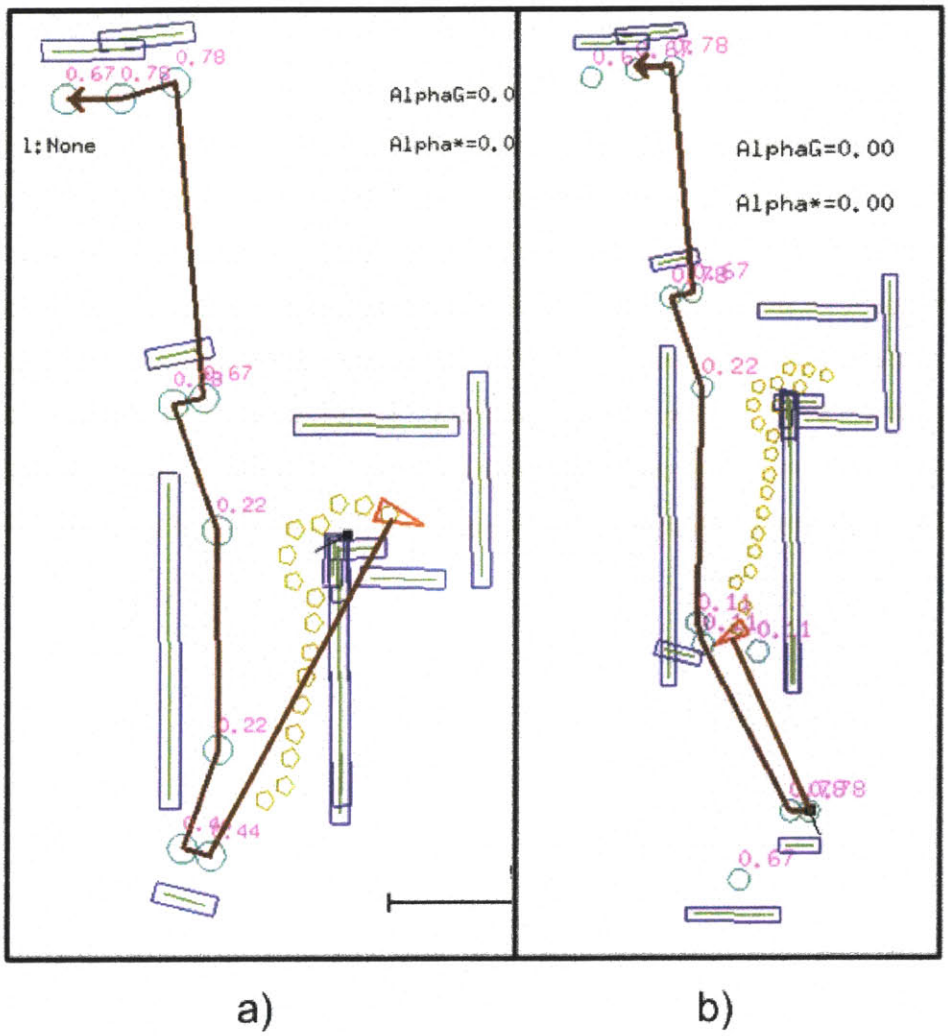


Figure 6.17 Middle of the 30m Receding Horizon Trial

6.4.3 15by15Room Trials

The 15by15Room is a fictional environment. We designed the 15by15Room to be a very small and simple environment that tested the theory that the finite horizon approach would perform well in buildings with big rooms and wide doorways. Looking at the results in Table 6.3, however, we see that all of the observation planning methods that we tested performed almost identically.

The reason none of the finite horizon trials significantly outperformed the greedy trial in any metric of exploration was that the environment was too small and simple. We expect that finite horizon methods will do well in buildings with big rooms and wide

doorways because the robot should be able to see the overall structure of the environment early on. However, in the 15by15Room, once the robot knew where all the large obstacles and interesting places to map were, the robot had basically mapped the whole environment. Therefore, the finite horizon methods never got to capitalize much on knowing the overall structure of the environment.

Figure 6.18 shows the final map of each trial listed in Table 6.3. In particular, Figures 6.18a, 6.18b, 6.18c, and 6.18d show the greedy, 15m receding horizon, 15m fixed horizon, and 8m fixed horizon trials respectively. For all observation planning methods, the robot was able to explore about two rooms over the 33 meter mission. Within these two rooms, all methods saw about the same amount of the walls in the environment.

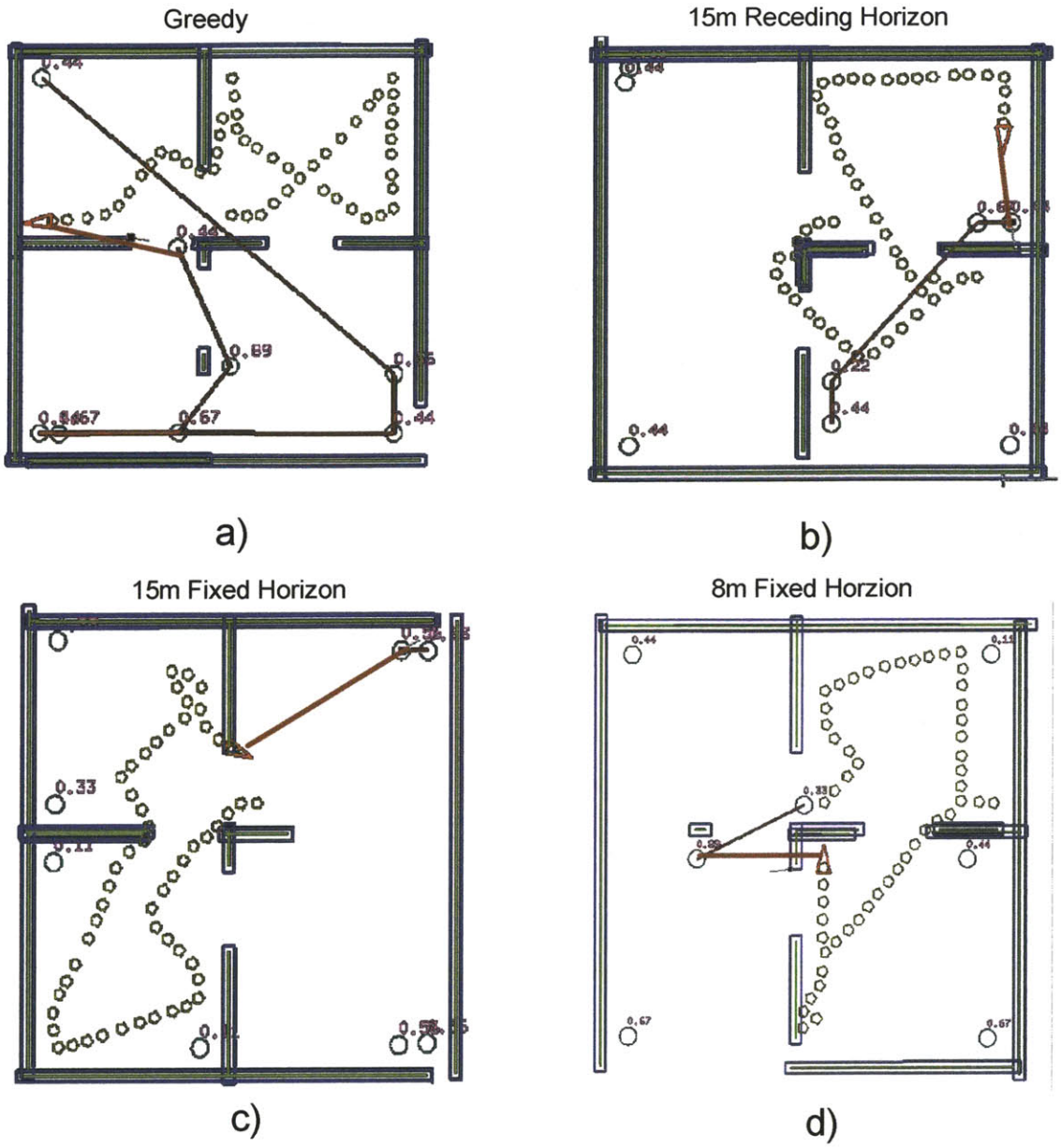


Figure 6.18 Final Maps of Trials in 15by15Room

In the 15by15Room environment, we also performed four trials in which the robot explored until there were no candidates left. This environment and the RandomRocks environment were the only two that were small enough for exploring until there were no more candidates to be feasible. In all other environments, the set of candidates would eventually become so large that the candidate identification and scoring code and the observation planning code would run too slowly to be usable.

Table 6.7 shows the distance the robot had to travel in order to visit all of the candidates in each trial. The robot traveled significantly farther in the two finite horizon trials than it did in the greedy or full horizon trials. The reason the robot's exploration was so much less efficient in the finite horizon trials was that the finite horizon paths often skipped visiting the low utility candidates in its current room to go to high utility candidates in other rooms. Unfortunately, the robot would eventually have to come back to the candidates that it had skipped, in order to visit all of the candidates in the map. In addition, the robot was susceptible to planning paths through walls that it had not mapped yet when it was using a finite horizon method, because the robot kept switching rooms. Often these paths caused the robot to make a very inefficient maneuver to get around these unmapped walls when the robot finally saw them.

Observation Planning Method	Horizon	Distance Traveled
Greedy		79.5689m
Receding Horizon	15m	107.428m
Fixed Horizon	8m	108.099m
Full Horizon		86.4245m

Table 6.7 Results of Visiting Every Candidate in 15by15Room

The greedy and full horizon methods, on the other hand, do not take the utility of the candidates into account. Therefore, these methods tended to have the robot explore the room it was in thoroughly, before moving to the next room. The greedy and full horizon methods, therefore, performed much more efficiently by the measure of visiting all of the candidates. As we see from these trials, however, analyzing how far the robot must travel to visit every candidate is not so much an overall measure of how efficiently the robot explored, as it is a measure of how efficient the robot was at exploring thoroughly. It is still a useful measure, however, because the other measure of thoroughness, the average fraction of line feature extracted, did not show any difference between these observation planning methods.

6.4.4 25by45Room Trials

The 25by45Room is another fictional room that we designed in order to test the theory that buildings with big rooms and wide doorways will allow finite horizon methods to perform more efficiently than other methods. The 25by45Room environment is bigger and more complex than the 15by15Room. This increase in size is promising, since the 15by15Room is too small for the finite horizon approach to have a chance to perform well. In general, we found that the robot is able to discern much of the structure of the environment early on and to execute many of its paths to completion in the 25by45Room environment. Therefore, we expect that finite horizon methods should perform very efficiently in this environment.

Looking at Table 6.4 we see that the two trials that used the fixed horizon method with a horizon of 15 meters performed the best in terms of total length of line feature mapped. In terms of average fraction of line extracted, one of the 15m fixed horizon trials performed the best out of all trials and the other performed the worst. It is unusual that trials using the fixed horizon method performed the best in terms of length of line mapped, for our experiments in other environments show that using a fixed horizon usually causes the robot to explore thoroughly.

Figure 6.19 shows most of the second trial that used the fixed horizon method with a 15 meter horizon. The robot was able to execute all of the paths it plans with little or no change in this trial. In Figure 6.19a, the robot has traveled down the column of rooms to the left of the room the robot started out in. The path in Figure 6.19a ignores the lower utility points in the lower left corner of the robot's room and takes the robot to slightly higher utility points in the less explored right side of the map. In Figure 6.19b we see that the robot was able to execute this path to completion. Figure 6.3c shows the 15 meter path for the new horizon, and Figure 6.3d shows that the path changes after the robot executes the first part of it. In Figure 6.3e the robot has planned another 15 meter path for a new horizon that heads farther to the right into unexplored territory, since this is where the candidates with the highest utility are. Finally, Figure 6.3f shows that the robot was able to execute this path.

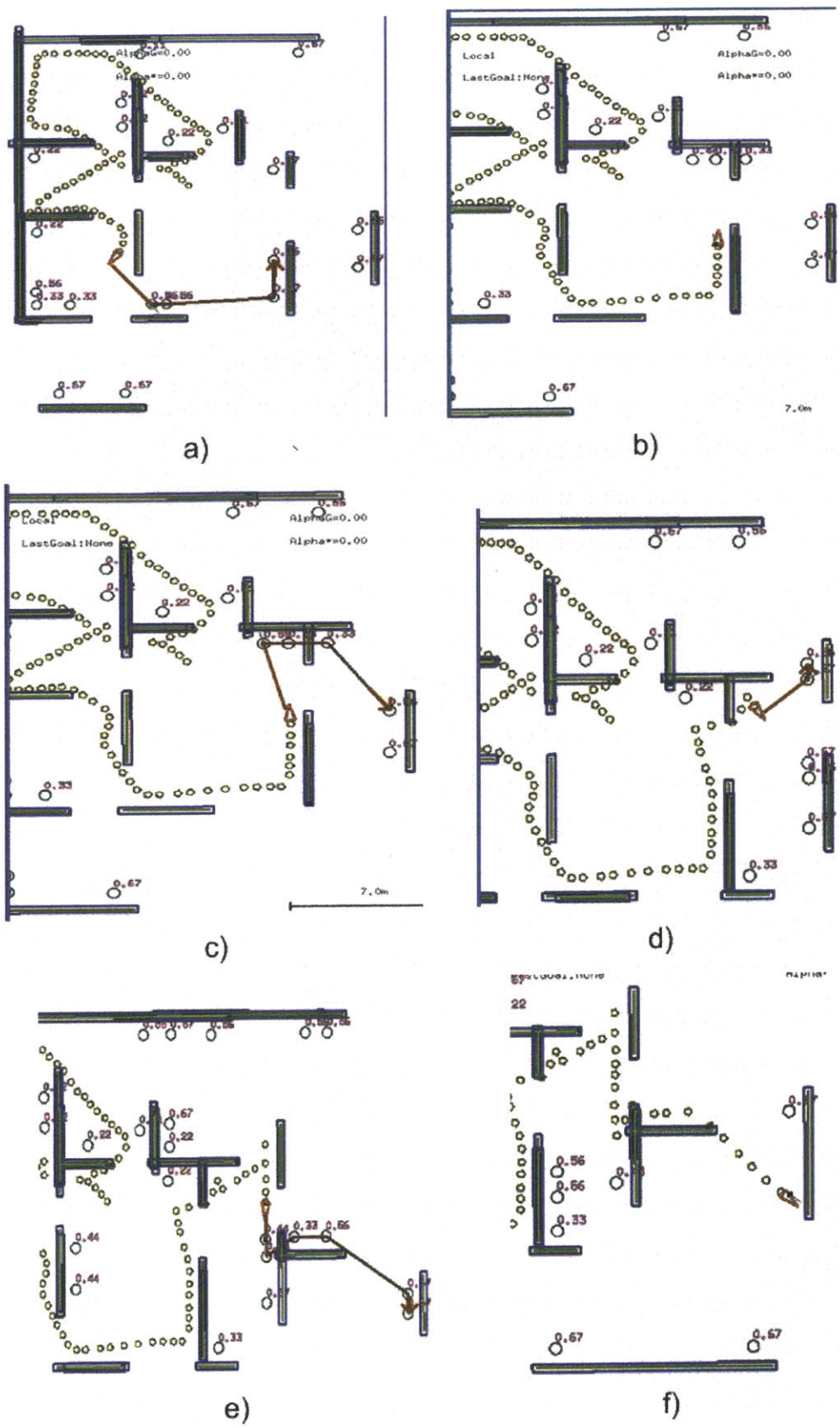


Figure 6.19 Part of the 15m Fixed Horizon Trial

As we see from these figures, the robot's 15 meter paths were only planned to be locally efficient. A 15 meter horizon is not long enough for the robot to plan a path that is globally efficient in this environment. It was, therefore, partially due to luck that the overall path that the robot executed explored new areas of the map efficiently. However, the fact that unexplored regions of the environment contained high utility candidates helped somewhat to guide the robot's paths towards the most unexplored regions in the map. Nevertheless, the 25by45Room environment is open enough where it seems like the robot would benefit from planning paths over a longer horizon.

We tried using the receding horizon method with a horizon of 40 meters, yet midway through exploration the S-TSP solver began taking a very long time to find a solution. We eventually had to halt the trial. The closest we got, therefore, to performing a trial with a horizon longer than 15 meters was to perform a trial using the full horizon method. Figure 6.20 shows a few snapshots from this full horizon trial. The robot was able to execute its full horizon path without the path ever drastically changing direction. However, small changes in the path did cause the robot to explore somewhat inefficiently. Figure 6.20a shows the full horizon path early on in the mission. In Figure 6.20b we see that the robot has executed almost half of the path to completion with the path only changing slightly. In Figure 6.20c, however, mapping errors have caused two redundant lines to appear to the left and above the robot. These redundant lines have created candidates back in regions of the map that the robot has already explored. The full horizon path has, therefore, changed slightly to revisit this area, which hurts the efficiency of the path that the robot executes. Indeed, looking at Table 6.4 we see that the robot performs the least efficient in terms of total length of line mapped in the full horizon trial. Another factor that contributes to the robot not mapping many of the line features in the full horizon trial is that the full horizon method does not take into account candidate utilities when planning its path. Therefore, the full horizon method does not prioritize heading into unexplored regions like the finite horizon method does.

