Learning a Motion Policy to Navigate Environments With Structured Uncertainty

by

Florence Tsang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

Statement of Contributions

Florence Tsang was the sole author for Chapters 1, 2, 3, 5, and 6 which were written under the supervision of Dr. Stephen Smith and were not written for publication.

The research presented in chapter 4 was conducted by Florence Tsang and Ryan A. Macdonald under the supervision of Dr. Stephen Smith. Ryan A. Macdonald wrote the problem setup (Section 4.2). Florence Tsang revised the solution approach (Section 4.3) and wrote the remainder of the chapter. Florence Tsang also did the coding, data collection, and data analysis.

Chapter 4 was written for publication.

Citations:

Abstract

Navigating in uncertain environments is a fundamental ability that robots must have in many applications such as moving goods in a warehouse or transporting materials in a hospital. While much work has been done on navigation that reacts to unexpected obstacles, there is a lack of research in learning to predict where obstacles may appear based on historical data and utilizing those predictions to form better plans for navigation. This may increase the efficiency of a robot that has been working in the same environment for a long period of time.

This thesis first introduces the Learned Reactive Planning Problem (LRPP) that formalizes the above problem and then proposes a method to capture past obstacle information and their correlations. We introduce an algorithm that uses this information to make predictions about the environment and forms a plan for future navigation. The plan balances exploiting obstacle correlations (ie. observing obstacle A is present means obstacle B is present as well) and moving towards the goal. Our experiments in an idealized simulation show promising results of the robot outperforming a commonly used optimistic algorithm.

Second, we introduce the Learn a Motion Policy (LAMP) framework that can be added to navigation stacks on real robots. This framework aims to move the problem of predicting and navigating through uncertainties from idealized simulations to realistic settings. Our simulation results in Gazebo and experiments on a real robot show that the LAMP framework has potential to improve upon existing navigation stacks as it confirms the results from the idealized simulation, while also highlighting challenges that still need to be addressed.

Acknowledgements

I would like to thank my supervisor, Stephen Smith for his patience and guidance during my time as a Master's student at the University of Waterloo. His insight and advice was invaluable to my growth in knowledge in the field of robotics and to the research that went into this thesis.

I would also like to thank everyone in the Smith Autonomy Lab for their support and help for the last two years. In particular, I would like to thank Tristan Walker, a fellow Master's student, for all his assistance in the implementation of my experiments and brainstorming with me on how to adapt the initial algorithm to realistic scenarios. As well as a big thank you to Olzhas Adiyatov and Jean-Luc Bastarache for spending a whole Saturday with me to aid in my experiment with the Jackal.

I am grateful to Alex Werner and Brandon J. DeHart, for without them, testing my algorithm on a real robot would have still just been a far away dream.

I would like to thank all my friends and family for their love and encouragement. I am especially thankful to Emily Leung, Dennis Tsang, Enoch Tsang, Geoffrey Mah, Sarah Ng, and Jenny Martin for being with me in the highs and lows of this journey.

Finally, I would like to acknowledge my Creator, whose divine inspiration and strength sustained my spirit from the beginning to the end.

Dedication

This is dedicated to my younger brother Enoch Tsang. His passion in everything he does inspired me to challenge my own limitations and grow from the experience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Motion planning is a long standing problem in the robotics research community, as it is an essential part of what it means for a robot to be autonomous. Although much progress has been made in the past few decades, motion planning covers a very wide breadth of topics, and as the most recent disasters with self-driving vehicles would inform us, there is still room for great improvement. On the most basic, but by no means trivial, level, there is the matter of computing an appropriate trajectory for reaching a specific point in space. This can be extended to path following, where a series of trajectories are computed to follow an arbitrary path. On a higher level, there has been work on generating these paths to reach some objective, which is captured by a cost or a value function.

There are many different objectives a robot could have, with many areas of research for specific problems. For example, some robots are used for exploring unknown areas, in which case their objectives would be to maximize new map information while minimizing time or effort [5, 60]. Others might be patrolling a certain area [42], or collecting measurements at specific locations [64, 68] under time or fuel constraints. There are many more, but what they all have in common is the basic aim to move from one location to another.

We will refer to the task of moving from a start location to a goal location as *navigation*. While navigation seems to be a simple task in comparison to the ones mentioned earlier, determining optimal paths remains a challenge. If the environment is fully known (i.e. all the obstacle information is known beforehand) then there are multitudes of efficient solutions that use sampling (e.g. PRM [28], RRT [40]). However, even this case has its challenges when noise in the sensors and controls are considered. As a result, algorithms like the one in [51] have been proposed to be more robust to this type of internal noise. But for robots to really become ubiquitous in our homes, hospitals, restaurants, and workplaces, they must also be able to handle uncertainties in these dynamic environments that are beyond just sensor noise.

Uncertainties in an environment can be divided into the following types: dynamic and semi-static. Dynamic uncertainties refer to obstacles that are currently in motion, this can be other humans, animals, vehicles, etc. Semi-static uncertainties refer to obstacles such as doors, beds, or boxes that are generally not in motion when the robot passes by, but could be moved the next time the robot passes the same area. These kind of uncertainties can block areas that were previously traversable and free areas that were not. *Uncertainty* in this work will refer to semi-static uncertainties, unless otherwise stated. For completion, static obstacles refer to obstacles that are in a floor plan such as walls. In this work, these obstacles are not expected to be uncertain, and could also be referred to as permanent obstacles.

A navigation system must have two capabilities,

- the ability to locate itself and its target in a model of the environment, commonly known as *localization*, and
- the ability to plan a route to its goal.

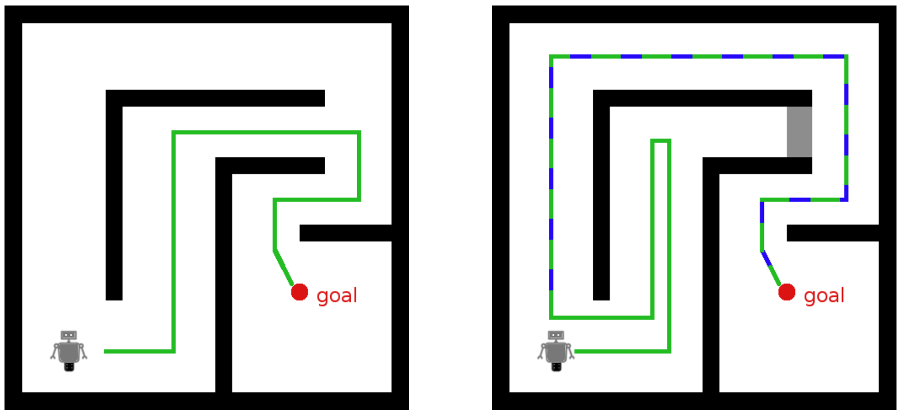
The robot will first need a model of the environment before it can navigate. There are different methods of processing sensor data before it is given to the planner. If the robot has a Light Detection and Ranging (LiDAR) sensor which measures the distance from the robot to obstacles via laser pulses, this sensor data can be fed directly into a neural network [26,65] which then outputs trajectories, alternatively the LiDAR data can be processed into a 2D or 3D occupancy grid [22,73]. A more advanced version of an occupancy grid is a costmap, which assigns different costs to cells (as opposed to a probability) based on the distance from obstacles and other factors. The most abstract representations compress the data into a graph with varying levels of geometrical interpretation (imagine how the accuracy in scale of hand-drawn maps can be varied, but can still be interpreted) [7, 35, 70]. The vertices can represent an area (for example, a room), and edges indicate if there exists a path between two vertices. There may be other constraints to decide whether an edge exists or not depending on the algorithm used to define the graph. In this thesis, the term *map* refers to an occupancy grid or a graph used to model an environment.

There are two ways to obtain a map, it could be given to the robot, or a more common approach is for the robot to generate its own map. The heavily researched simultaneous localization and mapping (SLAM) problem addresses the issue of generating and localizing in that same map. Surprisingly, there is not much work in translating a not-to-scale map into a model the robot can use. Brunner [9] did some work on learning to read maps viewed by a camera, while Kakuma [24] looked into extracting semantic information from floor plans. The most straight-forward method is providing a to-scale occupancy grid (either manually curated or generated by a SLAM algorithm) that can be used to localize the robot. While the map format is important in the latter half of this thesis, the specifics of map generation are not the focus, but the above information is useful to keep in mind.

Only once the robot has a map of the environment, and has localized itself, can any meaningful planning be done. When navigating without uncertainties, a planner can simply output a single path for the robot to follow to reach its goal. However, this is an insufficient plan in the presence of uncertainties, since there could exist a shorter path or the path cannot be fully executed because an unexpected obstacle is blocking the way. Therefore, what is needed is a plan on how to react in different circumstances. Plans that define how a robot should behave during navigation are referred to as *policies*.

A common approach to dealing with uncertainties while navigating is to plan a shortest path given the current map, if the path is blocked, then update the map and replan the shortest path with the updated map with algorithms such as D*Lite [30] or Anytime Dynamic A* [43]. These are known as *optimistic* policies. These policies are suitable for situations where the robot is not expected to operate in the same environment for a long period of time (e.g. search and rescue missions, exploration missions). However, from a long-term operation perspective (e.g. deployment in a mall or warehouse), this is not ideal for two reasons. The first reason is that there could be significant distortion of the map due to noise, however, this can be avoided by using a layered costmap [48] that differentiates between dynamic, semi-static, and static obstacles. Updates to the map can be made to the dynamic and/or semi-static layer, while maintaining and localizing to the static obstacle positions, avoiding distortion.

The second reason is that this approach only considers the given static obstacles and other obstacles within its immediate sensor range when forming its initial strategy, which could result in consistently 'bad' routes to the goal. Figure 1.1 illustrates an example of this.