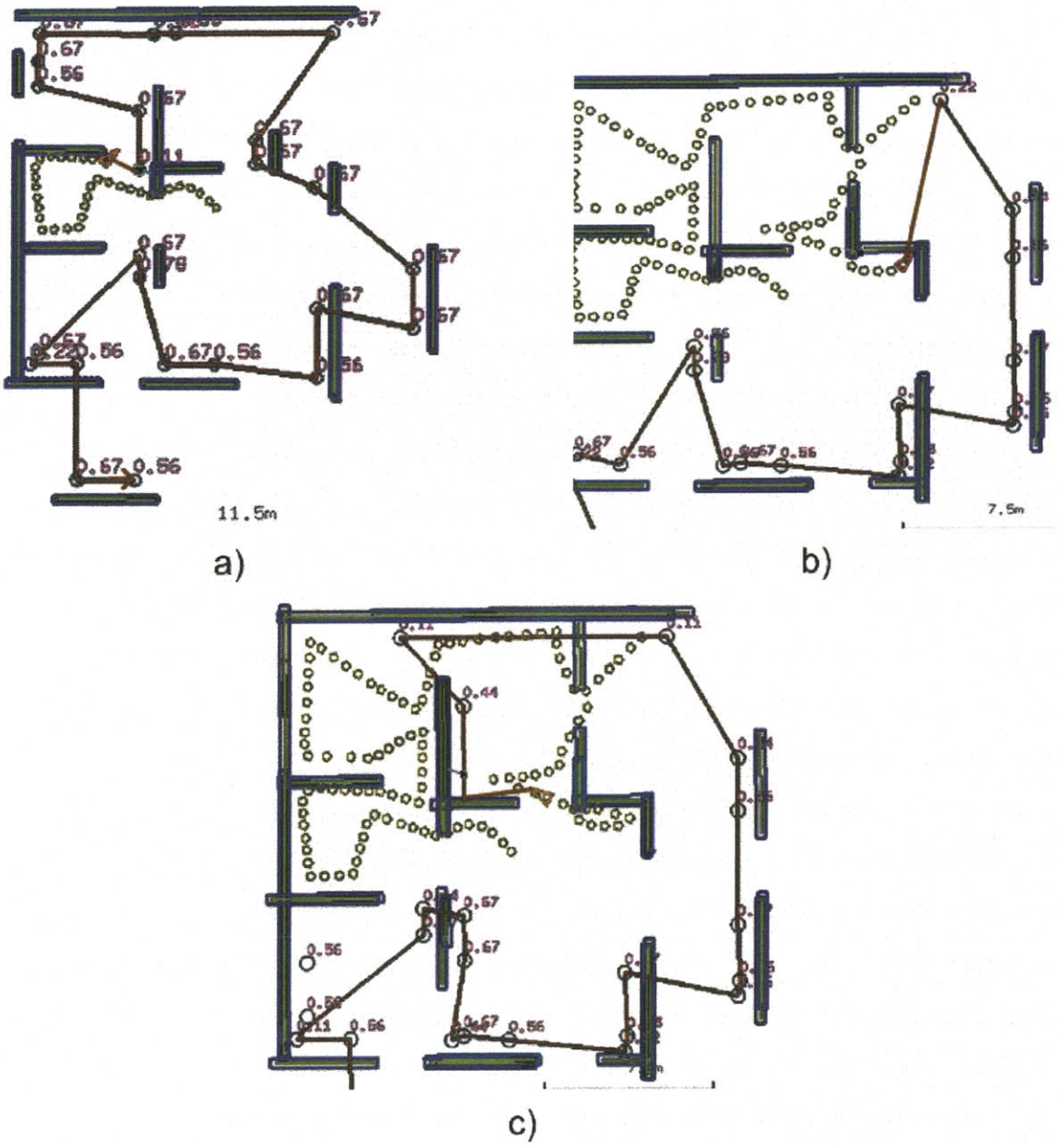


Figure 6.20 Moments in the 25by45Room Full Horizon Trial

A final point to note about the 25by45Room trials is that, while the 15 meter fixed horizon trials performed the best out of all trials, the 15 meter receding horizon trial performed second worst in both total length of line feature mapped and average fraction of line extracted. Part of the reason that the receding horizon trial did worse than the fixed horizon trials is that the robot is less likely to finish executing its path when it uses the receding horizon method, as we noted in Chapter 3. In general, the robot executed its

path to completion more often using the fixed horizon method than the receding horizon method in all environments. However, a lot of the receding horizon's inefficiency in the 25by45Room environment was due to getting unlucky, since a 15 meter horizon was not long enough for the robot to compute globally optimal paths.

6.4.5 NE43Floor8 Trials

Looking at Table 6.1, we see that no trials stand out as being unusually efficient or inefficient in any metric of exploration. The main lesson we learn from the trials in this environment is that it is very unlikely for the robot to get to execute much of its planned path without the path changing drastically in such a closed environment. Furthermore, we also learn that when the robot's path changes this frequently, most observation planning methods end up performing about as well as or slightly worse than greedy methods. In other words, the robot did not get caught making large unrewarded sacrifices too often in this environment, even though the robot's planned path changed constantly.

Figure 6.3 shows an example of the robot's path changing frequently in this environment. The only trial in which the robot often got to execute its planned paths to completion with little alteration was the eight meter fixed horizon trial. This fact confirms our reasoning in Section 6.4.1 that when the set of candidates changes frequently, the robot will have a better chance of executing paths planned over a short horizon. Eight meters was often short enough to keep the robot's planned path within the robot's current room. The fact that the robot executed many of its eight meter fixed horizon trials to completion also confirms that the robot is more likely to complete paths planned over a fixed horizon than paths planned over a receding horizon. Nevertheless, the robot did not perform efficiently in any metric of exploration in this trial. The fact that the robot did not explore its environment particularly thoroughly confirms that in order for planning over a short horizon to allow the robot to explore its environment thoroughly, changes must be made to the Newman, Bosse, and Leonard candidate identification and scoring method.

6.4.6 Building10Floor1 Trials

We discuss the two most interesting trials in the Building10Floor1 environment, namely the 30 meter receding horizon trial and the 15 meter fixed horizon trial, in Section 6.4.1. We noted in Section 6.4.1 that it is only out of luck that the robot performed better using a 30m receding horizon than it did using the greedy method. One additional point is that the robot executed a very similar path when it used the full horizon method to the path that it did when it used the greedy method. These two executed paths were similar because both got drawn up the long corridor to the top of the environment early on and, therefore, did not have time to fully explore the lower part of the environment. Therefore, it was also only out of luck that the robot performed better using the receding horizon method with a 30 meter horizon than it did using the full horizon method.

6.5 RandomRocks Trials

The RandomRocks environment is a fictional outdoor environment. Looking at Table 6.5, we see that all of the observation planning methods we tested performed similarly in all metrics in this environment. That no method stood out in terms of length of line feature mapped in this environment was largely due to the fact that the environment has no large scale obstacles in it, and the objects are fairly regularly dispersed throughout the environment. Therefore, no matter what direction the robot headed in, it mapped about the same amount of new features in this environment. Another problem was that the Newman, Bosse, and Leonard method does not encourage the robot to explore around the corners of squares very well. Figure 6.21 shows part of the eight meter fixed horizon trial. Figure 6.21a shows the path the robot has planned to explore what is actually a square rock in the environment. In Figure 6.21b, the robot has visited most of the candidates around this rock. Nevertheless, the robot still has not seen the other two sides of the square. Therefore, executing this path did not add any new line features to the robot's map. This scenario of the robot not seeing all the sides of a rock was very common. In addition, this environment was not conducive to the robot extracting many new line features because the lines in the environment were all only one meter long. Therefore, the robot only mapped lines that were very close to it, because

lines that were far away appeared too small for the SLAM algorithm to be certain enough about them to extract them.

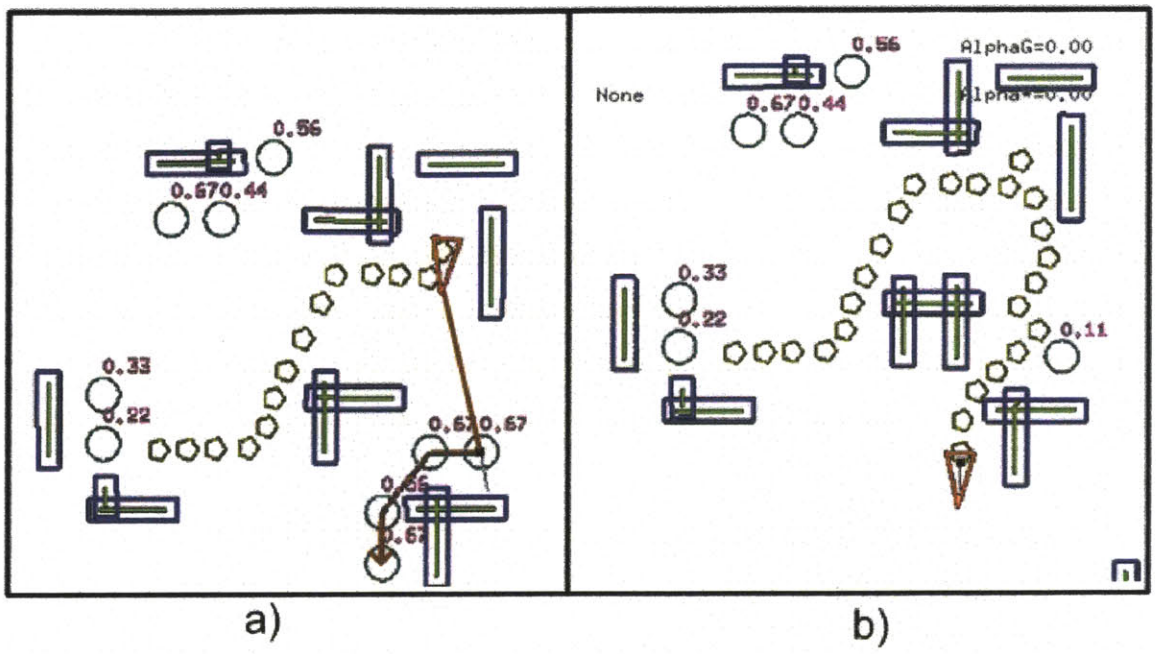


Figure 6.21 Difficulties Exploring around Squares in RandomRocks 8m Fixed Horizon Trial

The environment not being conducive to the robot extracting new line features also caused all of the observation planning methods to explore the environment thoroughly. At about the same point in all trials, the robot stopped mapping parts of new rocks to explore despite the fact that there were still unmapped rocks. From this point on, the robot simply explored the rocks that it knew about until it had mapped these rocks thoroughly and had nothing else to do.

6.6 Summary

The most important conclusion from these experiments is that in open environments, such as the Building10Floor1 and 25by45Building environments, the robot can sense the overall structure of the surroundings up to a certain radius. The robot can then plan an efficient path within this radius using the finite horizon approach and execute the path without much change. Trials in the Building10Floor1, StructuredRocks, and 25by45Building environments show that planning and executing these paths should allow the robot to explore its environment more efficiently than when it uses the greedy

method, on average. In particular, a robot using the finite horizon approach appears to have the potential to explore up to 25 percent more efficiently than a robot using the greedy approach in terms of total length of line feature extracted, the

We have also seen from these experiments that even when the environment is not open enough for the robot to discern the overall structure of the environment early on in the mission, the robot should still be able to plan and execute paths over a short horizon. Executing these paths causes the robot to visit most of the candidates in its local area before moving on to a new area. Therefore, if improvements are made to the way candidates are identified and scored in the Newman, Bosse, and Leonard exploration method, planning over a short horizon might enable the robot to explore its environment thoroughly.

From the trials in the NE43Floor8 and Building10Floor1 environments, we have drawn the conclusion that using the fixed horizon method allows the robot to execute more of its paths to completion than using the receding horizon method. The robot does not necessarily explore more efficiently when it uses the fixed horizon method than when it uses the receding horizon method, however.

Finally, the real world trials showed us that the results of the trials we ran in simulated environments are probably similar to the results we would have gotten if the trials had been in corresponding real environments.

7 Future Work

A lot of work remains to be done on the general problem of exploration. The ideal exploration system would take as input any combination of exploration goals and sub-goals, and would then control the robot to efficiently achieve these goals. For example, one might specify that the robot should try to get to a specific point A, but, along the way, the robot should explore to expand the coverage of its map and make sure the map uncertainty does not exceed a given upper bound.

Unfortunately, we do not have room in this chapter to survey directions for future work in the entire field of autonomous exploration. Therefore, we focus on future work for observation planning, in the context of exploring to increase map coverage. In particular, we discuss three important research topics that this thesis has suggested: testing approaches to observation planning with other candidate identification and scoring methods, improving the algorithm for solving the S-TSP as an OCSP, and predicting what the robot will see at candidates, in order to improve the effectiveness of planning multiple observations ahead.

7.1 Further Testing

In this thesis we only had time to test the greedy, finite horizon, and full horizon approaches to observation planning with the Newman, Bosse, and Leonard candidate identification and scoring method. Therefore, we do not know anything definite about how these approaches to observation planning will perform for the Gonzalez-Banos and Latombe or frontier-based candidate identification and scoring methods. As we discussed in Chapter 6, the candidates change unusually frequently in the Newman, Bosse and Leonard method, and these frequent changes makes it harder to plan efficient paths. With other candidate identification and scoring methods, therefore, we might expect that the finite horizon approach will be effective in more environments than just open ones. As a result, implementing and testing these three approaches to observation

planning with other candidate identification and scoring methods is a worthwhile endeavor.

There is also important work to be done on the testing procedure itself. Because we are interested in the *average case* efficiency of the various approaches to observation planning, it is important to be able to perform and evaluate many experiments in many different environments. Therefore, we would like to automate all aspects of running an exploration experiment, including generating a random simulated environment, driving the robot during exploration, and processing of the results. Although random indoor environment generators might be difficult to design, random outdoor generators should be straight-forward to implement, as long as the obstacles are small separate entities such as rocks or trees. In order to drive the robot automatically, all one needs is a controller that can execute the paths that the exploration path planner outputs. The most difficult task to automate is processing the results of experiments, because often these results require some interpretation. However, the running and processing hundreds of exploration trials would provide very valuable statistical verification of our initial impressions about the efficiency of the observation planning methods and would likely lead to new insights. Therefore, automating testing is an important problem to address.

7.2 Improved Methods for Solving the S-TSP as an OCSP

The constraint-based A* approach to solving the S-TSP as an OCSP that we presented in Chapter 4 is a very basic approach. One inefficiency that we can improve upon is that every time constraint-based A* checks the consistency of an assignment, the approach solves the TSP on the sub-graph corresponding to the assignment. Furthermore, constraint-based A* enumerates every full assignment in best-first order; thus, the approach has to solve the TSP on all possible sub-graphs in order of largest sub-graph to smallest, until it finds a consistent sub-graph. Yet solving the TSP on a graph becomes time consuming as the number of vertices in the graph increases. Thus, the constraint-based A* approach always starts out by solving the TSP on the most time-consuming sub-graphs. If the number of vertices in the input graph is very large, and if the horizon length is such that the solution path only visits a small number of vertices, then the approach will have to solve the TSP on a large number of very large sub-graphs.

In this section we discuss two methods of addressing this inefficiency. First, future approaches can avoid having to solve the TSP every time they check the constraint by finding bounds on the length of the solution to the TSP. If a lower bound on the length of a TSP path is longer than the horizon length, then the TSP path must be longer than the horizon length. Conversely, if an upper bound on the length of a TSP path is shorter than the horizon length, then the TSP path must be shorter than the horizon length. Second, future approaches can avoid explicitly considering every sub-graph with more vertices than the solution sub-graph by using conflict-directed A* [52] to perform the best first enumeration, instead of constraint-based A*. Conflict-directed A* is a sound and complete method of solving an OCSP that avoids enumerating assignments to the variables that it knows must be inconsistent.

7.2.1 Utilizing Bounds

Solving a TSP every time the approach needs to check a constraint is very inefficient. While it is feasible to check the TSP on graphs with 10 or 30 vertices, for graphs with a few hundred vertices, finding a solution to the TSP can take an unacceptably long amount of time [19]. Fortunately, since often we only care whether or not the length of the solution to the TSP is above or below L, we can improve the constraint-based A* approach to sometimes avoid solving the TSP by computing upper or lower bounds on the length of the optimal tour.

In particular, when checking the constraint on a sub-graph with a large number of vertices, the approach could first compute a lower bound on the length of the optimal tour. If the lower bound is greater than L, then the constraint must be violated. Only if the lower bound is less than L must the algorithm calculate the actual solution to the TSP. This lower bounding prevents the algorithm from wasting time finding the solution to large sub-graphs that obviously do not satisfy the constraint. Note that calculating an upper bound could not help the approach in this situation, for if the upper bound is shorter than L, then the assignment corresponding to the sub-graph must be the solution to the OCSP, and the approach will need to return the TSP path as the solution to the S-TSP. If the upper bound is longer than L, then we do not know whether the TSP path is longer or shorter than L, and the approach still needs to solve the TSP.

There are many methods for finding a lower bound on the length of the solution to the TSP [19]. The tighter the bound is and the faster it can be calculated, the better the bound computation. A simple lower bound on the length of a solution to the TSP on a graph is the cost of a minimum spanning tree for the graph. Using Prim's algorithm [40] it is possible to calculate a minimum spanning tree in time O(E + VlgV), where E is the number of edges in the graph and V is the number of vertices.

Upper bounds on the length of the solution of the TSP are useful for extracting conflicts, as we explain in the next section. Any approximation algorithm for the TSP can serve as an upper bound. The Concorde TSP solver comes with an implementation of the Chained Lin-Kernighan approximation algorithm [35]. If an upper bound is less than L, then the TSP path must have a length less than L. If an upper bound is greater than L, then we still do not know whether the TSP path has a length that is greater or less than L.

7.2.2 Utilizing Conflict-directed A*

Like constraint-based A*, conflict-directed A* is a method for enumerating the possible assignments to the decision variables of an OCSP in best-first order. The interesting feature of conflict-directed A* is that it takes advantage of information about the substructure of the constraints in the OCSP to avoid enumerating assignments during the best-first search that it knows cannot be consistent. Conflict-directed A* also uses the fact that the utility function is MPI in order to speed up search in a way that is similar to what constraint-based A* does. Given an admissible heuristic, conflict-directed A* is an optimal and complete search strategy for OCSP's.

Conflicts are central to conflict-directed A*. A conflict is any partial assignment to the decision variables of an OCSP that is inconsistent with the constraints. In an OCSP, any partial assignment that contains a conflict as a subset of its assignments must be inconsistent. Conflict-directed A* can be thought of as a method of returning the next best assignment that resolves all known conflicts. An assignment resolves a conflict if the conflict is not a subset of the assignment. Thus, conflict-directed A* bypasses all states that it knows must be inconsistent. Conflict-directed A* can incorporate new conflicts into its search strategy at any time.

We can use conflict-directed A* instead of constraint-based A* to perform the best first enumeration of assignments when solving the S-TSP. In terms of our OCSP formulation of the S-TSP, any sub-graph that has an optimal tour that is longer than L corresponds to an inconsistent partial assignment; therefore, any such sub-graph corresponds to a conflict. Furthermore, if the length of an optimal tour through a given sub-graph is greater than L, then the length of an optimal tour on any sub-graph that contains this initial sub-graph must also be greater than L. Therefore, our OCSP formulation of the S-TSP satisfies the requirement that any partial assignment that contains a conflict must be inconsistent. As a result, if we ever find any sub-graphs that have optimal tours longer than L, we can call this sub-graph a conflict and use conflict-directed A* to avoid explicitly enumerating any sub-graph containing the conflict sub-graph.

In order to use conflict-directed A* to solve the S-TSP, however, we need a method of finding conflicts. There are two main objectives for a method of finding conflicts. The first objective is that the method should find conflicts quickly. If it takes longer for a method to find a conflict than the amount of time it would have taken to check all of the assignments the conflict rules out, then it is not worth it to find conflicts.

The second objective is that the method should try to find conflicts that are as small as possible. In general, the fewer assignments there are in a conflict, the more full assignments the conflict will rule out. This rule is true because the smaller the partial assignment is, the larger the set of supersets (assignments containing the partial assignment) will be. In other words, the fewer the number of vertices there are in a sub-graph, the larger the set of sub-graphs that contain this sub-graph is. The only caveat is that some of the full assignments that a conflict rules out might already have been ruled out by other conflicts. Therefore, in addition to being small, we would also like the conflicts found to overlap as little as possible.

The usual approach to finding conflicts is as follows. We start out not knowing about any conflicts. Conflict-directed A* then produces the assignment to the variables that has the best utility, and the constraint checker checks this assignment for consistency. If the assignment is found to be inconsistent, then conflict-directed A* looks for partial assignments contained by this assignment that are inconsistent. Conflict

directed A* then uses these partial assignments as conflicts as it continues its search. Every time conflict-directed A* produces an assignment and the constraint checker finds the assignment to be inconsistent, conflicts are extracted from this assignment.

The advantage of this approach to finding conflicts is that the approach uses the assignments that are already known to be inconsistent and only searches through subsets of these assignments to find conflicts. The difficulty in implementing this approach is that we need to develop a quick method of finding small conflicts from an assignment that we know is inconsistent. In other words, we need to develop a quick method of taking a graph that has an optimal tour of length greater than L and finding small sub-graphs of this graph that also have optimal tours of length greater than L. Ideally, this method would take $O(n)$ time, where n is the number of vertices included by the assignment. Finding a conflict extraction method that meets these requirements is an interesting area for future research.

There is another possible approach to finding conflicts, however. Instead of waiting to use the assignments that conflict-directed A* determines are inconsistent in order to find conflicts, a method of finding conflicts could search for certain conflicts before running conflict-directed A*. In particular, the method would search for all conflicts that correspond to sub-graphs that have less than or equal to a given number of vertices. For example, if the maximum number of vertices were four, then the method would check all possible sub-graphs of the input graph of two, three and four vertices for consistency. The approach would then use any inconsistent sub-graphs it found as conflicts when we run conflict-directed A* on the input graph. The motivation for this approach is that small conflicts are much more useful than large conflicts. For example, if the OCSP has j variables and we have a conflict of k variables, then there are 2^{j-k} full assignments that contain of this conflict, since the size of the domain of the variables in our case is two. If we decrease the size of the conflict by one variable, then there are $2^{j-(k-1)}$ full assignments that contain this conflict; in other words, the number of full assignments ruled out by a conflict increases by a factor of two for every variable we remove from the conflict. Furthermore, small sub-graphs are very easy for a TSP solver to solve. Therefore, it is feasible to search through all small sub-graphs for inconsistent

partial assignments before beginning conflict-directed A*'s best-first enumeration of assignments.

A final point to note is that any method of finding conflicts can make use of both upper and lower bounds on the length of the solution to the TSP in order to speed up constraint checking. The reason we can use upper bounds for finding conflicts is that if we find that a partial assignment corresponds to a sub-graph whose upper bound is less than L when we are searching for conflicts, we do not then need to find the TSP tour through the sub-graph, as in constraint-based A*. Instead, we simply do not use the sub-graph as a conflict.

There is a good chance that approaches to solving the S-TSP that use conflict-directed A* would perform drastically better than approaches using constraint-based A*. Conflict-directed A* approaches to solving the S-TSP may even end up outperforming existing branch-and-cut algorithms for solving the S-TSP. Therefore, future work in conflict-directed A* approaches to the S-TSP holds a lot of promise.

7.3 Predicting the Outcomes of Observations

In Chapter 3 we noted a major concern about the finite and full horizon observation planning methods. Specifically, if a candidate identification and scoring algorithm is doing its job, then every time a robot visits a candidate, the robot should add previously unexplored area to its map, and the set of candidates should change. Furthermore, if the set of candidates changes every time the robot visits a candidate, then we expect that the robot will not be able to effectively plan ahead, and thus the greedy observation planning method will probably perform similar to or better than either the full or finite horizon method. We often saw the candidates changing this frequently in the experiments discussed in Chapter 6, and indeed, the greedy method was almost always one of the most efficient methods. In order to know for sure if the candidates change this frequently with other candidate identification and scoring methods, we need to perform more experiments.

If the set of candidates changes frequently, then the only way a robot can effectively plan for the future is to predict how the set of candidates will change. There are two ways the robot could make such predictions. Either the robot could start out with

a model of the candidate dynamics a priori, or, if the environment is somewhat regular, the robot could learn a model of the candidate dynamics from past experience. The exploration method of Burgard et al. [10] explained in Chapter 2 learns a very simple model of how the candidates will interact every time a robot visits a candidate. However, there are currently no other methods, that we know of, of predicting how the candidates will change as a robot explores. Therefore, we can only speculate about what good approaches to prediction might entail.

The first question to address is how we would model the way the set of candidates changes. We would like to know how the set of candidates will change for each possible action the robot could take. In order to cut down on the number of actions to keep track of, we could consider only actions that drive the robot directly to a candidate, and have one action for each candidate. We cannot know the result of an action for certain in our model, since otherwise there would be no reason to explore the environment. Therefore, our model should have multiple possible outcomes for each action, and it should assign probabilities to each outcome.

We might, therefore, decide to represent our model as a Markov Decision Process [5]. Recall that an MDP consists of a set of states $\{\mathbf{S}\}$, an initial state S_0, a transition model $T(s_1, a, s_2)$ that gives the probability that taking action a in state s_1 will put the agent in state s_2, and a reward function $R(s_1, a, s_2)$ that assigns a real number to the situation in which taking action a in state s_1 results in the agent being in state s_2. Because the transition function of MDP's depends on the current state and no previous states, the state must capture all of the information that is important in determining what the next state will be. It is tempting to make the current set of candidates the state, however, the set of candidates does not capture all of the information that is relevant in determining the next state. In general, it helps to know the entire current map, in addition to the current set of candidates, in order to predict how the set of candidates will change as the result of some action. After all, exploration methods use the current map to identify and score candidates. If we can predict how the map changes as the result of an action, then we can predict how the candidates will change. Therefore, we represent the state of any given moment during exploration by the entire current map and the current set of candidates.

Because we define the reward function to depend on the action that the robot takes, we can factor into the reward the distance that the robot travels in taking an action. In addition, we define the reward function to depend on both the current and next state. By comparing the next state to the current state, we can see how much information the action adds to the map. Therefore, the amount of information the robot adds to its map is also a factor in the reward. The reward function must then combine the distance the robot travels and the amount of information the robot adds to its map into one number. If we wanted to avoid combining these two values, we could redefine what we consider to be an action so that all actions have the robot travel the same distance.

Given a fully specified MDP model of how the set of candidates changes as the robot explores, we could, in theory, calculate an optimal infinite horizon discounted policy for the robot. The optimal policy is the policy that has the highest expected total discounted reward. If we were not completely confident in our model, however, we might not want to plan a policy all the way out to the end of the robot's mission. Instead, we could generalize the finite horizon approach to observation planning to plan a policy N actions into the future, where N is a constant. This approach to planning policies is also referred to as the finite horizon approach. A form of the finite horizon approach to finding policies has been used successfully in fault diagnosis [27]. Comparing the performance of infinite horizon discounted policies to the performance of finite horizon policies is an interesting area of future research.

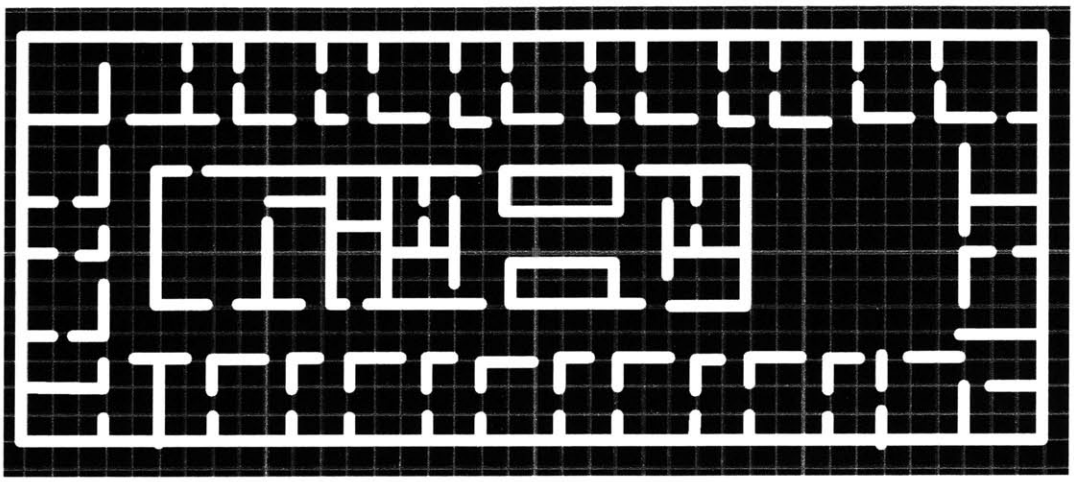
One major concern with our MDP model of the way the map changes as the robot explores is that the state space will have to be very large. There must be a state for every possible map the robot could ever build by executing a sequence of actions. In addition, this state representation is somewhat wasteful. The robot will almost never return to a previously visited state because the robot's current map does not usually regress to be identical to earlier maps. Therefore, another important area of future research is to find representations for these models that are more compact than the simple MDP representation that we have outlined.

The final question to address is how a robot could learn a probabilistic model of how the map changes as the robot explores. It is much more desirable for a robot to be able to learn a model of its environment then for us to supply it with such a model a

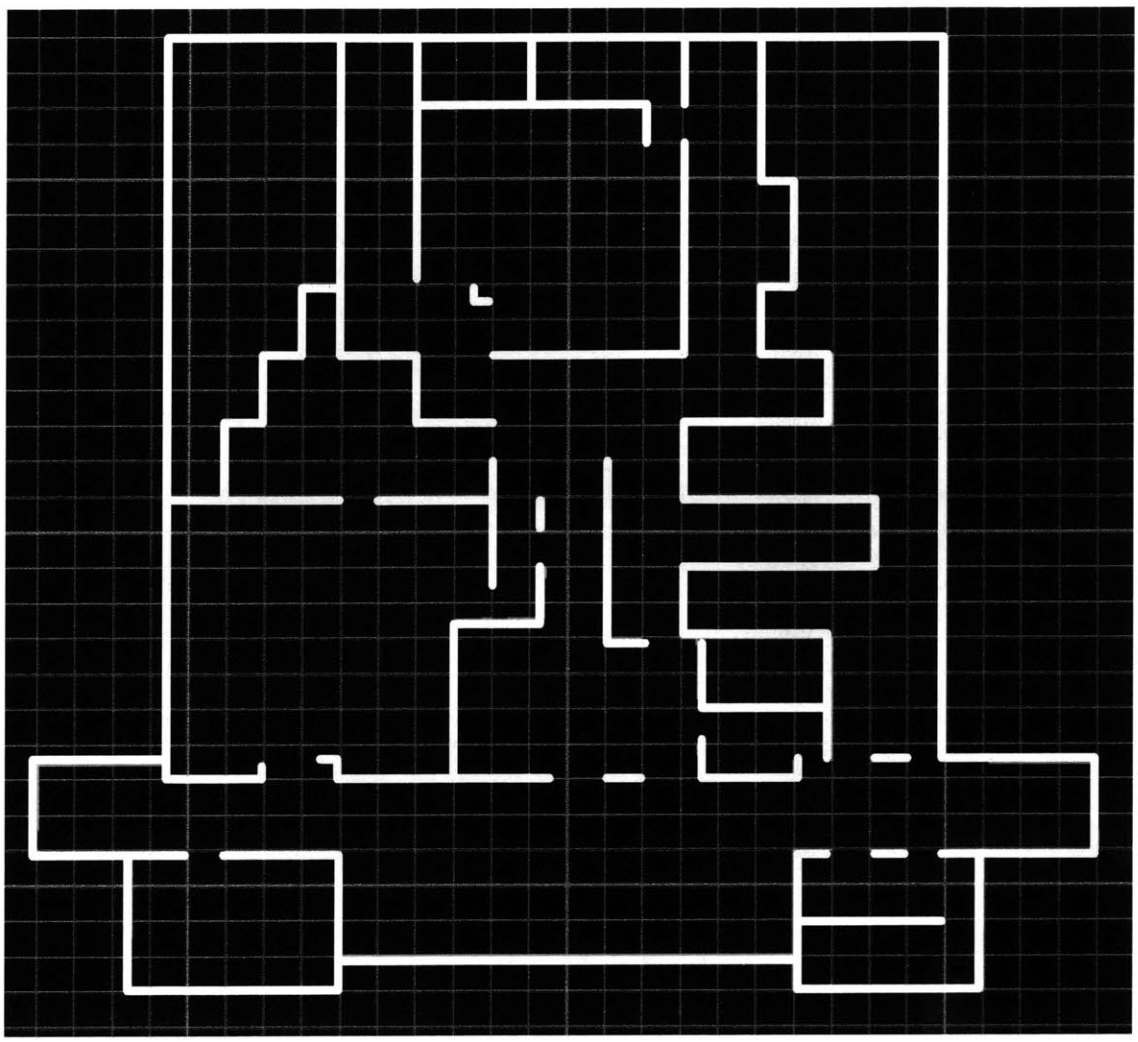
priori, because usually we have no good source of such models. The field of statistical learning [23] deals with learning probabilistic models from past experience. Therefore, a study of how well various statistical learning techniques work to learn regularities while mapping environments is another interesting area of future research.