(a) (b)

Figure 1.1: In (a), the robot takes the green path, which is the shortest path. This is also the map that the robot has to plan with. In (b), there is a semi-static obstacle which blocks the original shortest path, so the robot replans once it sees the obstacle, resulting in a longer path (green) than the optimal (blue).

Consider the case where the environment is as shown in figure 1.1b most of the time (say 99%) and only occasionally as shown in figure 1.1a (say 1%). Then the optimistic approach will continually attempt the green path when it executes the same navigation task, rather than considering that overall, it's better to take the blue path. One could say to use the map in figure 1.1b instead, but what if the probabilities were reversed? Then the robot will take the blue path when the green path (a) is frequently available. This raises two questions, how to learn these uncertainties, assuming there is some structure to them (i.e. they can be predicted), and how to compute a policy that minimizes the cost over a period of time by exploiting these learned uncertainties. From a high level, this is the problem that this thesis attempts to address.

1.1 Thesis Contributions

The following are the key contributions from this thesis:

- Chapter 4 proposes a method of learning the environment uncertainties given no prior uncertainty information and an algorithm for exploiting that information that minimizes the expected navigation cost over a given number of task executions. This work was published in [71].
- Chapter 5 investigates the challenges of implementing the system from chapter 4 onto a robot and proposes a modular framework that could be used for comparing different policies on robot hardware.

1.2 Outline

The remainder of this thesis is organized as follows:

Chapter 2 covers related work on learning in general and in navigation, navigating in uncertainties, and some more details on the different mapping representations.

Chapter 3 provides the general notation used throughout this thesis and information on previously researched problems that the reader should be aware of before continuing onto chapters 4 and 5.

Chapter 4 first proposes the learned reactive planning problem (LRPP), then proposes a method to capture environmental uncertainties when only given a base map with static obstacles (such as a floor plan), assuming the robot must execute navigation tasks in the same environment multiple times. Next, an algorithm is proposed to solve the LRPP using the previous method to learn the uncertainties and exploit them to minimize the expected cost for a specific navigation task, iteratively improving the policy over time. Finally, simulation results are presented to show the effectiveness of the algorithm.

Chapter 5 first highlights and addresses the challenges of implementing the algorithm on an actual robot operating in a real environment instead of operating in a graph, which is the setting in which most of the literature covering this problem operates in. This chapter then proposes a modular framework for integrating the algorithm and uses its policy to navigate in a realistic environment. The modularity allows for easy substitution of different algorithms for a trajectory planner, map representations, and even a different policy than the one proposed in chapter 4. Preliminary experimental results are provided to confirm the findings from chapter 4 and observations resulting in new challenges for the robotics community from the attempt to implement a smarter navigation system are discussed.

Chapter 6 summarizes the algorithms and findings from the previous two chapters and presents directions for future research on improving and extending the LRPP solution, improvements to the framework introduced in chapter 5, and how it can be used to further research on different policies for navigating uncertain environments.

Chapter 2

Literature Review

As mentioned in the introduction, a robot must have two key abilities to be able to navigate: 1) localization, and 2) planning how to reach the goal. This chapter will first review the literature on navigation without uncertainty, covering the research that has been done in localization and path planning. Next we will review navigation with uncertainty, motivating the main contributions of this thesis. Finally, we relate how this work fits with the broader reinforcement learning literature and why this particular navigation problem requires special treatment.

2.1 Navigation Without Uncertainty

In this section, we will first review the literature in path planning, before reviewing localization. Path planning is an important initial building block of forming a plan to get to the goal, but as was mentioned in the introduction, a path and a policy are different.

The primary objective of any path planner is to find a path from a start position to a goal position. Given a map of the environment where the free space and the occupied space is specified, there have been two main approaches to this problem, one can plan a path directly on this map, or discretize it before running a graph-based path planner such as A* or Dijkstra's algorithm. Rapidly-exploring Random Trees (RRT) [40] and many of its variants (RRT-connect [38], parallel RRT [12], etc) are based on the former, building a graph from a given starting point, incrementally adding vertices and edges until a complete path to the goal is found. Approaches based on first discretizing the environment can take a random approach to selecting vertices in the free space such as the probabilistic roadmap (PRM) [28], using a set of motion primitives to form a lattice [66], or discretizing the environment into a grid [11].

Initially, the main objective of path planners was to find the shortest possible path in a given environment. However, with the presence of noise in both sensor data and controls, it was also important to consider how likely the robot may collide with obstacles, giving rise to variations of path planners that take this into consideration such as the Rapidly-exploring Random Belief Trees (RBT) [10] algorithm or sampling probabilistic maps [51].

Localization is also a key requirement for navigation. In the absence of a precise positioning system like GPS (which is a common occurrence in indoor environments), alternative approaches are needed. One approach is to add specialized markers like April tags [61] or ARTag [16], known as fiducial markers to the environment, which can inform the robot of its current position.

However, it is desirable to be able to localize without needing to modify the environment prior to robot deployment. One of the most prevalent algorithms for localization is the Augmented Monte Carlo Localization (AMCL) [13] algorithm which uses a particle filter to determine the most likely position the robot is in given a map. This approach and many other approaches assumes the robot is equipped with LiDAR or sonar to sense the distance it is from obstacles. Visual Teach and Repeat (VT&R) [18] uses stereo camera images to localize and follow a taught path. Other SLAM algorithms uses landmarks to recognize previously visited areas (a problem known as loop closure), like ORB-SLAM [56] and FASTSLAM [54].

The main point is that both path planning and localization are heavily researched areas, with many algorithms and their results available. Therefore, the remainder of this thesis will assume that the robots considered are reasonably adept at localization and path planning. It is important to note that all of the aforementioned path planning algorithms assume that the environment does not change.

2.2 Navigation with Uncertainty

In the presence of uncertainties, the optimistic policy has been the most prevalent choice because of its simplicity to implement. There have been many algorithms developed to improve computational efficiency that are based on the path planning algorithms in Section 2.1 and essentially replanning when the original path is blocked. Examples of this include D*Lite [30], LPA* [32], etc.

For uncertainties that result in small discrepancies in the original path, this is a reasonable approach. Imagine moving around a chair or box that was not in the original map. However, the problem becomes more apparent when more effort is expended because of an unexpected obstacle that is not in the map used for planning, like in figure 1.1 where a door to a hallway is closed (assuming the robot is not capable of opening the door), which may result in a long detour that could have been avoided depending on the situation.

A well known problem that can capture this scenario is the Canadian Traveller's Problem (CTP), Section 3.3 provides details on the problem definition. Informally, the CTP states that given a graph, find the optimal path for an agent to take from start to goal. However, some of the edges are blocked (the robot cannot traverse that edge), and these edges are only revealed as being blocked to the agent when it reaches an endpoint of the edge.

A common variant of this problem is the stochastic CTP, where each edge has a known probability of being blocked. For brevity, when we refer to CTP, we will mean the stochastic CTP for the remainder of this thesis. The original problem was defined by Papadimitriou et al. [62], and from it many other variations have been suggested. Bnaya et al. [8] introduced CTP with remote sensing, where for a known cost, the robot can sense the state of any edge. Lim et al. [45] introduced the Bayesian CTP where instead of treating each edge independently, a probability is assigned to different configurations of the graph, which is similar to our approach of modeling the environment. Guo and Barfoot [21] introduced the robust CTP, where the variability of the policy cost is also part of evaluating policies, reducing the worst-case cost. We show that the problem we introduce in chapter 4, the Learned Reactive Planning Problem (LRPP), can be reduced to the CTP.

Optimal policies for the CTP can be calculated using AO* based algorithms. The most notable being CAO* [1] which guarantees optimality with an impressive runtime compared to other optimal algorithms such as value iteration and the original AO*. However, they limit the number of observations the robot is allowed to make in their results. Some algorithms for the CTP, such as UCT-CTP [15], optimize a policy by sampling policies and estimating their expected cost by running the sampled policy over a series of environments based on the edge probabilities. These runs are known as *rollouts*. The more rollouts that are performed, the more accurate the cost estimate will be, and generally the better the resulting policy. As expected, their runtimes are quite high as a result of these rollouts. Other algorithms use a heuristic to approximate the next step in a policy, like the Hedged Shortest Path under Determinization (HSPD) algorithm proposed in [45] or the algorithm proposed for the Reactive Planning Problem [49]. Both assume there is structure to the environment certainties that can be exploited to find a minimum cost policy.

There has been very little work on two fronts, the first being the CTP and all of its variations (and subsequently, their solutions) assume the probability of an edge being blocked is known, but in practice this information is generally not known. Only the work by Nardi and Stachniss [57] addresses the problem of capturing this information from the environment. They employ factor graphs, which can approximate the correlation of edge traversability between two edges. In contrast, we propose a solution in chapter 4 that can capture the correlation of edge traversability between many edges.

The second shortcoming of the existing literature is that there is little, if any, work that combines these CTP algorithms with existing navigation systems to form a pipeline that can be implemented on a real robot, which is what chapter 5 attempts to address.

2.3 Reinforcement Learning

Aksakalli et al. [1] showed that the stochastic CTP can be modelled as a Markov Decision Process (MDP) and a deterministic Partially Observable Markov Decision Process (POMDP), both of which are formulations frequently used in reinforcement learning. However, generic POMDP solvers cannot be used to find optimal policies for the CTP in practical applications because the state space for the stochastic CTP is exponential.