Appendix A: Environments Used for Testing

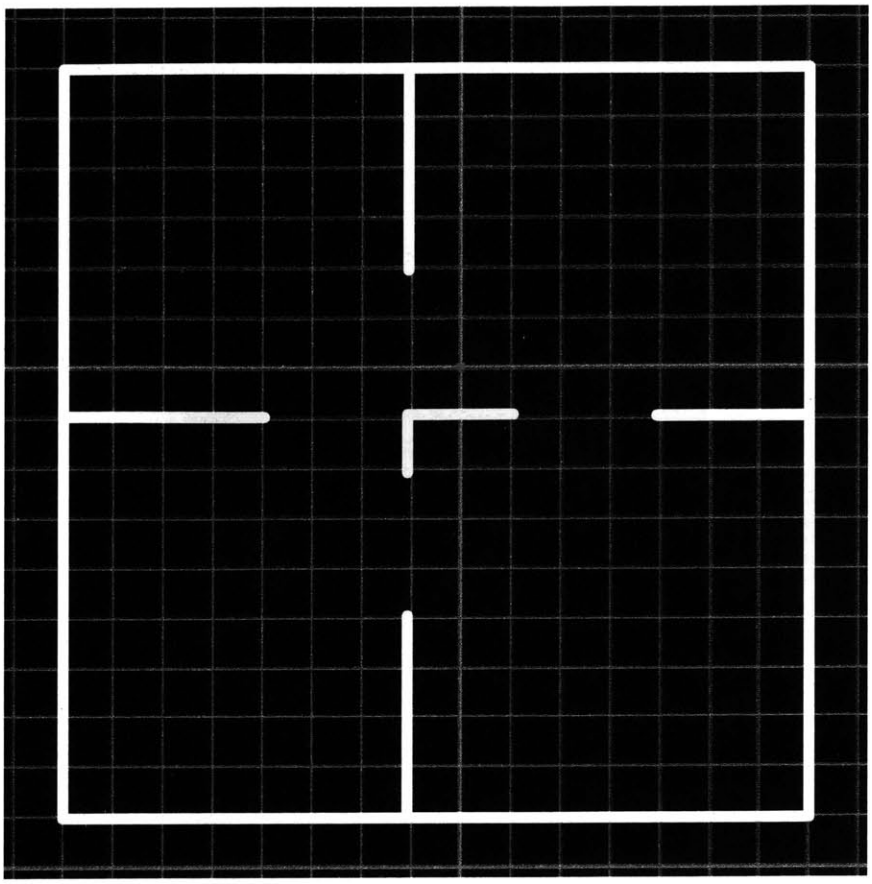
The figures below depict the floor plans of the simulated environments that we used. The grid cells in these figures are one meter by one meter.



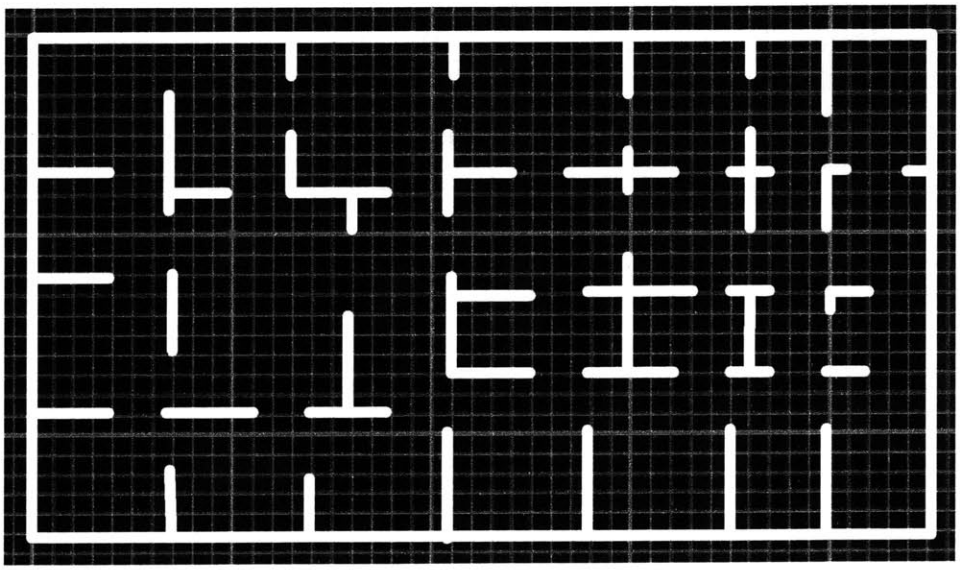
NE43Floor8



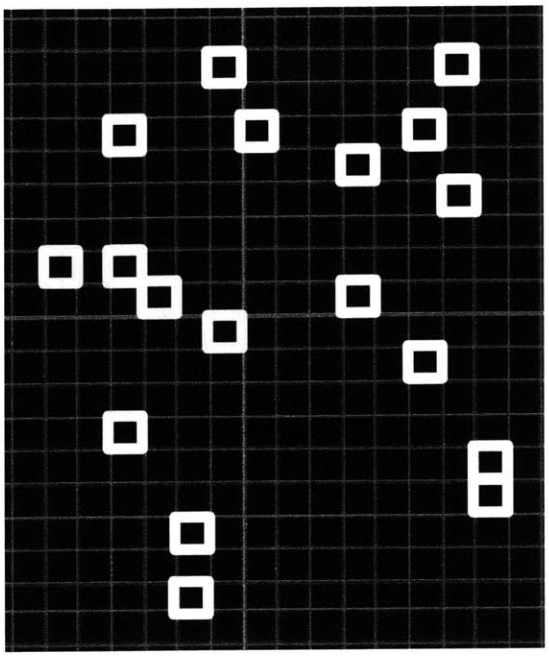
Building10Floor1



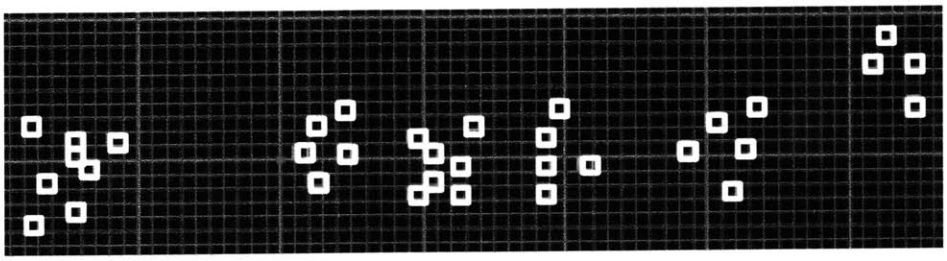
15by15Room



25by45Room



RandomRocks

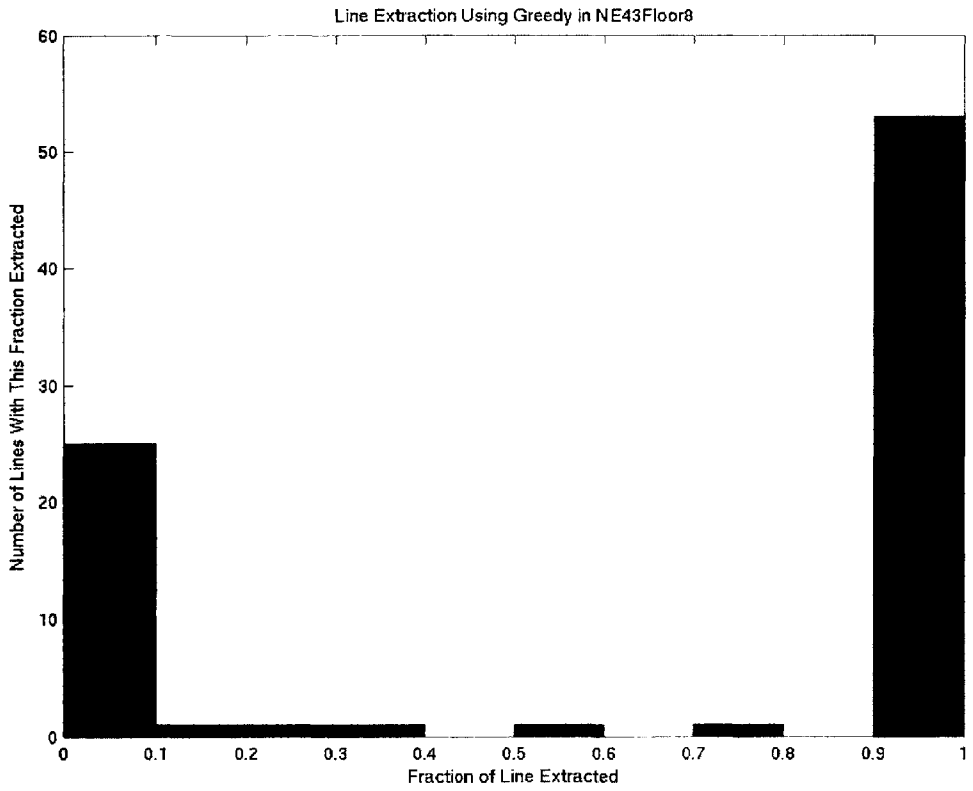


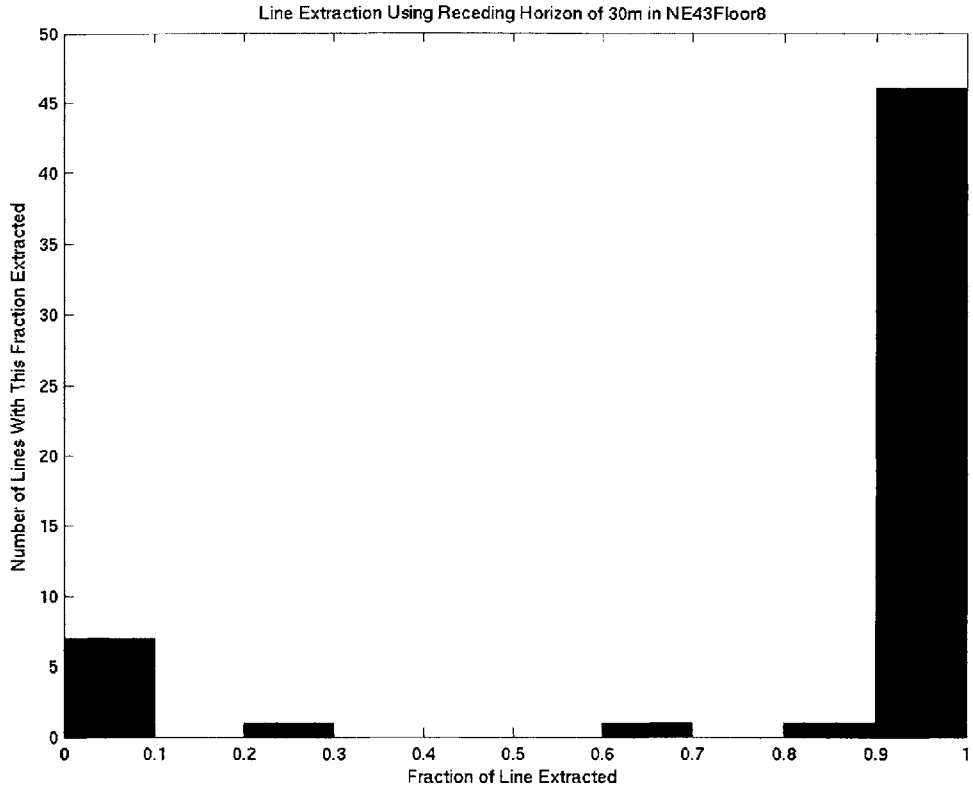
StructuredRocks

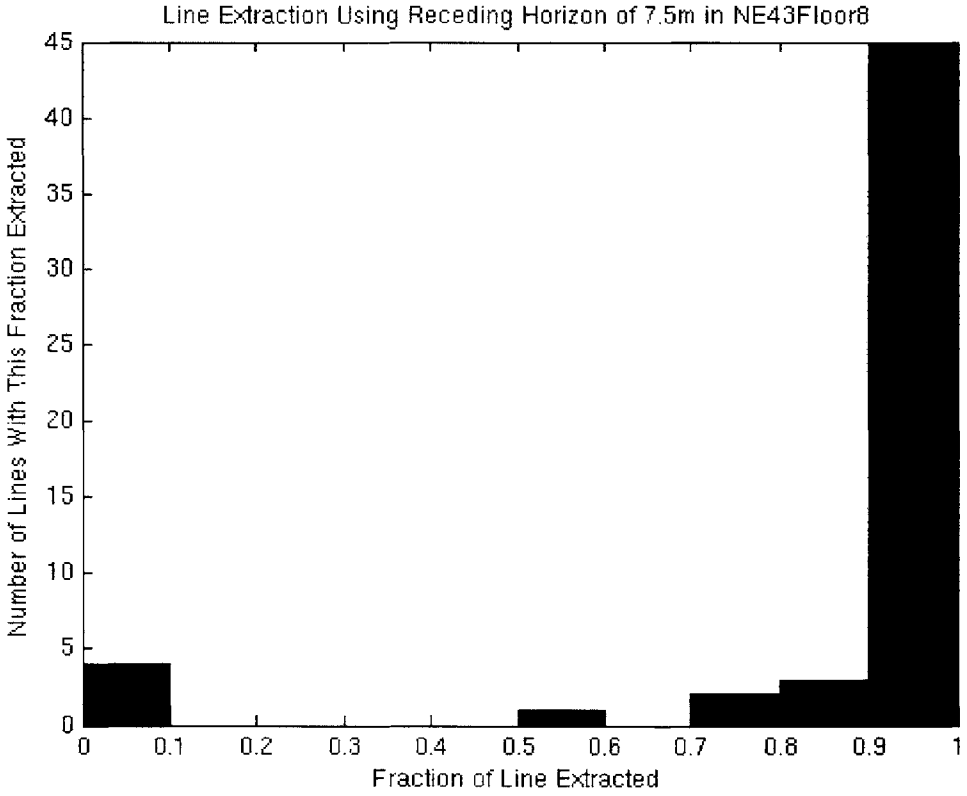
Appendix B: Line Extraction Histograms

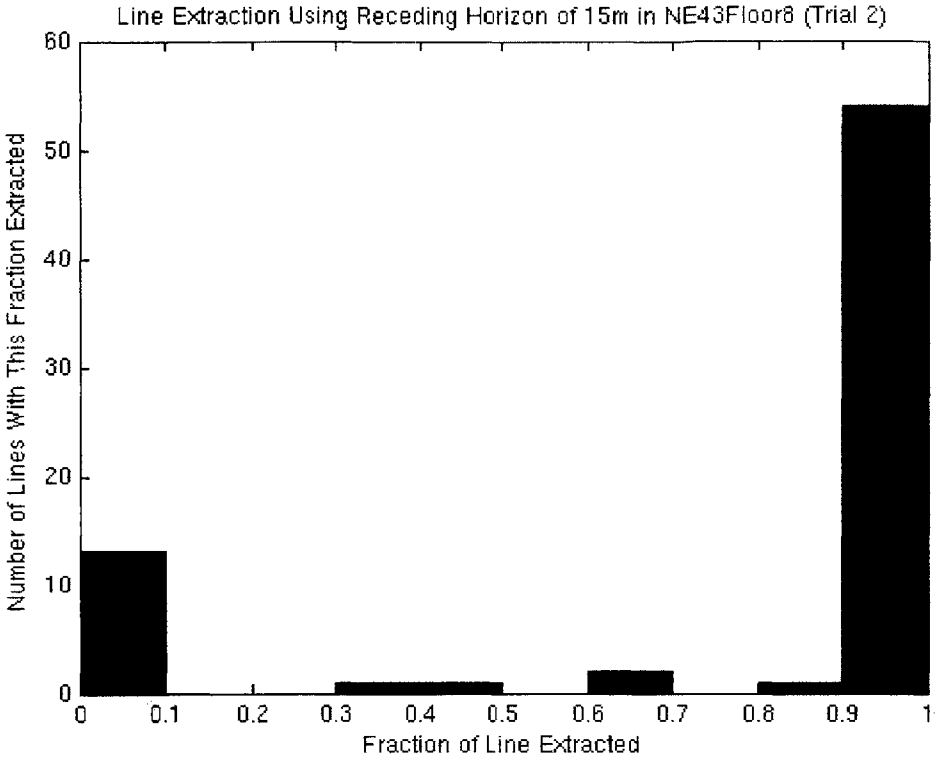
These histograms depict how thoroughly the robot explored its environment in the experiments described in Chapter 6. Specifically, in each trial, we defined the region of the environment that the robot explored (see Chapter 6 for an explanation of how we define this region). For each line within this region, we then calculated the fraction of the line that the robot extracted. The histograms below show the distribution of these fractions for each trial.