There have been advances to utilize neural networks to approximate value functions for MDPs with large state and action spaces. This combination of neural networks and traditional reinforcement learning is coined Deep Reinforcement Learning (DRL). Mnih et al. [53] introduced the Deep Q Network (DQN) that is the first success at DRL to learn how to play Atari games from pixels. There have been many robotic applications for DRL, from improving exploration [74], navigating towards a visual target [75], to learning to read maps [9]. Kanezaki et al. [26] proposed a path planning algorithm based on DRL that performs well in the presence of many unexpected obstacles, but the behaviour is that of an optimistic algorithm. Furthermore, Lillicrap and Hunt et al. [44] introduced the deep deterministic policy gradient (DDPG), an algorithm to solve reinforcement learning problems with continuous action spaces (for example, balancing a quadroped, running, arm manipulation, etc). The drawback from using these techniques for navigation is the need for training before the algorithm can be used in practice. The difficulty with training navigation with uncertainty, and in particular the solutions for the CTP, is that unlike tasks such as learning to play a specific video game or learning how to walk, the state space can vary significantly with each new environment. For example, imagine navigating an unfamiliar university campus, knowing your own university really well does not translate into knowing how to navigate the new campus. Even knowing all other campuses on the planet may not help with navigating this particular one. This implies that if the deployment environment is different from the training data, the algorithm may perform poorly and have to be retrained. Obtaining data for training navigation with uncertainties is also a challenge. In contrast, our solution does not require any training, and can be deployed with minimal setup.

The UCT-CTP approximation algorithm introduced by Eyerich et al. in [15] is based on the Upper Confidence Tree (UCT) algorithm that has been successfully applied to many MDP and POMDP problems. It takes advantage of an easily computable upper bound on the optimal cost of a policy. But both Nardi et al. [57] and Lim et al. [45] show that the cost of the policies their algorithms provided are less than the policies returned by UCT-CTP, suggesting that specially-designed approximation algorithms may be better suited to finding policies for environments with correlated edge traversabilities. In addition, [15] reported a runtime of approximately 5 minutes for a graph with 100 vertices and 280 edges for UCT-CTP, whereas in chapter 4, we show that a policy can be computed by our approximation algorithm in a few seconds for a graph with 400 vertices and 1654 edges.

Chapter 3

Background

This chapter provides an introduction to the notation that will be used throughout this thesis and it also details existing problems that have been previously researched, but are essential to understanding the following chapters. The final section summarizes the current types of maps that a robot can use to navigate.

3.1 Basic Notation

A weighted undirected graph G is defined by the pair $G = (V, E)$ with the cost $c : E \to \mathbb{R}_{\geq 0}$ for traversing each edge $e \in E$. A path P in a graph is defined by a sequence of vertices $v_1, \ldots, v_k$ that satisfies $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}_{k-1}$ with cost of traversal defined by $c(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1})$. With some abuse of notation for $v, u \in V$, we let $c(v, u)$ denote the minimum cost of a path from v to u. An edge $e = (v, u) \in E$ is said to be *incident* with vertices v and u.

There are $m = 2^{|E|}$ edge subsets of E. Let the complete set of subgraphs of G be denoted by $\mathcal{G} = \{G_1, \ldots, G_m\}$, where $G_i = (V, E_i)$ and $E_i \subseteq E$.

This thesis frequently uses the ℓ_2 norm or the euclidean distance, commonly denoted as $||x, y||_2$ which is the euclidean distance between $x \in \mathbb{R}^2$ and $y \in \mathbb{R}^2$.

3.2 Problem Complexity

When developing algorithms, common measures of how 'good' a solution is compared to other solutions is how much time it takes to compute a solution and how much space in memory is required, known as time complexity and space complexity respectively. While giving empirical measurements of time or space can be valuable, values can fluctuate depending on the hardware the algorithm was run on, and in addition, algorithms commonly scale to how large the input to the algorithm is. For example, suppose the problem is to count the number of times the letter a appears in a given word, and the solution is to check each letter in the list starting from the beginning. A bigger word means it will take longer for the solution to finish counting all the a's.

The field of computer science has standard notation when referring to these relative complexities, the only one that we are concerned with in this work is the big O notation, or $O(.)$, which indicates the worst relative complexity by a constant factor. Using the previous example, its worst time complexity would be $O(n)$ where n is the number of letters in the word, because you have to check each letter once. If there was a solution that required every letter in the word to be checked at least twice, the worst case time complexity would still be $O(n)$ because 2 is a constant.

It is useful to classify problems by this relative complexity to get a sense of how difficult the problem is to solve. For example, if it is proven for a solution to a problem requires at most polynomial space to solve, or $O(n^x) \quad x \in \mathbb{Z}_+$, then the problem is in the class of PSPACE-complete. PSPACE-hard would indicate that at best, the solution would require polynomial space to solve the problem. There are many complexity classes for time and space, while we only provide a very brief introduction to the concept of complexity classes, there are many resources that explain the specifics of different classes and how problems are classified. This thesis will classify the formally introduced problems, but they will not be the focus or a main result.

3.3 Stochastic Canadian Traveller's Problem

The Canadian Traveller's Problem (CTP) captures the problem of executing a navigation task where the roads are known, but their conditions are unknown a priori. Similar to travelling in Canada during the winter, some roads may be blocked because of adverse weather conditions. The roads are given as an undirected weighted graph G where some of the edges are blocked, but the robot is unaware of which edges are blocked. This is a well known problem in robot navigation as it models the case where only partial information is known about the environment the robot is operating in, but it is desirable for the robot to navigate in a way to minimize its travel cost or some other cost (such as fuel).

The stochastic CTP is a variation of the original CTP where the robot is aware of the probability of each edge being blocked. Formally, an instance of the CTP can be represented as a tuple $\mathcal{I} = (G, p, c, v_s, v_g)$ where

- $G = (V, E)$ defines the connected, undirected graph,
- $p : E \to [0, 1]$ defines the blocking probability of an edge,
- $c : E \to \mathbb{R}_{\geq 0}$ defines the cost of traversing each edge, and
- $v_s, v_g \in V$ are the start and goal vertices.

A weather is a subset of E that contains all of the unblocked edges in a particular task execution. As the robot is executing the task, its belief state can be represented by three sets: the unblocked edges, blocked edges, and the unknown edges. It only knows the state of an edge if it has visited one of the endpoints. It is important to note that the state of an edge does not change during a task execution. Thus a solution would be a policy that maps the belief state to an action. The goal is to compute a good policy that minimizes the expected cost to navigate from v_s to v_g, given that the robot does not know the weather.

Computing an optimal policy is difficult, and this problem was shown to be PSPACE-hard in [17]. Typically, heuristics are used to calculate the policy. The simplest approach is the optimistic algorithm, which assumes all of the edges are unblocked, unless proven otherwise while traversing its given environment. More sophisticated solutions yield policies with smaller expected costs using sampling methods like UCT-CTP [15], while in more recent work, approximations [45] and graph search methods [1] were employed.

3.4 Reactive Planning Problem (RPP)

Suppose a robot must repetitively execute the same start-to-goal task (for example, moving items from a truck to a warehouse), and suppose it is aware of the different configurations the environment could be in and their probabilities. The configuration could change with each task execution. How should the robot navigate to minimize the average cost to execute this task? This problem was formally posed by MacDonald [49] as the Reactive Planning Problem (RPP). A solution to the RPP allows the robot to go to edges to observe whether they are blocked or unblocked to draw conclusions about the current environment configuration. This problem assumes the robot is much more likely to operate in certain environment configurations than others, thus its observations may inform the robot which environment it is in.

Definition 3.4.1 (Reactive Planning Problem (RPP)) *Given a graph $G = (V, E)$ and a subset of the subgraph edges in $\mathcal{G}$ represented by $S = \{E_i, E_j, \ldots\}$, start and goal vertices v_s, $v_g \in V$, and a corresponding random variable X that has a known pmf over S and observations $\Theta_v \quad \forall v \in V$, find a complete policy π that minimizes $\mathbb{E}_X(\pi)$ for traveling from v_s to v_g in the subgraph induced by random draw x from X.*

A complete policy π guarantees that the robot will always arrive at the goal if a path in x exists, otherwise it will correctly assess that such a path does not exist. The expected value of a policy can be evaluated as

$$\mathbb{E}_X(\pi) = \sum_{x \in \mathbb{N}_{|S|}} cost(\pi | X = x) p(X = x) \tag{3.1}$$

The policy is represented as a decision tree (figure 3.1) and each node is a tuple (Y, v, O) where

- $v \in V$ is the vertex the robot is currently at,
- $O \in \Theta_v$ is the observation to be taken at vertex v, and
- $Y \subseteq \mathbb{N}_{|S|}$ is the belief of the robot prior to taking observation O.

The leaf and root nodes do not have observations, therefore, at those nodes, $O = \emptyset$. The leaf nodes are terminal states where the robot has either arrived at the goal (g) or concluded that there is no path to goal (ng) in the current subgraph. Legs are the edges

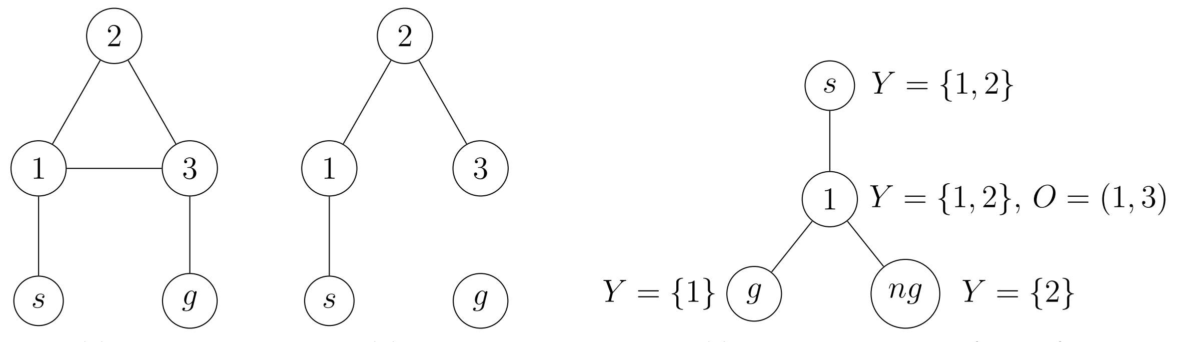


Figure 3.1: Example of environment and corresponding policy

between the nodes of the policy, and they represent the path the robot will take to travel from one node to another. The leg that the robot takes depends on the outcome of the observation taken at the parent node.

Building the policy requires balancing the act of taking observations (known as exploration) and moving towards the goal (known as exploitation). First, observations are classified as constructive or not by looking at their possible outcomes. If they have more than one outcome, then they are *constructive* and can be considered as potential observations for the policy in equations 3.2 and 3.4.

For example, consider the environment $\{G_1, G_2\}$ in figure 3.1. Edge $(1, 2)$ would not be considered a constructive observation as it is unblocked in both G_1 and G_2, but edge $(1, 3)$ is constructive because it is unblocked in G_1 and blocked in G_2. The robot would be able to differentiate its current environment between G_1 and G_2 by observing $(1, 3)$.

$$c_{\bar{G}}(v, v_g) \leq c_{\bar{G}}(v, u) + \mu(O) + \mathbb{C}_Y(u, v_g) \tag{3.2}$$

Equation 3.2 balances the act of exploration and exploitation. The known graph, $\bar{G}$, is what the robot knows about its current environment. Going back to the example, if the robot is at s and has not moved yet, then it knows edges $(s, 1), (1, 2)$ and $(2, 3)$ are unblocked, but it would be unsure of $(1, 3)$ and $(3, g)$. Then $c_{\bar{G}}(v, v_g)$ is the cost of traveling from vertex v to v_g in the known graph; if $c_{\bar{G}}(v, v_g) = \inf$, this implies there is no certain path that the robot can take to go to v_g. The right side of equation 3.2 is the sum of the cost to travel from v to the observation location u, the cost of taking observation O, and the expected cost of traveling from u to v_g, given the robot's current belief (calculated using equation 3.3). To summarize, the robot will only consider taking an observation if the overall cost of the observation is less than the known cost to go to the goal. If all of the constructive observations that can be taken from the current vertex v with the belief Y satisfy equation 3.2, then the robot should go straight to v_g from v.

$$\mathbb{C}_Y(u, v_g) = \sum_{i \in Y} p(X_Y = i) c_{G_i}(u, v_g) \tag{3.3}$$

Otherwise, we select the observation that satisfies equation 3.4 to be the next observation. The set of constructive observations that do not satisfy equation 3.2 are denoted by

R_v. To score each observation in R_v, we use the same overall cost as in equation 3.2 and multiply it by $\mathbb{E}[H(X_Y|O)]$, the expected entropy of taking the observation O. Simply put, the entropy is a numerical measurement for how uniform a pmf is. If all the subgraphs are equally likely, then the entropy is high, but if one subgraph is more likely than all the others, then the entropy is low. For a more detailed explanation of the entropy term, the reader can refer to [49].

$$O_{min} = \underset{(O,u)\in R_v}{\operatorname{argmin}} \Big[(c_{\bar{G}}(v,u) + \mu(O) + \mathbb{C}_Y(u,v_g))\mathbb{E}[H(X_Y|O)]\Big] \tag{3.4}$$

A dynamic programming approach is used to implement the heuristics and to determine the observations that should be taken at every level of the decision tree. Algorithm 1 outlines the procedure explained above.

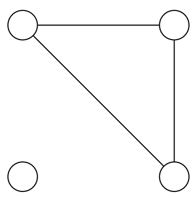
Algorithm 1: RPP Policy Builder

```
   Input: Graph $G$, edge subsets $S$, probabilities $p$, vertices $v_s$ & $v_g$
   Output: policy $\pi$
1  Compute $c_{G_i}(v, v_g)$ for all $v \in V$ and $i \in \mathbb{N}_{|S|}$;
2  Let $Q$ contain only $(\mathbb{N}_{|S|}, s)$;
3  while $Q$ not empty do
4      Remove $(Y, v)$ from $Q$;
5      if $c_{G_i}(v, g) = \inf \forall i \in Y$ then
6          Mark $\pi$ at $v$ for $Y$, no goal terminal state;
7      else
8          Compute $(R_v, D_v) = PossibleObservations(G, S, (Y, v))$;
9          Remove elements of $R_v$ that satisfy (3.2);
10         if $|R_v| = 0$ then
11             Add leg from $v$ to $v_g$, mark $\pi$ as goal terminal state;
12         else
13             Let $(O, u) \in R_v$ be the minimum of (3.4);
14             Add leg from $v$ to $u$ and node $(Y, O)$ to $\pi$;
15             Add $(Y_{new}, u)$ to $Q$ for each outcome of $O$;
16     Return $\pi$
```

3.5 Vertex Clique Cover Problem

Given an undirected graph, a clique is a subset of vertices where every vertex has an edge to every other vertex in the subset.

The Vertex Clique Cover (VCC) Problem is defined as finding the minimum k cliques to cover every vertex in an undirected graph. It has been shown that the VCC problem is equivalent to the graph coloring problem [27], which is defined as given an undirected graph, find the minimum k colours such that no two adjacent vertices share the same colour.


(a) VCC

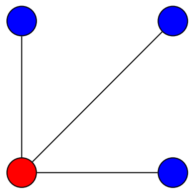
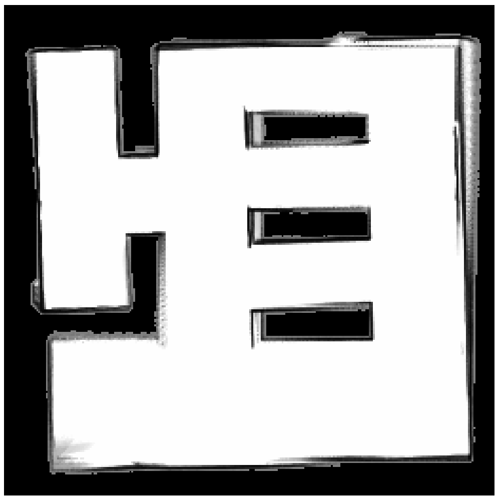

(b) Graph coloring

Figure 3.2: Example of VCC to Graph Coloring

3.6 Map Representations

Occupancy grids are a very common type of map that discretizes the environment into a grid, and each cell is a square. The most basic variant is where each cell is marked as either occupied, free, or unknown. The resolution of an occupancy grid indicates how much area each cell covers. For example, if an occupancy grid has a resolution of 5cm, then each cell has a side length of 5cm, covering an area of 25cm^2. In probabilistic occupancy grids, each cell is given a probability of being blocked or is marked as unknown. It is important to note that a probability of 0.5 is not equivalent to unknown; unknown indicates that no information is available about the occupancy of the cell. Depending on the application, path planners may or may not be permitted to plan a path through unknown areas. Also, the probabilistic occupancy grid may be thresholded into the trinary values of occupied, free, or unknown by thresholding the probability. In this case, unknown cells will remain unknown rather than being set as occupied or free.


(a)

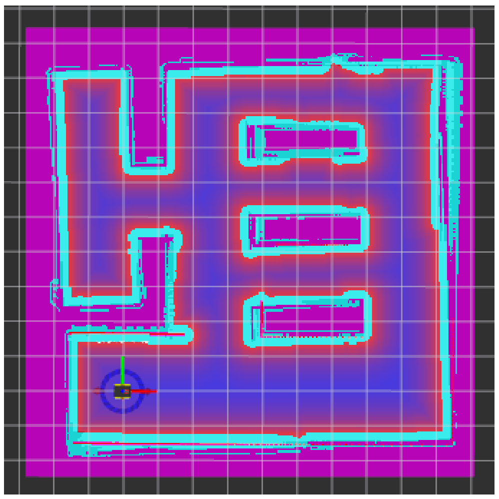

(b)

Figure 3.3: (a) Example of a probabilistic occupancy grid, white indicates a free cell while the darker cells indicate a higher probability of the cell being occupied. The pure black cells indicate unknown space. (b) Example of a cost map of the same environment as (a). Lethal areas are marked with cyan, while the free space has a gradient from red to blue, with red areas having a higher cost. The magenta areas indicate unknown space.

A more advanced occupancy grid would be a costmap, where instead of a probability, a cost is assigned to each cell. If a cost exceeds an assigned lethal value, the cell would be considered occupied and a path will not be planned through it. Obviously, occupied cells will be assigned a lethal value. Free cells are assigned varying values of cost that could depend on multiple factors. For example, the cost may increase the closer a cell is to an occupied cell, since there is a higher likelihood of the robot colliding with an obstacle if a path is planned through that cell. The varying costs can be used to encourage planners to plan paths that are not too close to obstacles. Again, unknown cells are differentiated from occupied and free cells.

These grid-based maps are also known as metric maps because precise paths can be planned on these maps. Alternatively, topological maps can also be used for planning. Pure topological maps are usually undirected graphs with vertices representing points of interest, such as in figure 3.4.

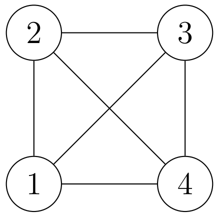


Figure 3.4: Topological map example. Only transition information between vertices and their relative positions are known, the exact locations of vertices are not

In the latter half of this thesis, we use a hybrid map that fuses a grid with a graph, also known as a topological-metric or topometric map. These maps allow for quickly calculating an approximate path on the environment as a whole, and only calculating a precise path on the occupancy grid from vertex to vertex. This approach has been shown to perform faster on large environments [7, 34, 70] while returning close to optimal paths. An example of this kind of map is shown in figure 3.5.