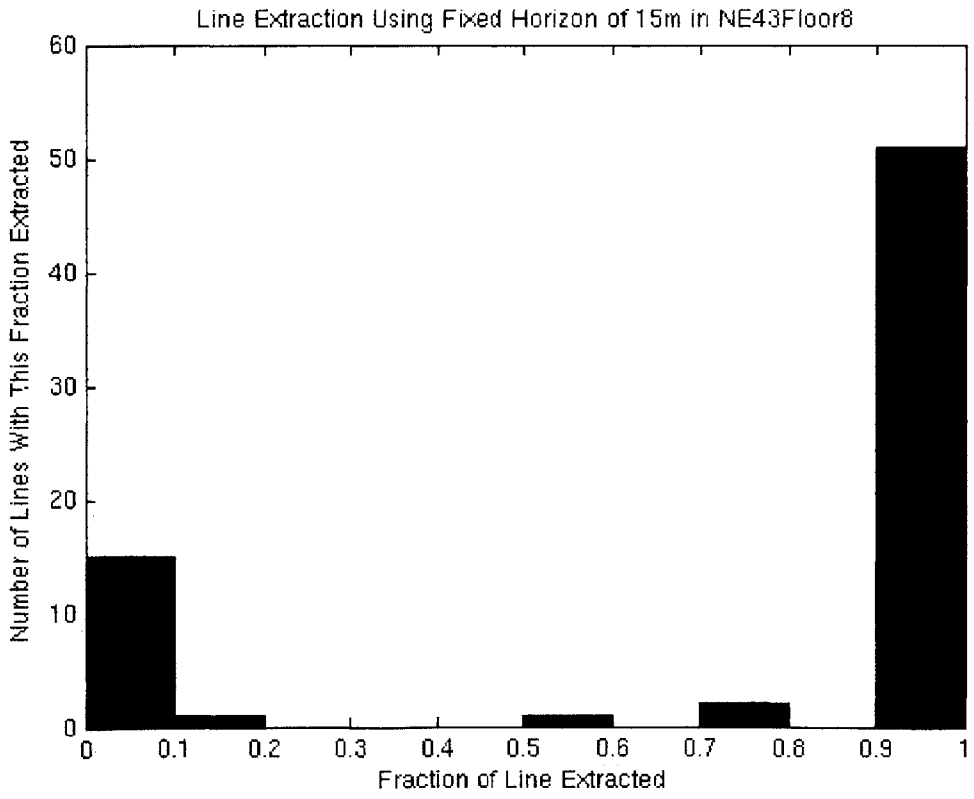
NE43Floor8 Trials

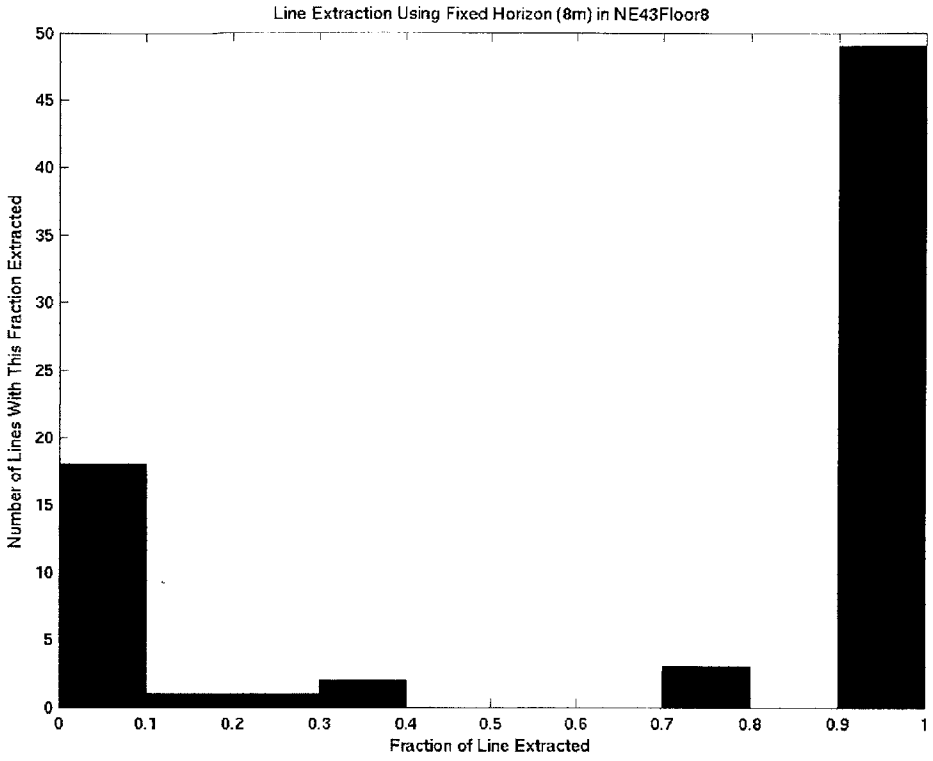


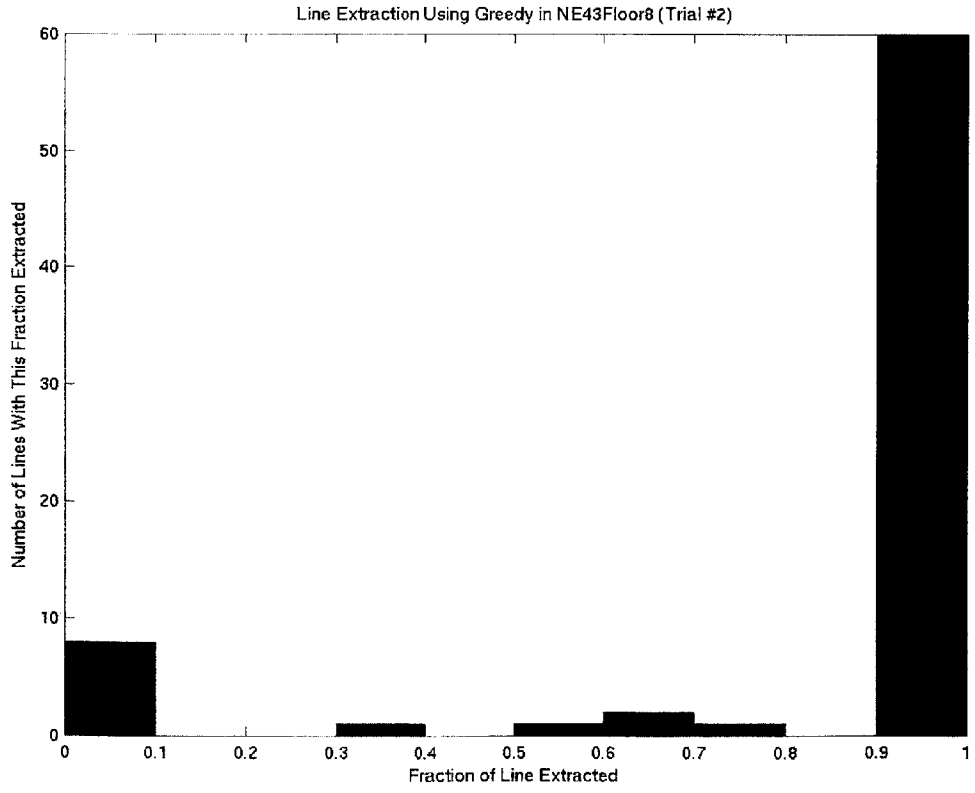

Line Extraction Using Receding Horizon of 30m in NE43Floor8
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
45
50
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Line Extraction Using Receding Horizon of 7.5m in NE43Floor8
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
45
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

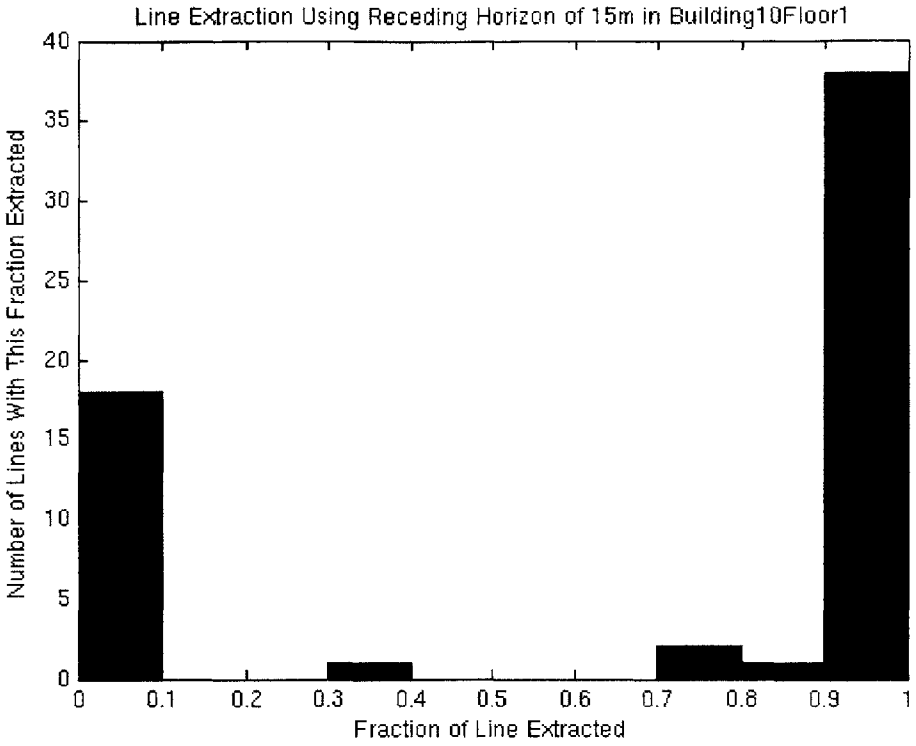

Line Extraction Using Receding Horizon of 15m in NE43Floor8 (Trial 2)
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
10
20
30
40
50
60
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

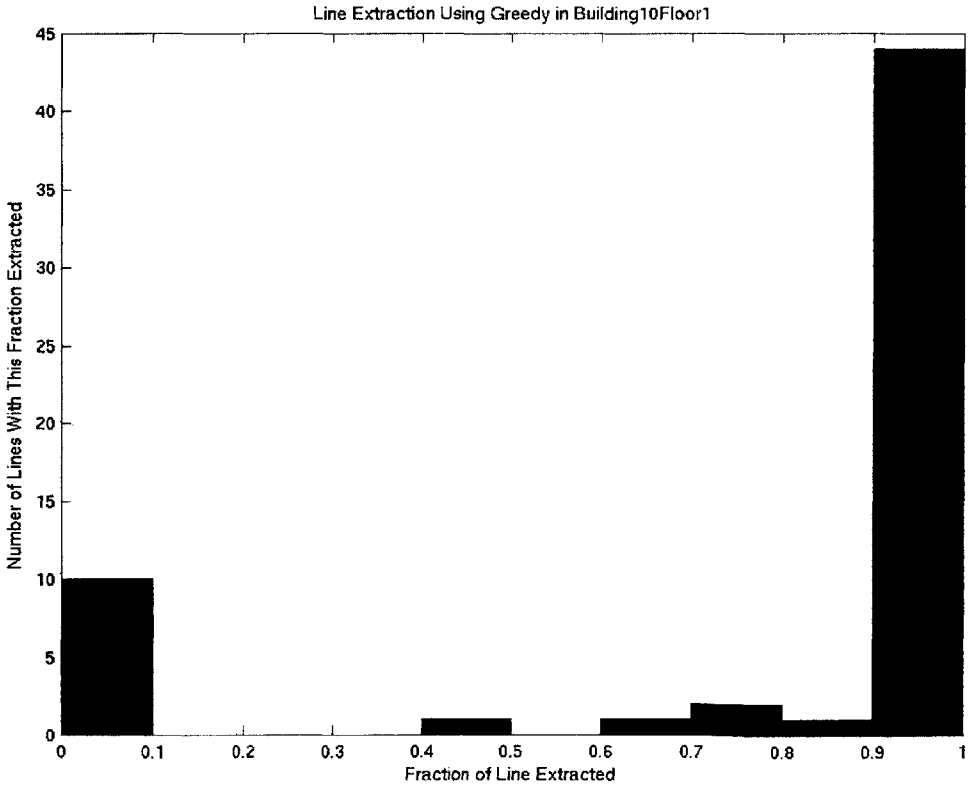

Line Extraction Using Fixed Horizon of 15m in NE43Floor8
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
10
20
30
40
50
60
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

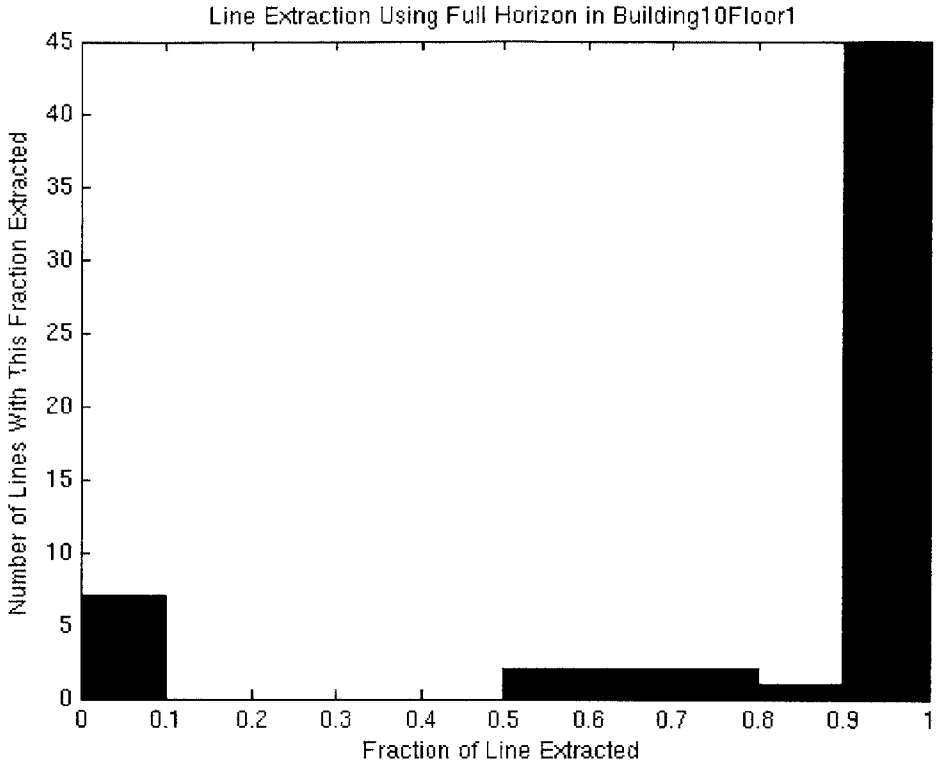

Line Extraction Using Fixed Horizon (8m) in NE43Floor8
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
45
50
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

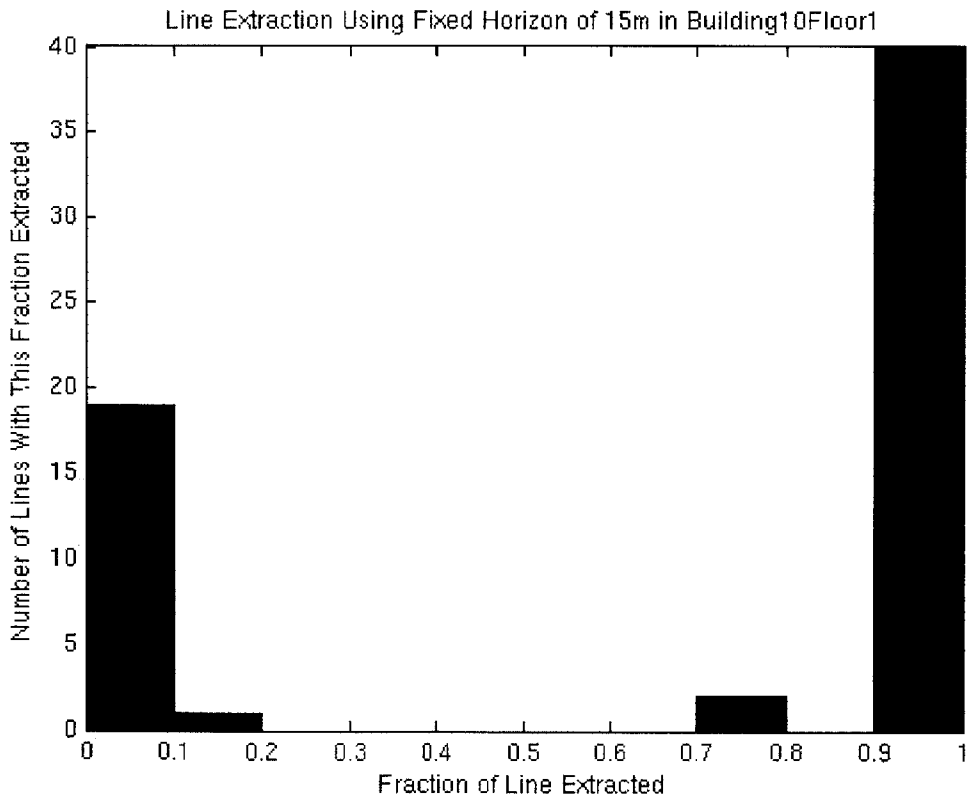

Line Extraction Using Greedy in NE43Floor8 (Trial #2)
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
10
20
30
40
50
60
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

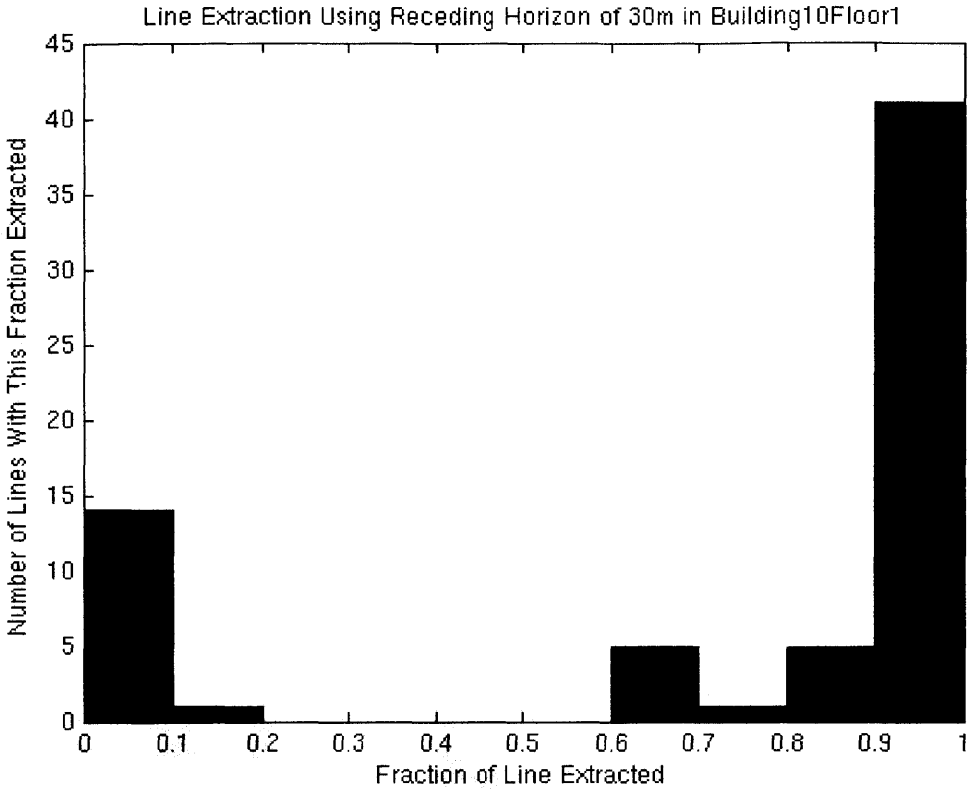
Building10Floor1 Trials



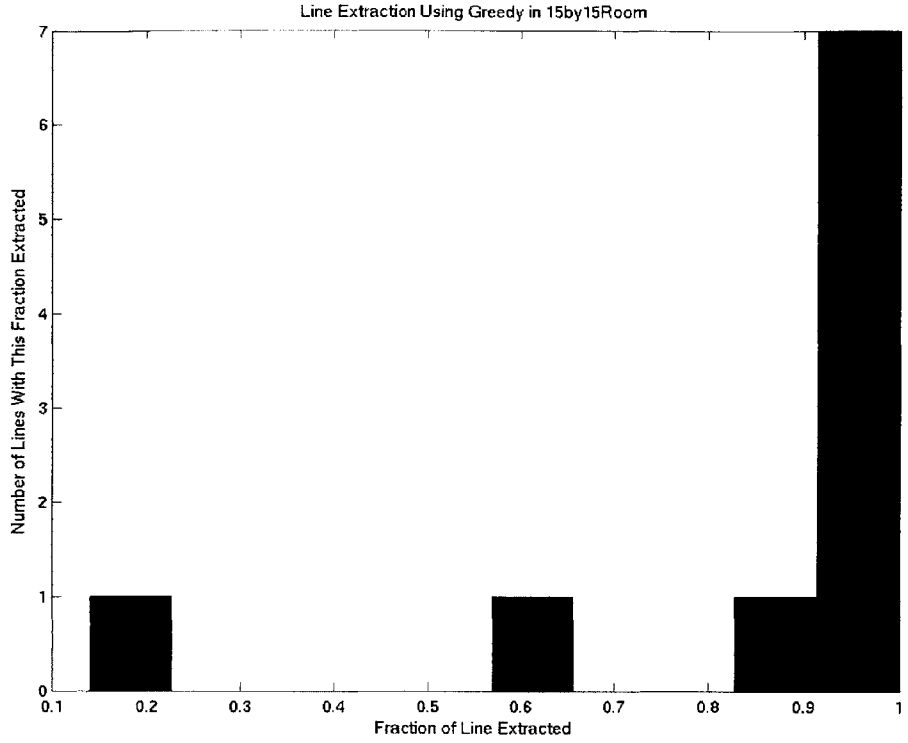


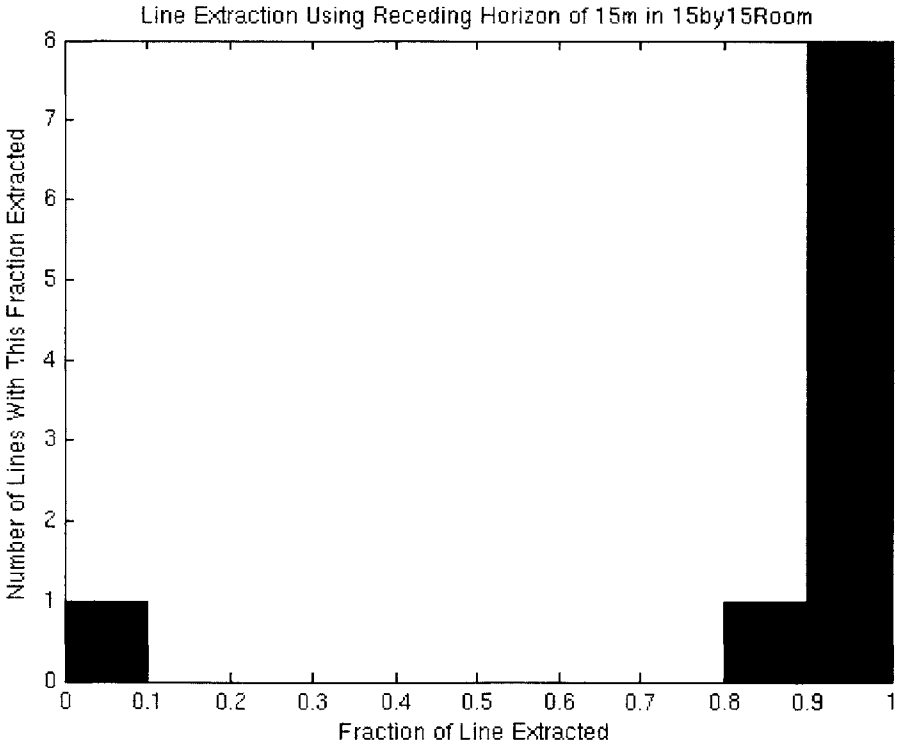

Line Extraction Using Full Horizon in Building10Floor1
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
45
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

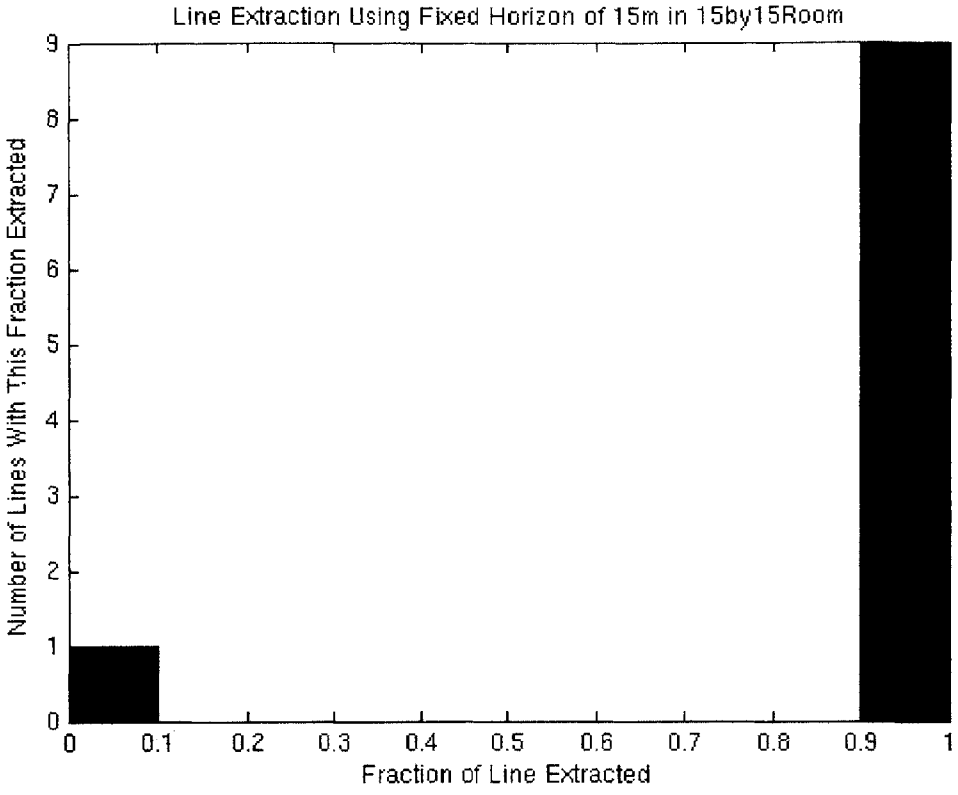

Line Extraction Using Fixed Horizon of 15m in Building10Floor1
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

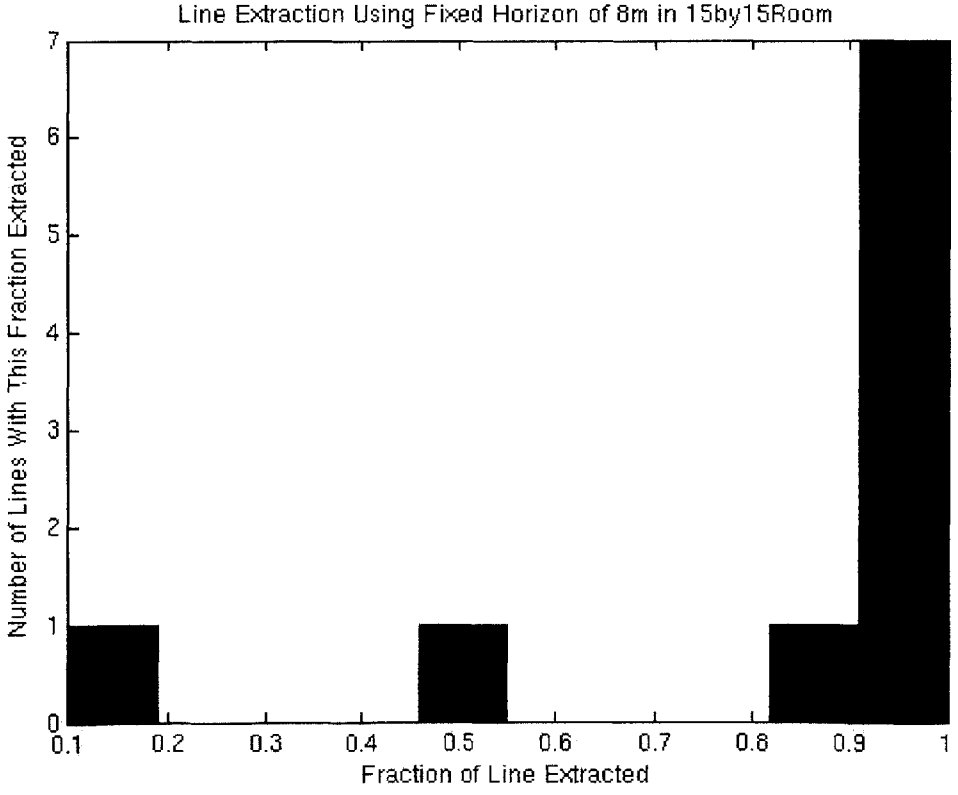

Line Extraction Using Receding Horizon of 30m in Building10Floor1
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
40
45
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