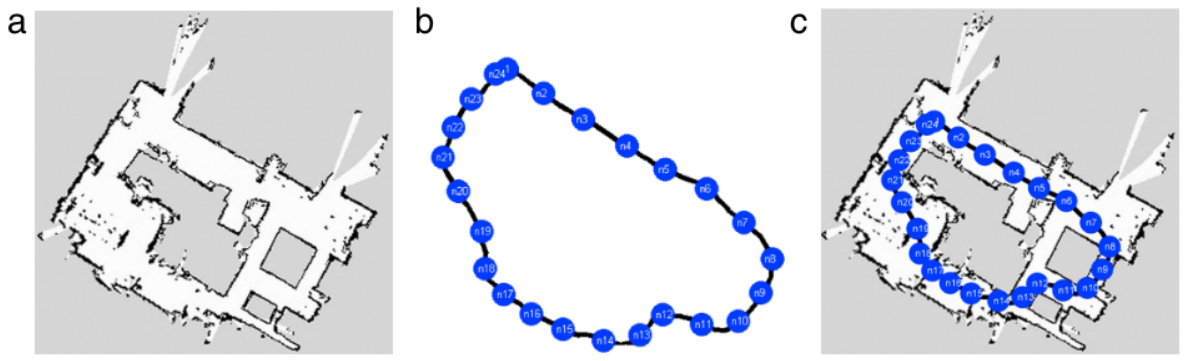


Figure 3.5: (a) An example of a 2D metric map of an explored indoors environment, (b) the respective topological graph and (c) the hybrid topometric map, where each node in the topological graph is associated with a specific region of the occupancy grid. This figure was taken from [35].

Chapter 4

Learning Motion Planning Policies in Uncertain Environments through Repeated Task Executions

4.1 Introduction

As mentioned in chapter 1, environmental uncertainty makes it difficult to harness a single map to complete repeating point-to-point or navigation tasks; either resulting in the robot re-mapping during each task execution, or using only the most recent map along with heuristics to navigate around unexpected obstacles. In [49], the authors proposed an alternative solution to this problem, calling it the Reactive Planning Problem (RPP). The idea was to generate a motion policy, that balanced the competing tasks of identifying which environment configuration (or map) the robot was operating in, and efficiently navigating to the goal. However, their solution required the robot to be given the full set of possible configurations of the environment, and their relative likelihoods *a priori*. In practice this information is difficult to obtain and likely to be inaccurate. As a result, their proposed approach lacked robustness in that it cannot adapt to new or unexpected environments. This chapter builds on the RPP and proposes a new solution in which only one initial map is required *a priori*, and instead the robot learns about the environment uncertainties and iteratively builds a motion policy through repeated executions of a navigation task.

The typical approach to navigation is to encode the environment into a map before running a planning algorithm to generate trajectories. To this end, there is a plethora of mapping algorithms for different applications, ranging from complex 3D surroundings [69] to highly dynamic environments [52]. Given a map along with robot dynamics, the task of selecting desirable robot actions *prior* to task execution can be a computationally complex task [39]. To address this, [37] considers obstacle correlations only between neighboring regions dependent on the direction from which the robot enters. The computational burden is further reduced by allowing the robot to re-plan during execution as its map changes. Algorithms like D* Lite [31] and lifelong planning A* [32] provide fast re-planning in order to approach real-time reaction to obstacles. In this work, the mapping objective is to capture only regions of the environment critical to task completion. We discuss conditions to encourage the robot to map only regions that may benefit future tasks.

The topic of reinforcement learning in robotics, reviewed in [29], presents a method to iteratively improve performance of difficult tasks. Q-Learning has been used to solve similar reinforcement learning problems [33, 63]. More recent work has combined these concepts with deep learning called deep reinforcement learning [9, 41, 75]. For example, [74] uses a deep reinforcement learning strategy to improve the exploration of office buildings. In contrast, our work does not require extensive training data.

Our work is focused on minimizing the total cost for a given number of repetitions of a task (or episodes). For this problem, there is an explicit reward/cost for an action in the current task, but there is also an implicit reward/cost for an action in the current task that will influence future tasks. This is further complicated as the interaction between the current and future tasks may become less important as the robot approaches the final task. Several works discuss the inverse reinforcement learning problem (IRL), which builds a model of the reward function [25, 36, 58]. Much of this work requires expert examples to learn the underlying reward function [3]. For our work, this is unavailable to the robot. Instead, we focus on predicting the implicit reward of an action on future tasks.

4.1.1 Chapter Contributions

The contributions of this chapter are the following:

- we introduce the Learned Reactive Planning Problem (LRPP),
- we present an algorithm that condenses past experiences into what we call *super maps* in order to generate and update a motion policy between tasks, and
- we present simulation results showing the strengths and weaknesses of using this approach.

4.2 Problem Setup

Consider a single robot that must repeatedly navigate from a start to a goal location in a partially known static environment. It must perform this task $T \geq 1$ times, and obstacles may be added or removed from the environment in between tasks. Our goal is to minimize the total cost to complete these T tasks. In the following subsections we define the environment, robot model, and its motion policy before formalizing the problem.

4.2.1 Environment Model

The robot functions within a graph drawn from the full set of subgraphs of G labelled $\mathcal{G} = \{G_1, \ldots, G_m\}$, with a probability mass function (pmf) capturing the likelihood a given graph will be drawn. Contrary to the RPP [49], the robot does not know the pmf over $\mathcal{G}$. The robot experiences a sequence of T random graphs $G_{X_1}, \ldots, G_{X_T}$, where $X_1, \ldots, X_T$ are independent and identically distributed (i.i.d) random variables according to the pmf over $\mathbb{N}_r$ (i.e., $\mathbb{P}(X_t = i) = p_i$ for $i \in \mathbb{N}_m$ and $t \in \mathbb{N}_T$ where $p_1, \ldots, p_m$ is the pmf). We drop the index when referring to the underlying pmf and use random variable X. The robot executes task t in the realization G_{x_t} of G_{X_t} without knowing G_{x_t}.

We are interested in applications where a small (cardinality much less than r) subset of $\mathcal{G}$ dominates the pmf. Thus graphs in this subset are much more likely to be drawn. For cases where each graph is equally likely, namely $p_i = \frac{1}{r}$ for any $i \in \mathbb{N}_m$, our approach will operate similarly to an optimistic policy. From a practical point of view, we are interested in structured environments (even though that structure is unknown at first), and for which that structure has occasional unexpected modifications. This captures environments where certain areas are often blocked or unblocked (e.g., a doorway) but others are expected to be in a given state (e.g., it is unlikely a wall will suddenly be absent). Our work still reacts to the unexpected case, but we wish to speed up reaction time for the most probable cases.

4.2.2 Robot Model

Suppose for some task t the robot functions within the realization $G_{x_t} = (V, E_{x_t})$. If the robot occupies $v \in V$, it may sense an edge $(v, u) \in E$ to check if it is blocked and thus not traversable. Formally, the sensing action is defined by the mapping $\gamma_v : I_v \to \{\text{blocked}, \text{unblocked}\}$ where $\gamma_v(e) = \text{unblocked}$ for $e \in E_{x_t}$ and $\gamma_v(e) = \text{blocked}$ otherwise, this is the edge's state. If the robot, positioned at $v \in V$, wishes to traverse $e = (v, u) \in E$, it first senses the edge e. If $\gamma_v(e) = \text{unblocked}$, then the robot will proceed to traverse e and arrive at u, incurring the transition cost $c(e)$. For simplicity, we assume there is no cost to sense the state of e and that the robot is capable of sensing whether e is blocked or not. Although we assume no sensing cost, it can be added without significant changes to the problem or solution.

After the robot performs n actions within the environment, let $E_{t,n} \subseteq E$ denote the set of edges for which the robot knows the state. We define the robot's understanding, or *map*, of G_{x_t}, after its nth action, as the tuple $M_{t,n} = (E^{\text{b}}_{t,n}, E^{\text{u}}_{t,n})$ for known blocked edges $E^{\text{b}}_{t,n} = \{e \in E_{t,n} | e \notin E_{x_t}\}$ and known unblocked $E^{\text{u}}_{t,n} = \{e \in E_{t,n} | e \in E_{x_t}\}$. Note that $E^{\text{b}}_{t,n}$ and $E^{\text{u}}_{t,n}$ form a partition of $E_{t,n}$. When the task is finished, the robot stores the *map* in the list $\mathcal{M}_t = [M_1, \ldots, M_t]$ for $t \in \mathbb{N}_T$, where n is removed to indicate the task is completed.

4.2.3 Complete Policy

Consider a single task $t \in \mathbb{N}_T$, and to reduce notational complexity in what follows, we drop the index t. The robot state space is defined as $V \times 2^E \times 2^E$ where $v \in V$ is the robot's position, $E^{\text{b}} \in 2^E$ is the set of known blocked edges and $E^{\text{u}} \in 2^E$ is the set of known unblocked edges. At a given state $(v, E^{\text{b}}, E^{\text{u}})$, the robot selects an action defined by an outgoing edge $e \in I_v$, and a command from the set $\mathcal{C}$. The most primitive commands being $\{\text{move, terminate}\}$, but in later sections we add an observe and switch policy command. Formally, a policy maps the robot state space to the set of actions, $\pi : V \times 2^E \times 2^E \to I_V \times \mathcal{C}$. The move command updates E^{u} if $e \in E_x$ and modifies v since the robot has moved or it only updates the set of blocked edges E^{b} if $e \notin E_x$. The terminate command ends task execution and should only be used when the robot is in a *terminal state*. Given a start and a goal $v_s, v_g \in V$, a state $(v, E^{\text{b}}, E^{\text{u}})$ is said to be *terminal* if $v = v_g$ or the optimistic graph $G_o = (V, E \setminus E^{\text{b}})$ has no path from v_s to v_g. We now define a complete policy.