15by15Room Trials

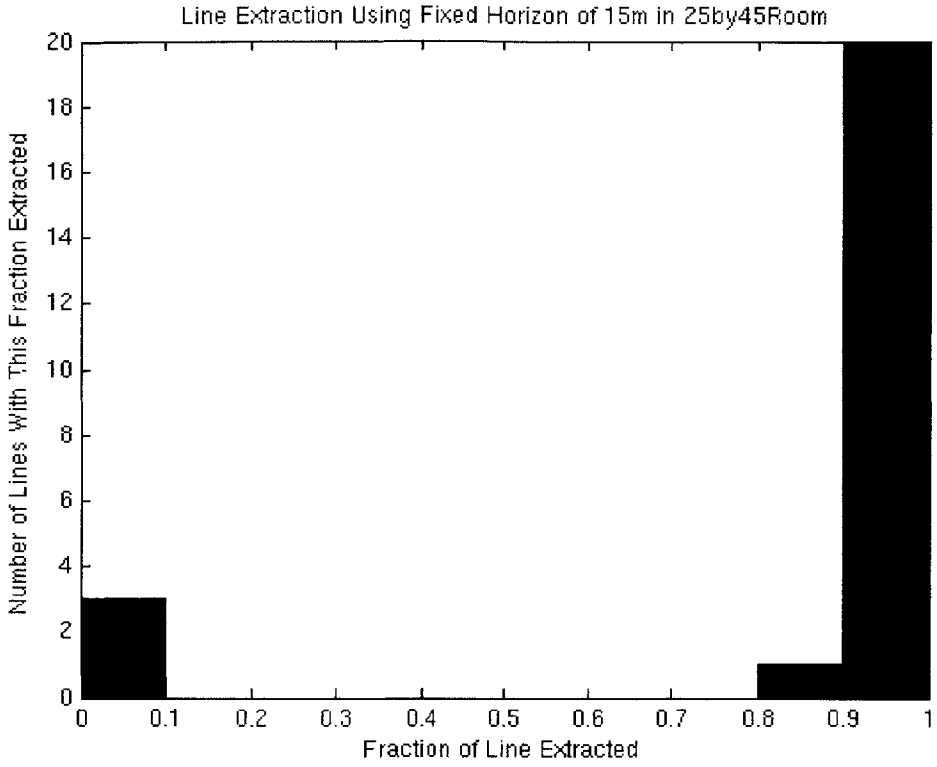


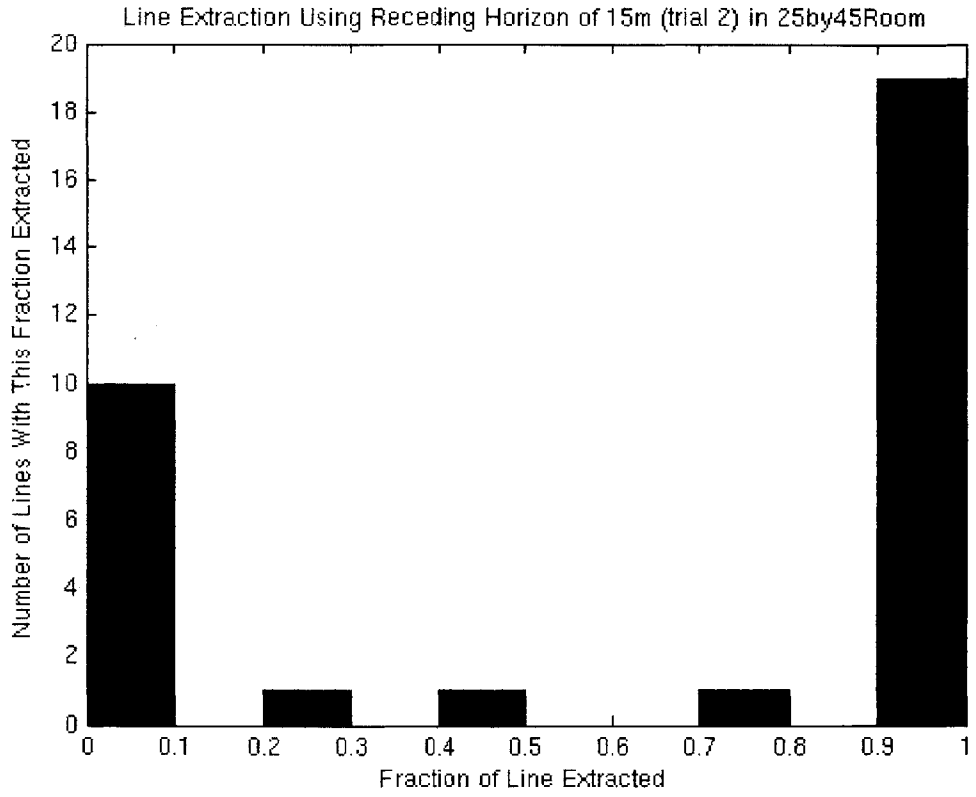


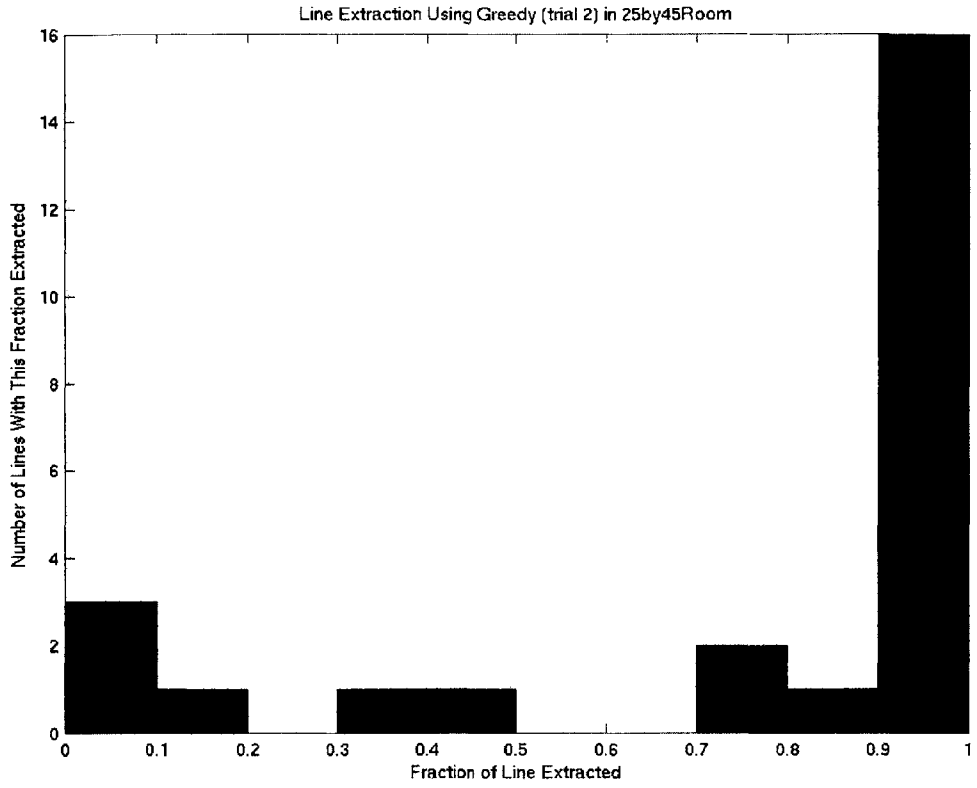

Line Extraction Using Fixed Horizon of 15m in 15by15Room
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
1
2
3
4
5
6
7
8
9
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

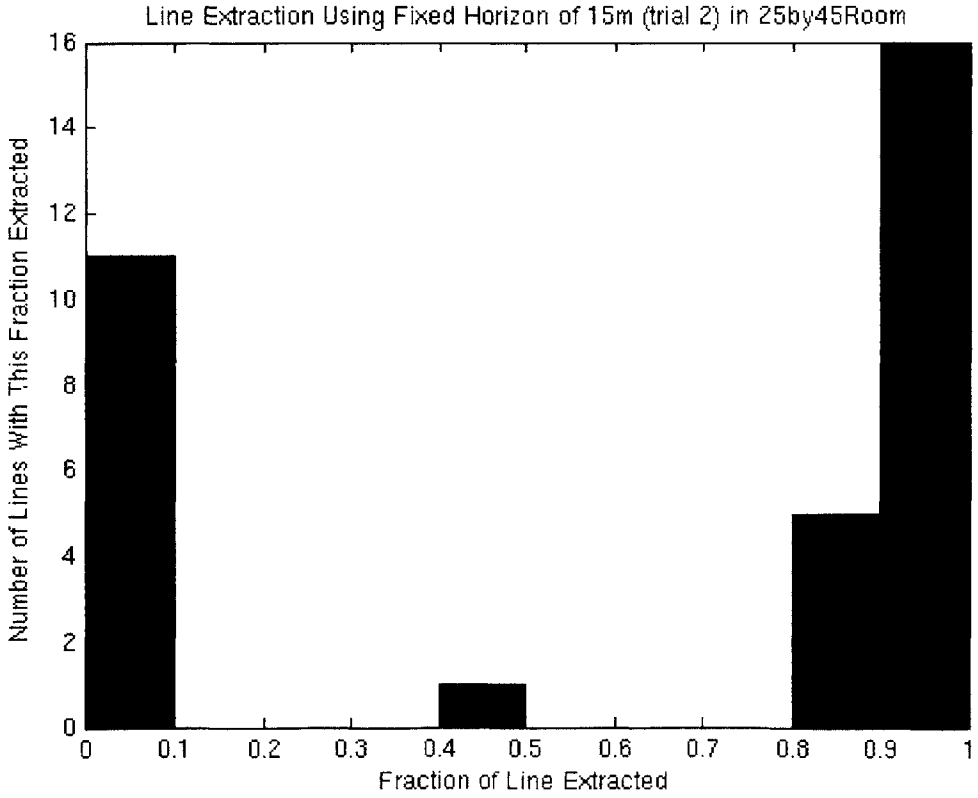

Line Extraction Using Fixed Horizon of 8m in 15by15Room
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
1
2
3
4
5
6
7
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

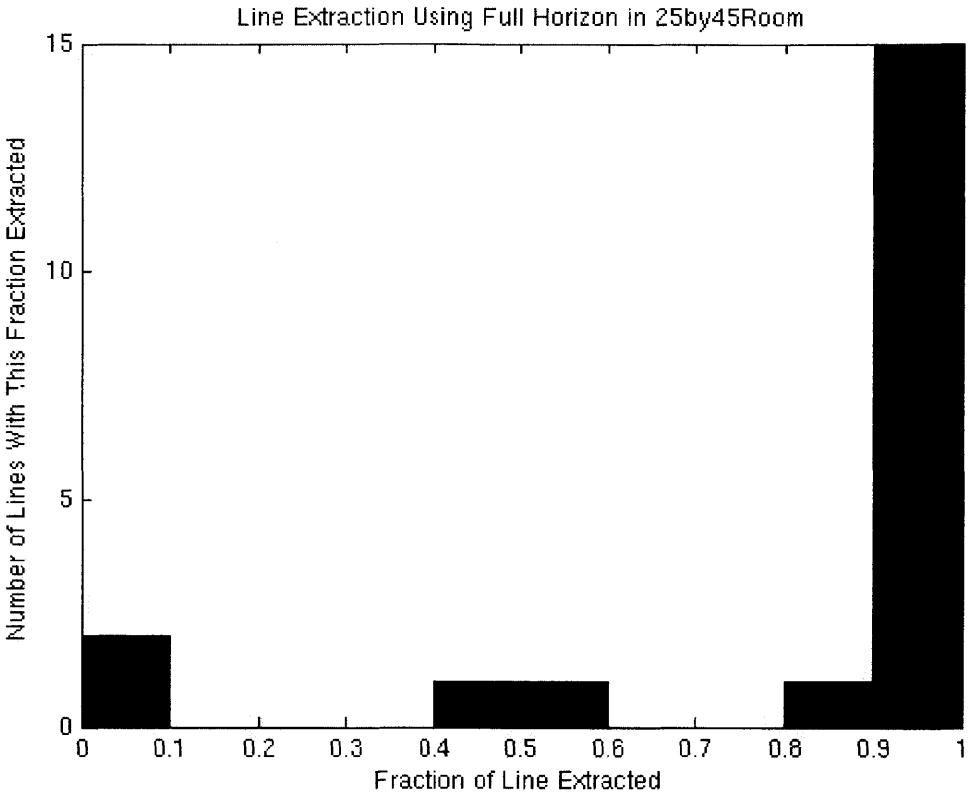
25by45Room Trials



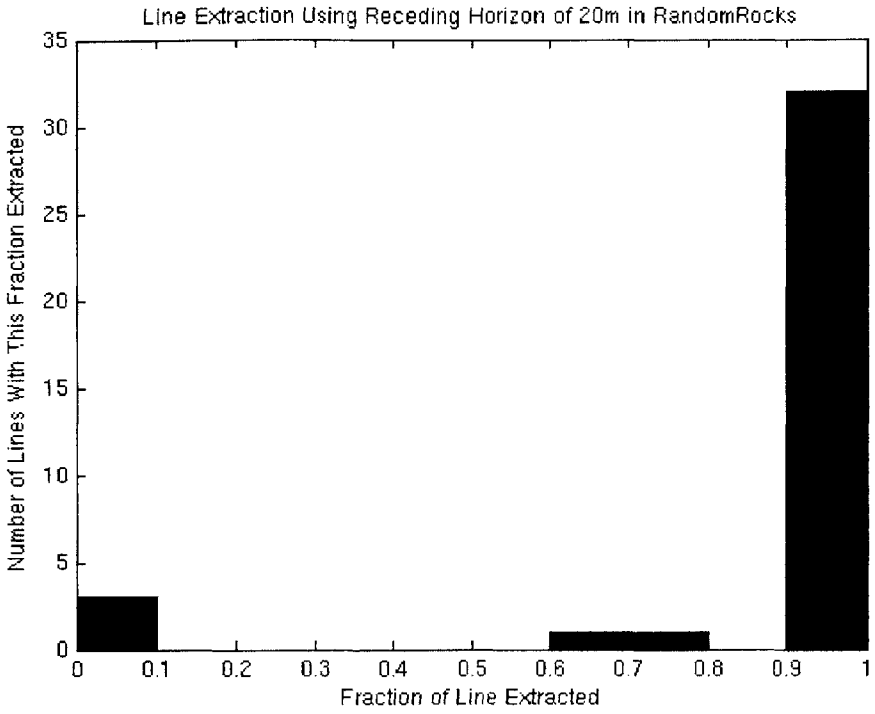


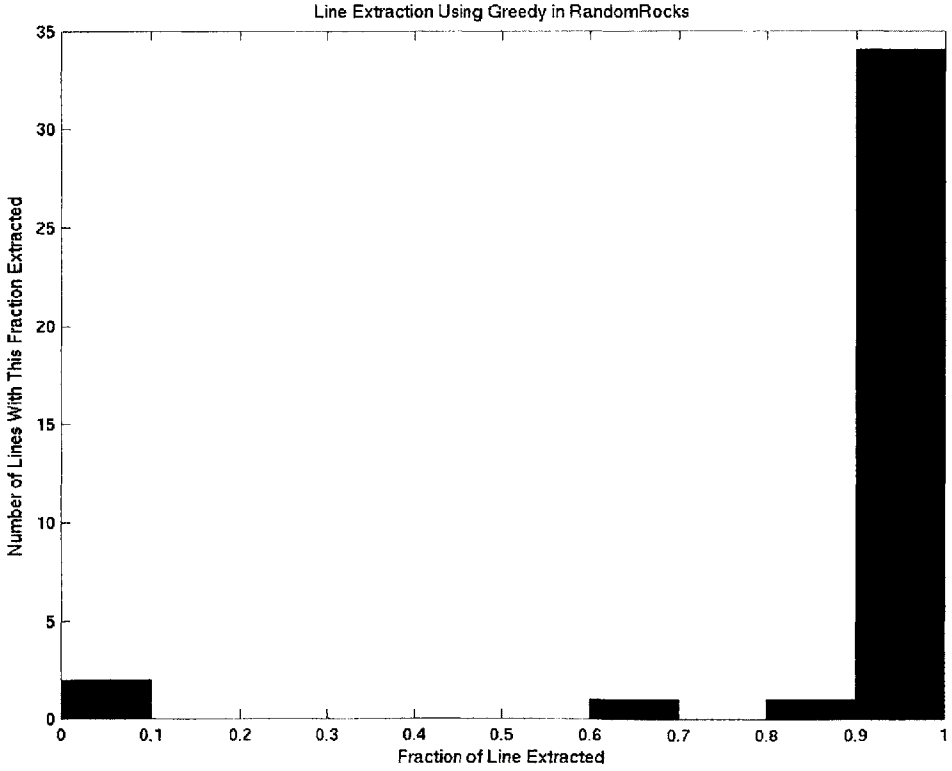

Line Extraction Using Greedy (trial 2) in 25by45Room
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
2
4
6
8
10
12
14
16
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

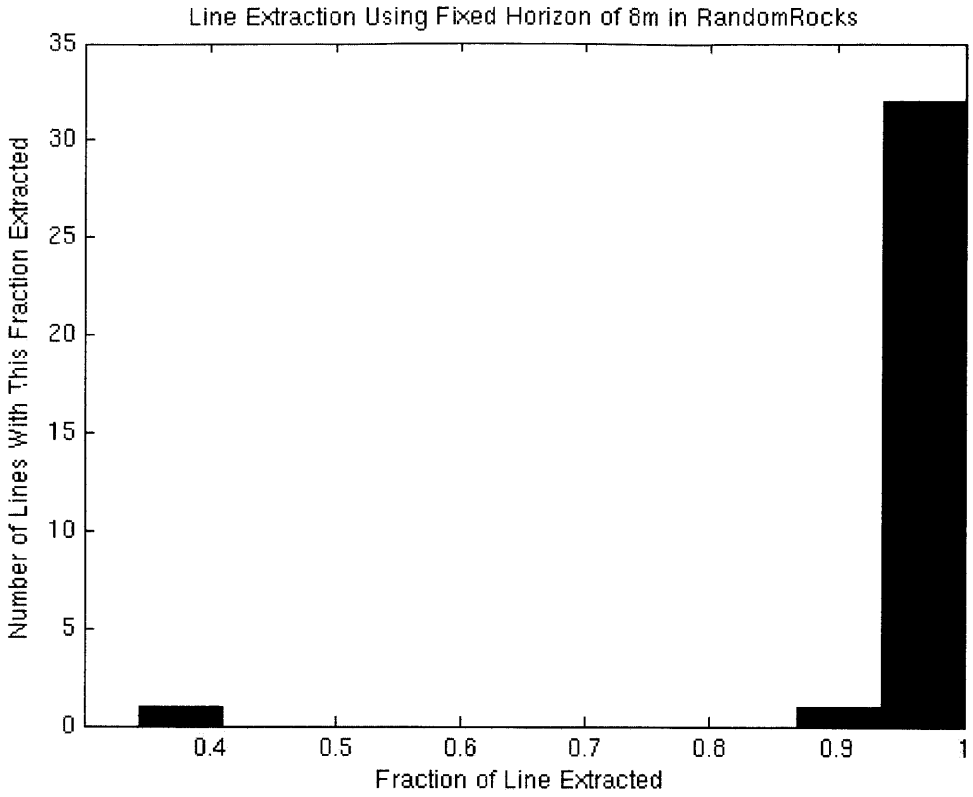

Line Extraction Using Fixed Horizon of 15m (trial 2) in 25by45Room
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
2
4
6
8
10
12
14
16
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Line Extraction Using Full Horizon in 25by45Room
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

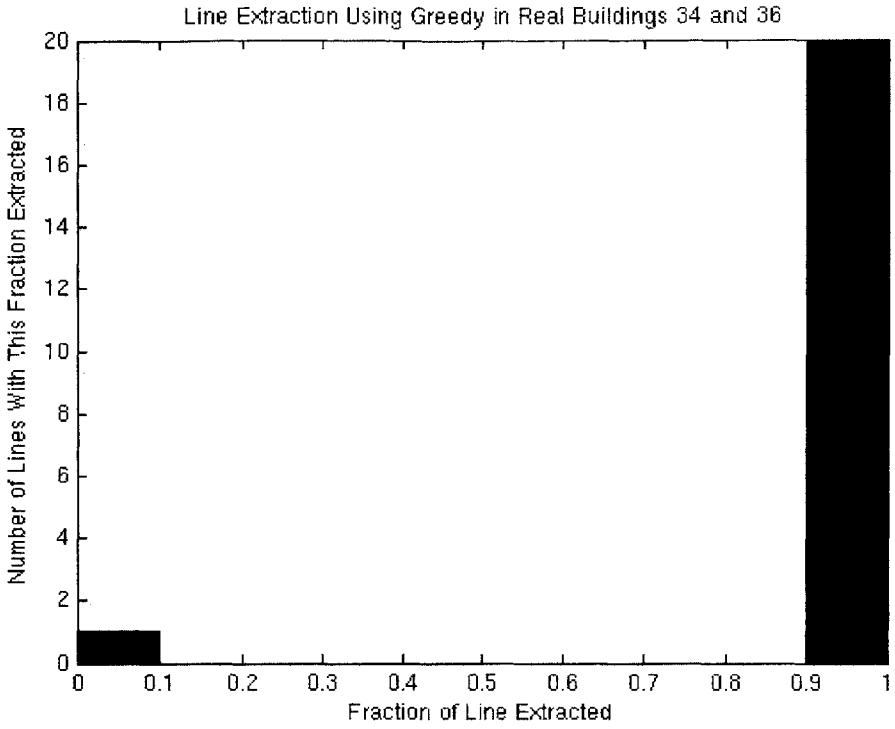
RandomRocks Trials

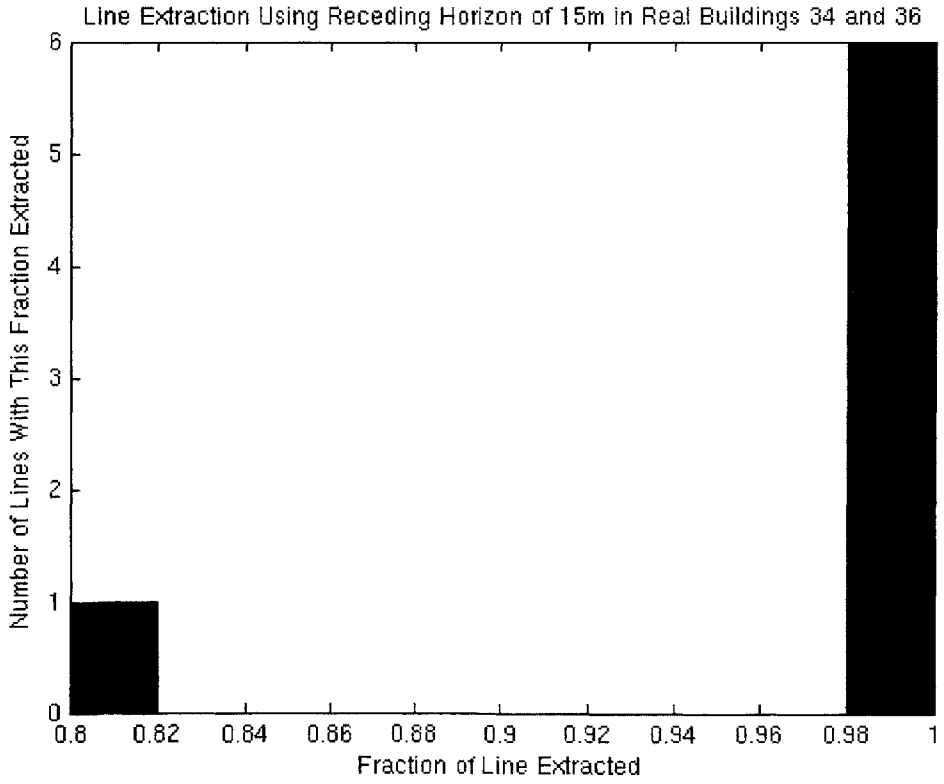


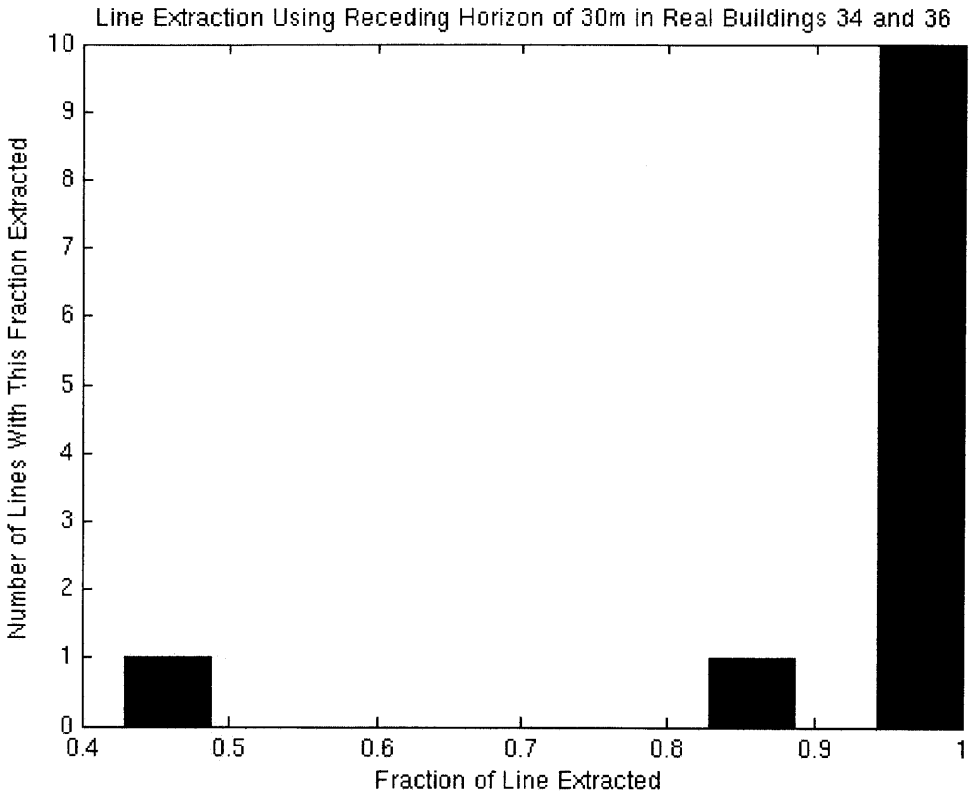



Line Extraction Using Fixed Horizon of 8m in RandomRocks
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
5
10
15
20
25
30
35
0.4
0.5
0.6
0.7
0.8
0.9
1

Real Buildings 34 and 36 Trials






Line Extraction Using Receding Horizon of 30m in Real Buildings 34 and 36
Number of Lines With This Fraction Extracted
Fraction of Line Extracted
0
1
2
3
4
5
6
7
8
9
10
0.4
0.5
0.6
0.7
0.8
0.9
1

References

[1] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook, "On the solution of traveling salesman problems," *Documenta Mathematica*, extra volume ICM III, pp. 645-656, 1998.

[2] R. Bauer and W. D. Rencken, "Sonar feature based exploration," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1995, vol. 1, pp. 148-153.

[3] J. G. Bellingham, C. A. Goudey, T. R. Consi, J. W. Bales, D. K. Atwood, J. J. Leonard and C. Chryssostomidis, "A second generation survey AUV," in *Proc. IEEE Conference on Autonomous Underwater Vehicles*, pp. 148-155, 1994.

[4] J. G. Bellingham and J. S. Willcox, "Optimizing AUV oceanographic surveys," in *Proc. IEEE Symposium on Autonomous Underwater Vehicle Technology*, 1996, pp. 391–398.

[5] R. E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geomtetry*. Berlin; New York: Springer-Verlag, 2000.

[7] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller, "An Atlas framework for scalable mapping," in *Proc. IEEE International Conference on Robotics and Automation*, 2003, pp. 1899–1906.

[8] M. Bosse, P. M. Newman, J. J. Leonard, and S. Teller, "SLAM in large-scale cyclic environments using the Atlas framework." To appear in the International Journal of Robotics Research.

[9] F. Bourgault, A. A. Makarenko, S. B. Williams, B. Grocholsky, and H. F. Durrant-Whyte, "Information based adaptive robotic exploration," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and System*, 2002, vol. 1, pp. 540–545.

[10] W. Burgard, D. Fox, M. Moors, R. Simmons, and S. Thrun, "Collaborative multi-robot exploration," in *Proc. IEEE International Conference on Robotics and Automation*, 2000, pp. 476-481.

[11] J. Casper and R. R. Murphy, "Human-robot interactions during the robot-assisted urban search and rescue response at the World Trade Center," *IEEE Transactions on Systems, Man and Cybernetics, Part B*, vol. 33, no. 3, June, pp. 367–385, 2003.

[12] M. W. M. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 3, June, pp. 229-241, 2001.

[13] A. Elfes, "Sonar-based real-world mapping and navigation," *IEEE Journal of Robotics and Automation*, vol. 3, no. 3, June, pp. 249-265, 1987.

[14] H. J. S. Feder, J. J. Leonard and C. M. Smith. "Adaptive mobile robot navigation and mapping," *International Journal of Robotics Research*, Special Issue on Field and Service Robotics, vol. 18, no. 7, July, pp. 650-668, 1999.

[15] M. Fischetti, J. J. Salazar-Gonzalez, and P. Toth, "Solving the orienteering problem through branch-and-cut," *INFORMS Journal on Computing*, vol. 10, no. 2, pp. 133-148, 1998.

[16] M. Gendreau, G. Laporte, and F. Semet, "A branch-and-cut algorithm for the undirected selective traveling salesman problem," *Networks*, vol. 32, pp. 263-273, 1998.

[17] B. L. Golden, L. Levy, and R. Vohra, "The orienteering problem," *Naval Res. Log.*, vol. 34, pp. 307-318, 1987.

[18] H. Gonzalez-Banos and J. Latombe, "Navigation Strategies for Exploring Indoor Environments," *The International Journal of Robotics Research*, vol. 21, no. 10-11, October-November, pp. 829-848, 2002.

[19] G. Gutin and P. Punnen, Eds., *The traveling salesman problem and its variations*. Dordrecht; Boston: Kluwer Academic Publishers, 2002.

[20] J-S. Gutmann and K. Konolige, "Incremental mapping of large cyclic environments," in *International Symposium on Computational Intelligenc in Robotics and Automation*, 1999.

[21] D. Hahnel, D. Schulz, and W. Burgard, "Map building with mobile robots in populated environments," in *Proc. IEEE International Conference on Intelligent Robots and Systems*, 2002, vol. 1, pp. 496-501

[22] M. Hayes and J. M. Norman, "Dynamic programming in orienteering: Route choice and siting of controls," *J. Oper. Res. Soc.*, vol. 35, pp. 791-796, 1984.

[23] D. Heckerman, "A Tutorial on Learning with Bayesian Networks," in *Learning in Graphical Models*, M. I. Jordan, Ed. Dordrecht, Netherlands: Kluwer, 1998.

[24] J. Hsu and L. Hwang, "A graph-based exploration strategy of indoor environments by an autonomous mobile robot," in *Proc. IEEE International Conference on Robotics and Automation*, 1998, pp. 1262–1268.

[25] T. Huntsberger, H. Aghazarian, Y. Cheng, E. T. Baumgartner, E. Tunstel, C. Leger, A. Trebi-Ollennu, and P.S. Schenker, "Rover autonomy for long range navigation and science data acquisition on planetary surfaces," in *Proc. IEEE International Conference on Robotics and Automation*, 2002, vol. 3, pp. 3161–3168.

[26] R. Jonker and T. Volgenant, "Transforming asymmetric into symmetric traveling salesman problems," *Operations Research Letters*, vol. 2, no. 4, November, pp. 161-163, 1983.

[27] L. P. Kaelbling, M. L. Littman, and A. W. Moore, ``Reinforcement learning: a survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285, 1996.

[28] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566-580, 1996.

[29] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, pp. 100-117, 1987.

[30] M. Kontitsis, K. P. Valavanis, and N. Tsoureloudis, "A UAV vision system for airborne surveillance," in *Proc. IEEE International Conference on Robotics and Automation*, 2004, vol. 1, pp. 77 – 83.

[31] R. Krishnan, "Solving hybrid decision-control problems through conflict-directed branch & bound," Masters thesis, Massachusetts Institute of Technology, 2004.

[32] G. Laporte and S. Martello, "The selective traveling salesman problem," *Discrete Applied Mathematics*, vol. 26, no. 2-3, March, pp. 193-207, 1990.

[33] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, May, pp. 378-400, 2001.

[34] J. J. Leonard and P. M. Newman, "Consistent, convergent, and constant-time SLAM," *International Joint Conference on Artificial Intelligence* Acapulco, Mexico, 2003.

[35] O. Martin, S. W. Otto, and E. W. Felten, "Large-step markov chains for the tsp incorporating local search heuristics," *Operations Research Letters*, vol. 11, pp. 219-224, 1992.

[36] P. Moutarlier and R. Chatila, "Stochastic mulisensory data fusion for mobile robot location and environment modeling," in *Robotics Research: 5th International Symposium*, H. Miura and S. Arimoto, eds. Cambridge, MA: MIT Press, 1989, pp. 85-94.

[37] P. Newman, M. Bosse, and J. Leonard, "Autonomous feature-based exploration," in *Proc. IEEE International Conference on Robotics and Automation*, 2003, pp. 1234 – 1240.

[38] P. Newman, J. Leonard, J. D. Tardos, and J. Neira, "Explore and return: experimental validation of real-time concurrent mapping and localization," in *Proc. IEEE International Conference on Robotics and Automation*, 2002, pp. 1802-1809.

[39] J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun. "Towards robotic assistants in nursing homes: challenges and results," *Robotics and Autonomous Systems*, vol. 42, no. 3-4, March, pp. 271-281, 2003.

[40] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, pp. 1389-1401, 1957.

[41] A. P. Punnen, "The Traveling Salesman Problem: Applications, Formulations and Variations," in The Traveling Salesman Problem and its Variations, G. Gutin and P. Punen, Eds. Dordrecht; Boston: Kluwer Academic Publishers, 2002.

[42] R. Ramesh, Y. Yong-Seok, and M. H. Karwan, "An optimal algorithm for the orienteering tour problem," *ORSA Journal on Computing*, vol. 4, pp. 155-165, 1992.

[43] R. Sim and N. Roy, "Global A-optimal robot exploration in SLAM." In submission.

[44] R. Smith, M. Self, and P. Cheesemen, "Estimating uncertain spatial relationships in robotics," in *Autonomous Robot Vehicles*, L. Cox and G. Wilfong, Eds. Springer-Verlag, 1990, pp. 167-193.

[45] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proc. IEEE International Conference on Robotics and Automation*, 1994, pp. 3310-3317.

[46] R. Stokey, T. Austin, B. Allen, N. Forrester, E. Gifford, R. Goldsborough, G. Packard, M. Purcell, and C. von Alt, "Very shallow water mine countermeasures using the REMUS AUV: a practical approach yielding accurate results," in *Proc. MTS/IEEE Conference and Exhibition OCEANS*, 2001, vol. 1, pp. 149-156.

[47] S. Thrun, "Exploration and model building in mobile robot domains," in *Proc. IEEE International Conference on Neural Networks*, 1993, pp. 175-180.

[48] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artificial Intelligence*, vol. 99, no. 1, pp. 21-71, 1998.

[49] S. Thrun, "A probabilistic online mapping algorithm for teams of mobile robots," *International Journal of Robotics Research*, vol. 20, no. 5, May, pp. 335-363, 2001.

[50] T. Tsiligirides, "Heuristic methods applied to orienteering," *J. Oper. Res. Soc.*, vol 35, pp. 797-809, 1984.

[51] M. A. Wesley and T. Lozano-Perez, "An algorithm for planning collision-free paths among polyhedral objects," *Communications of the ACM,* vol. 22, no. 10, pp. 560-570, 1979.

[52] B. C. Williams, and R. Ragno, January 2003, "Conflict-directed A* and its role in model-based embedded systems," to appear in the *Journal of Discrete Applied Math* Special Issue on Theory and Applications of Satisfiability Testing.

[53] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *Proc. IEEE International Symposium on Computational Intelligence in Robotics and Automation*, 1997, pp. 146-151.

[54] Concorde is a TSP solver that is free for academic research. It can be downloaded at http://www.tsp.gatech.edu//concorde.html