Definition 4.2.1 (Complete Policy) *A policy π is complete for a graph G if it produces a finite sequence of actions that ends in a* terminal state *for any subgraph in the set $\mathcal{G} = \{G_1, \ldots, G_r\}$.*

Given a graph G_j with $j \in \mathbb{N}_r$, consider the sequence of actions $A_{G_j} = a_1, \ldots, a_z$ produced by π for some $z \in \mathbb{N}$. Each action a has a cost, which is the sum of all edges travelled during the action; we denote this set of edges as E_a. The total cost of A_{G_j} would be given by $\text{cost}(A_{G_j}) = \sum_{i=1}^{z} \sum_{e \in E_a} c(e)$. Therefore, the expected cost to complete a task is

$$\mathbb{E}_X[\text{cost}(\pi)] = \sum_{j \in \mathbb{N}_m} p(X = j)\text{cost}(A_{G_j}) \,. \tag{4.1}$$

For a complete policy to exist it is sufficient that the component containing v_s is strongly connected for each graph in $\mathcal{G}$ that has non-zero probability. This holds as long as the robot can exit each region that it can enter.

4.2.4 Learned Reactive Planning Problem (LRPP)

We consider a sequence of T tasks where the robot wishes to minimize the summed cost of completing each task. When considering a sequence of T tasks, note that all prior tasks affect the way in which the robot completes the current task. Therefore, the policy for task t may use the information collected during all prior tasks. Formally, we define this as the Learned Reactive Planning Problem.

Problem 1 (Learned Reactive Planning Problem (LRPP)) *Given a graph G with unknown pmf over all subgraphs $\mathcal{G}$, a start and goal $v_s, v_g \in V$ and number of tasks T, find a sequence of T complete policies, $\pi_1, \ldots, \pi_T$, that minimizes $\sum_{t=1}^{T} \mathbb{E}_{X_t}(cost(\pi_t))$, where π_t may depend on the observations made in tasks $1, \ldots, t-1$.*

We now characterize the complexity of this problem for the special case when the pmf over subgraphs is completely known, which occurs as $T \to \infty$.

Proposition 1 *Even if the pmf over subgraphs $\mathcal{G}$ is known, the Learned Reactive Planning Problem is PSPACE-hard.*

Proof: Consider an instance of the stochastic Canadian Travelers problem (CTP). This consists of a graph $G_{\text{CTP}} = (V, E)$, a cost on each edge $c_{\text{CTP}} : E \to \mathbb{R}_{>0}$, and a probability for each edge $p : E \to [0, 1]$, giving the probability $p(e)$ that the edge $e \in E$ is unblocked. The goal is to find a policy that minimizes the expected cost from start to goal. To reduce this problem to LRPP with a known pmf, we create the following instance of the LRPP: We set $G = G_{CTP}$, $c = c_{\text{CTP}}$, and for each subgraph $G_i = (V, E_i) \in \mathcal{G}$, we define its probability $\mathbb{P}(X_t = i) = p_i$ as

$$p_i = \prod_{e \in E_i} p(e) \prod_{e \in E \setminus E_i} \big(1 - p(e)\big).$$

An optimal policy for this instance of LRPP then minimizes the expected cost of completing a task. This policy then is also optimal for the CTP. Since the CTP is PSPACE-hard [17], the LRPP with a known pmf is also PSPACE-hard. □

Remark 1 *Notice that in the first task $t = 1$, the pmf is completely unknown, and thus minimizing the expected cost with an unknown distribution is equivalent to minimizing the worst-case cost. Thus, the first task is an instance of the non-stochastic version of the Canadian Travelers problem [62]. In this problem, the goal is to compute a policy that minimizes the competitive ratio, defined as the worst-case ratio over all subgraphs between the cost to navigate from start to goal using the policy, and the cost of the optimal path from start to goal in the subgraph. This problem is also known to be PSPACE-complete [62].*

4.3 Solution Approach

There are three key challenges to address when considering the approach in [49] to solve the LRPP. First, the subgraph set $\mathcal{G}$ and its pmf is unavailable to the robot for planning. We propose a Map Memory Filter in Section 4.3.1 to estimate $\mathcal{G}$ and its pmf by efficiently storing the robot's map $M_{t,n}$ for every task t into $\mathcal{M}_t$. Second, we need a method for the robot to reach the goal when it encounters an environment it has not experienced before. In Section 4.3.2 we introduce the idea of switching to an optimistic policy to handle such environments, and adding this command to the set C that can be used in a policy. Third, the robot needs to be able to update its navigation strategy from v_s to v_g as its estimate of $\mathcal{G}$ and its pmf changes between task executions. In Section 4.3.5 we explain how the policy generating algorithm proposed by [49] can be utilized to generate and update a complete policy π that can react to all the realizations it has experienced before. This approach also introduces the command *observe* to set C.

4.3.1 Map Memory Filter

After the nth action during task t, the robot's knowledge is defined by the tuple $(\mathcal{R}_{t,n}, \mathcal{M}_{t-1})$ where $\mathcal{R}_{t,n} = (v_{t,n}, E^{\mathrm{b}}_{t,n}, E^{\mathrm{u}}_{t,n})$ is the robot state after the nth action. The robot would only need to store map $M_{t,n} = (E^{\mathrm{b}}_{t,n}, E^{\mathrm{u}}_{t,n})$ if it did not *agree* with a map stored from a previous task. This is known as map agreement.

Definition 4.3.1 (Map Agreement) *Given maps M_1 and M_2, we say M_2 agrees with M_1 if $E^b_2 \cap E^u_1 = \emptyset$ and $E^u_2 \cap E^b_1 = \emptyset$.*

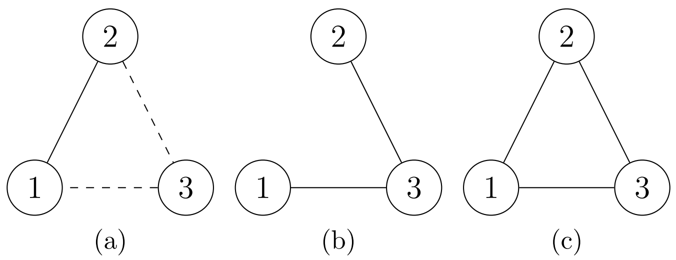


Figure 4.1: Example of map agreement

Consider the graph examples in Fig. 4.1 where the solid lines are unblocked edges, dashed lines are unknown edges, and the lack of a line indicates a blocked edge. Map (a) agrees with map (c), but not map (b) because edge (1,2) is missing. Note that the robot does not need to know all of the environment to accomplish its task and can leave regions unmapped, which may result in different realizations that seem identical to the robot.

Thus, the robot only needs to keep track of one map which we call a *super map*, formally defined below.

Definition 4.3.2 (Super Maps) *A map M_j with $j \in \mathbb{N}_t$ is a super map if all M_i for $j \neq i \in \mathbb{N}_t$ that agree with M_j satisfy $E_i^b \subseteq E_j^b$ and $E_i^u \subseteq E_j^u$.*

Then, to reduce storage and search space, we redefine $\mathcal{M}_t$ as the set of *super maps* at the end of task t. The problem of computing a minimal set of supermaps can be formalized as follows.

Problem 2 (Map Merging Problem) *Given a set of collected maps from each task, $\mathcal{M}_T = [M_1, M_2, \ldots, M_T]$, find a minimum partition of $\mathcal{M}_T$ such that every map in each subset agree with each other.*

Note that merging the maps in a subset forms a super map, and thus the solution to the Map Merging Problem provides a compressed representation of the robot's past experiences. We can show that the Map Merging Problem is NP-Hard through a reduction from the vertex clique cover (VCC) problem to the Map Merging Problem. Consider an instance of the VCC problem: Given a graph G, find the minimum number of cliques to cover every vertex. Given an instance of map merging, let each map M_t be a vertex, and let there be an edge between every two maps that agree with each other. This graph is called an *agreement* graph. Set G to be the agreement graph, and then the VCC of G provides a minimal partition of the maps in $\mathcal{M}_T$.

The agreement graph is built over time as the robot completes each task, so map merging is a form of online VCC. The VCC of a graph is equivalent to the minimum graph coloring of the complement of the graph [20]. The map merging method proposed in Algorithm 2 is a greedy approach that immediately adds a map to a subset via merging (lines 3 and 4) if it agrees with an existing super map. It is analogous to the First Fit approach to the online graph coloring problem, which, while not an approximation algorithm [72], provides good performance in practice.

Using this method of storage, we can simplify the expected cost estimate (4.1) to,

$$\mathbb{E}_X[\text{cost}(\pi_t)] = \sum_{M_j \in \mathcal{M}_t} \left(\frac{n_j}{t}\right) \text{cost}_{\pi_t}(M_j) \, , \tag{4.2}$$

where n_j is the number of maps the robot has experienced by task t that agree with super map M_j and $\text{cost}_{\pi_t}(M_j)$ is the cost of executing policy π_t in super map M_j. Thus the estimated probability of encountering M_j in the next task is $\hat{p}_{M_j} = n_j/t$. Let $\hat{P} = [\hat{p}_{M_0}, \hat{p}_{M_1}, \ldots, \hat{p}_{M_t}]$ where $M_j \in \mathcal{M}_t$, forming our estimate of the pmf of G. Note that $\mathcal{M}_0 = \{(\emptyset, E)\}$ and initialize $n_0 = 1$, leading to the robot's initial assumption of $\hat{p}_e = 1 \quad \forall e \in E$, i.e., all edges in G are unblocked.

Algorithm 2: mapFilter

```
Input: $M_t$, $\mathcal{M}_{t-1}$
Output: $\mathcal{M}_t$
1 for each $(E^{\text{b}}_j, E^{\text{u}}_j) \in \mathcal{M}_{t-1}$ do
2     if $E^{\text{u}}_t \subseteq E^{\text{u}}_j$ AND $E^{\text{b}}_t \subseteq E^{\text{b}}_j$ then
3         return $\mathcal{M}_{t-1}$;
4     if $E^{\text{u}}_t \cap E^{\text{b}}_j = \emptyset$ AND $E^{\text{b}}_t \cap E^{\text{u}}_j = \emptyset$ then
5         Update $M_j = (E^{\text{u}}_j \cup E^{\text{u}}_t, E^{\text{b}}_j \cup E^{\text{b}}_t)$;
6         return $\mathcal{M}_{t-1}$;
7 return $\mathcal{M}_{t-1} \cup M_t$;
```

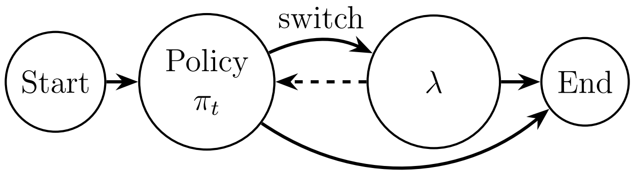


Figure 4.2: Integrating switching to the optimistic policy λ with our overall approach.

4.3.2 Switching to the Optimistic Policy

Consider the composite approach displayed in Fig. 4.2 for executing task t. The number of possible realizations for G_{X_t} may be exponential in the number of edges; therefore, our approach is to plan paths for only a subset of environments and use the optimistic policy λ for the remaining, adding a *switch* command to C.

Definition 4.3.3 (Optimistic Policy λ) *An optimistic policy λ computes online a sequence of move commands to lead the robot to a terminal state. Such a policy must guarantee that it can find a path from v_s to v_g if one exists.*

The optimistic policy λ allows the robot to handle unexpected environments as they are encountered. Meaning the robot will always enter a terminal state (in some finite number of moves) after it switches policies.

For a given task t, the robot starts by following the preplanned paths in the policy π_t until either 1) an obstacle prevents the robot from continuing (in which case the robot is in a new map) or 2) all super maps that are consistent with the robots observations have no path to the goal v_g. In either case, the robot switches to λ to finish the task. This satisfies the complete policy requirement, and the policy is updated each time a new task is completed. The preplanned paths are expected to be more efficient at reaching v_g than λ, and as such we wish to minimize the probability of the robot switching to λ.

Remark 2 *The dashed edge in Fig. 4.2 is not considered within this work as returning from λ may result in a large number of states that the policy must map to actions.*

4.3.3 Policy Structure

A policy for task t can be efficiently encoded into a binary tree $\pi = (N, L)$. The nodes N of the tree are given by tuples (Y, v, e) for belief $Y = \{i \in \mathcal{M}_{t-1} | M_i \text{ agrees with } M_{t,n}\}$ at vertex $v \in V$. The edge e is an *observation* at vertex v. For each node, L contains a path from the parent node to the current node. There are two possible outcomes for each observation, one corresponding to $e \in E_{x_t}$ and the other $e \notin E_{x_t}$. If $e = \emptyset$, then either $v = v_g$ or there is no path to goal in any of the agreeing super maps and the robot must switch to the optimistic policy. To match our robot model and to facilitate understanding, we will limit $e \in I_v$. Then in this work, we can now define the full command set $C = \{move, observe, switch, terminate\}$.

4.3.4 Build Policy

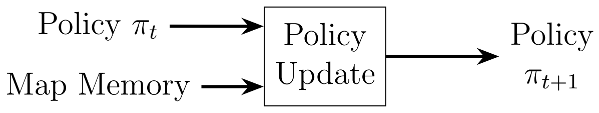
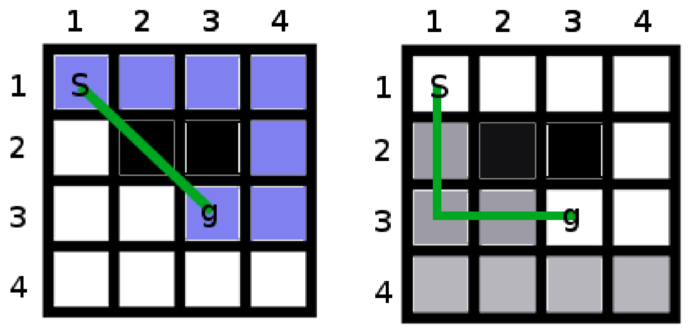


Figure 4.3: Overview of the policy update.

The key step in our approach is the update shown in Fig. 4.3 that occurs between tasks and builds a policy as more tasks are completed. After completing task t, we have our estimate of the set of subgraphs $\mathcal{M}_t$ and the pmf $\hat{P}$ based on all prior experience. We can solve the RPP problem from [49] using these estimates as the inputs (refer to algorithm 1 in Section 3.4 for details), resulting in a policy π which has the structure from Section 4.3.3. Note that the *switch* command must be explicitly added to every node that has no path to goal.

To use $\mathcal{M}_t$ as an input to algorithm 1, it first needs to be converted into a set of edge subsets, where each subset represents the passable edges in each super map. Since each super map in $\mathcal{M}_t$ may only be a partial representation of a realization, it is necessary to make some assumptions to fill in missing information. If the state of an edge in super map M_j is unknown (i.e., $e \notin E_j^{\text{b}}$ and $e \notin E_j^{\text{u}}$), we assume it to be unblocked. Formally, the edge subset for M_j will be $E_j^{\text{u}} \cup (E \setminus (E_j^{\text{u}} \cup E_j^{\text{b}}))$. This choice encourages the robot to explore, as it will attempt to traverse an unknown edge if it is beneficial.

For example, consider the scenario in Fig. 4.4. Assuming the robot only has an empty grid for M_0 in $\mathcal{M}_t$, it attempts to execute the task in (a) using the green path. However, it must switch to the optimistic policy, and at the end of the task, the robot stores the map in (a) as M_1 (b), the state of the grey squares are unknown. When building the policy, an observation for the edge ((1,1),(2,2)) will be selected, and in the case this edge is blocked, a path will be calculated from s to g in M_1 since no other maps exist. If only the partial map M_1 was available, the blue shaded path would be used in the policy. However, since we are assuming the grey squares are unblocked, the algorithm will select the green path in (b). Even if that assumption was proven wrong during task execution, it will result in more knowledge of the realization, and the next time the policy is built, the algorithm will not repeat the same path for that particular super map.



(a) (b)

Figure 4.4: (a) shows a realization for task t, and (b) is the collected M_t. The green line in (a) is the path determined by the policy π_t, in (b) by the policy π_{t+1}. The blue squares are the path that the robot actually took, the grey squares are unknown.

The edge subset only needs to be computed when the super map is initially added to the set of super maps, and when the super map is updated from a merge (line 5 in the mapFilter algorithm).

4.3.5 Policy Update

Finally, we present our entire solution in Algorithm 3, which covers task execution and policy building. In Line 1, we initialize the set of super maps $\mathcal{M}_t$ with E as a set of unblocked edges. In other words, the robot is aware of all edges that it could potentially move across. Such information could come from a floor plan of the environment, containing all permanent obstacles. The robot initially assumes that $\hat{p}_e = 1 \quad \forall e \in E$. This assumption ensures that the optimistic policy λ will always initially attempt the shortest possible path to v_g. In line 3, the policy π_t is constructed by the RPP algorithm using the set of super maps $\mathcal{M}_t$, which contains the edge subsets, and the estimated pmf $\hat{P}$. In lines 5-9, the robot executes the task by following the policy π_t constructed by the RPP algorithm until it reaches a terminal state, updating its set of super maps, along with the estimated pmf and edge subsets, in lines 11 and 12 before executing the task again.

4.4 Simulation

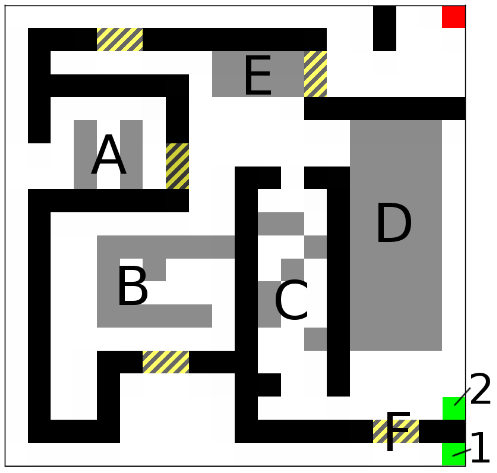
In this section, we describe the simulations we conducted with Algorithm 3 and compare the results with following only the optimistic policy. Of particular interest is the average cost of the path taken across all tasks given T. The optimistic policy used in our simulation calls A* to replan when it encounters an unexpected obstacle.

Algorithm 3: Sequential Task Completion

```
   Input: $G = (V, E)$,$v_s$,$v_g$
1  $\mathcal{M}_0 = [(E, \emptyset)]$;
2  for $t = 1, \ldots, T$ do
3      $\pi_t$ =buildRPPpolicy(G, $\mathcal{M}_{t-1}$, $\hat{P}_{t-1}$,$v_s$,$v_g$);
4      Initialize state $\mathcal{R}_{t,n} = (v_s, \emptyset, \emptyset)$ for $n = 0$;
5      do
6          Execute $\pi_t(\mathcal{R}_{t,n})$;  // if switched policies, wait until $\lambda$ terminates
7          Update $\mathcal{R}_{t,n}$;
8          Increment $n$;
9      while $\mathcal{R}_{t,n}$ not terminal;
10     $M_t = (E^{\mathrm{b}}_{t,n}, E^{\mathrm{u}}_{t,n})$ from $\mathcal{R}_{t,n}$;
11     $\mathcal{M}_t$ = mapFilter($M_t$,$\mathcal{M}_{t-1}$);
12     Update $\hat{P}_t$;
```

4.4.1 Test Environment

Our tests were conducted on the environment in Fig. 4.5. A floor plan with the black obstacles is given to the robot. The red square is the goal and the green squares are possible starting locations. The grey and striped obstacles are unknown to the robot, and the probability of them being present in a given task t is as shown in the table in Fig. 4.5. This information is hidden from the robot, but it can observe an adjacent single cell in each direction as it executes the policy. Each letter corresponds to the grey obstacles in that area, except for A which includes the closest striped obstacle to the right, and F is every striped obstacle in the environment. This map is given as a 20×20 8-direction grid, resulting in a graph with 400 vertices, 1654 edges, and 64 realizations.



Obstacle	Probability
A	0.5
B	0.4
C	0.6
D	0.8
E	0.5
F	0.2

Figure 4.5: Base map and obstacle distribution of the environment. The green and red squares are starting and ending points respectively.

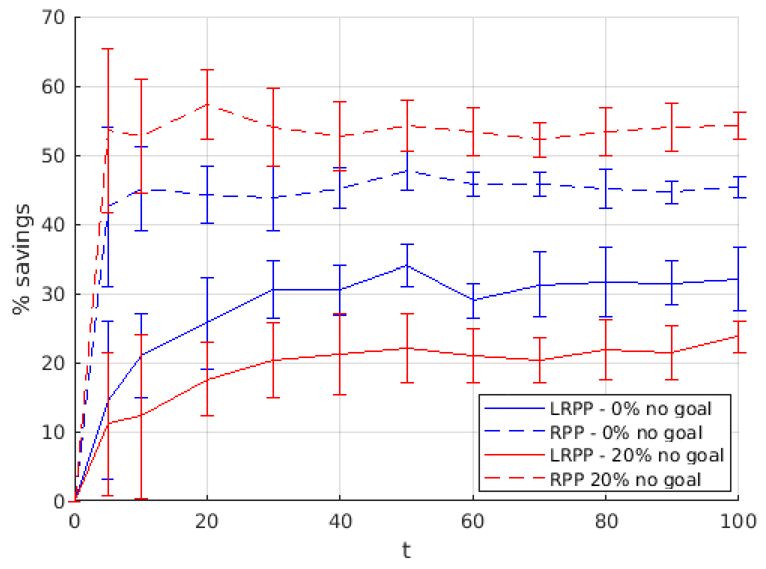


Figure 4.6: Average cost savings compared to following only the optimistic policy.

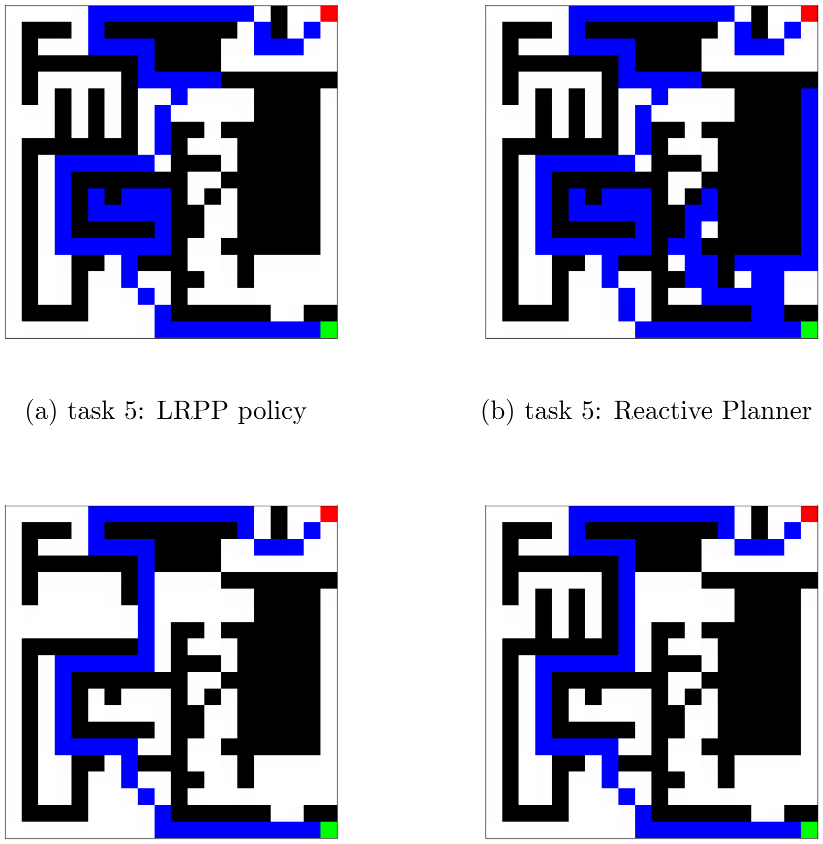
4.4.2 Results

Fig. 4.6 shows the average cost savings over t task executions using the online policy update (LRPP) and the policy generated by using the hidden environment data (RPP) compared to following only the optimistic policy. The data points are an average over 10 trials. The savings are generally greater than 20%, and none of the averages were greater than A*. Simulations for the blue lines were run on the map in Fig. 4.5, with location 1 as the start. Simulations for the red lines were run on the same map, with location 2 as the start where there is a probability of 20% for the robot to be in a realization with no possible path to the goal. The larger gap in performance between LRPP and RPP is because in LRPP, the robots switches to the optimistic policy when it thinks there is no path to goal, since it is always possible that the robot is in a new environment. On the other hand, RPP knows all possible maps, and depending on the pmf, it may be able to determine no path to goal exists without exhaustively searching the environment. Notice that in both cases, there is an overall logarithmic increase in savings as T increases. This was a surprising result for the LRPP-20% no goal trials due to how expensive the exhaustive search can be.

Fig. 4.7 shows the robot navigating the same map at task t_1 and t_2, where $t_1 < t_2$, and you can see the updated policy is able to avoid unnecessary backtracking and dead ends after just a few task executions. Fig. 4.6 shows that these savings can be quite significant.

Since the order of the realizations encountered affects the LRPP policy, it is possible for the robot to not take a shorter route if the estimated likelihood of backtracking and its cost is too high, which is a reason why the percent savings of LRPP does not converge to RPP, even in the 0% no goal case. This is a trade-off of not having a priori knowledge of the subgraphs and their pmf.

The runtime between tasks to update the policy did not change much as the number of super maps stored by the algorithm increased. In the experiments, the runtime for both 1 super map and 20 super maps was roughly 1s. For $T = 100$, the average number of super



(a) task 5: LRPP policy

(b) task 5: Reactive Planner

(c) task 8: LRPP policy

(d) task 5 and 8: RPP policy

Figure 4.7: Paths taken in the same realization at $t_1 = 5$ and $t_2 = 8$

maps was 8.3 and 16.9 for the 0% no goal and 20% no goal trials respectively, while there are 64 different realizations of the environment.

4.5 Summary

This chapter introduced the Learned Reactive Planning Problem (LRPP) as a way of modeling the problem of minimizing travel cost in uncertain environments and showed it to be PSPACE-Hard. We then proposed a two-part solution to the LRPP. First, we introduced an algorithm that filters collected past task execution experiences into super maps. Second, these super maps are exploited by the RPP solver to generate a policy that minimizes the expected cost of navigating environments the robot has already experienced. New experiences are handled by the optimistic policy. This solution is repeated for every task, which is shown by our simulation results to reduce the average cost of the task execution as the robot repeats the task.

Chapter 5

The Learn a Motion Policy (LAMP) Framework

5.1 Introduction

In this chapter, we discuss a method to implement the LRPP solver from chapter 4 on a robotic platform and the challenges that arose from this. Previously, we assumed that the robot was perfectly localized and there was no noise in the occupancy grid. In practice, localization provides an estimate of where the robot is and occupancy grids are quite noisy. We propose a hierarchical scheme where we convert the occupancy grid into regions and extract a navigation graph from the regions. The LRPP solver plans a policy on the navigation graph that guides which regions the robot should travel to, while a path planner plans the exact path for the robot in each region. Instead of observing whether an edge between two cells in the occupancy grid is blocked, we observe if a path exists between regions, which is more robust to noise in the robot's estimated pose and occupancy grid. Although there has been much work on converting occupancy grids into graphs [7, 46, 70], their navigation schemes do not consider the possibility that an edge could be blocked.

However, in this approach, we can no longer guarantee that the state of an edge can be observed from one of the endpoints, which was one of our assumptions in developing the LRPP solver. To address that, we propose a *direct inference* algorithm that uses the occupancy grid for *resolving* an edge (i.e. determining whether it is blocked or unblocked) under uncertainties. We also propose an *indirect inference* algorithm for making assumptions about an edge state based on the observations made so far. Afterwards, we introduce three different policies that utilize direct and indirect inference to determine how the robot should navigate. The first is a version of the optimistic policy that only uses direct inference, the second is a modified version of the LRPP solver introduced in the previous chapter, and the third is an online version of the LRPP solver that can react to unexpected information about the environment.

In order to navigate a real environment, a robot has multiple functions running in parallel. Often there are multiple sensors such as odometry and laser scanners that provides data streams for a mapping and/or a localization algorithm; a path planner that computes a path for the robot to follow; a path follower for generating trajectories to follow the path; and a controller that sets the motor outputs to attain the desired trajectory. This set of programs that work together, also known as a *navigation stack*, enables the robot to navigate the world around it. Later in the chapter, we demonstrate integrating our system with the navigation stack in the Robot Operating System (ROS) — an ubiquitous open-source framework for controlling robots.

5.1.1 Chapter Contributions

This chapter has the following contributions:

- a map framework that converts an occupancy grid into a hybrid map,
- direct and indirect inference algorithms for resolving edges in the hybrid map,
- an algorithm based on the LRPP solver to compute a RPP policy online,
- the LAMP framework for integrating our work into an existing navigation stack, and
- results from simulation and hardware experiments to show the functionality of the LAMP framework.

5.1.2 Chapter Organization

This chapter is organized as follows, in Section 5.2 we discuss the motivation behind a hybrid map and how to construct it from an occupancy grid. In Section 5.3, we propose the direct and indirect inference algorithms for resolving edges. Section 5.5 introduces the aforementioned algorithms for constructing the different policies. Section 5.6 shows how our system can be integrated into an existing navigation stack, and Section 5.7 presents our experiment results with the full system from simulation and on a real robot. Finally, in Section 5.9, we present a theoretical extension to our algorithm for direct inference to be more robust.

5.2 Map Framework

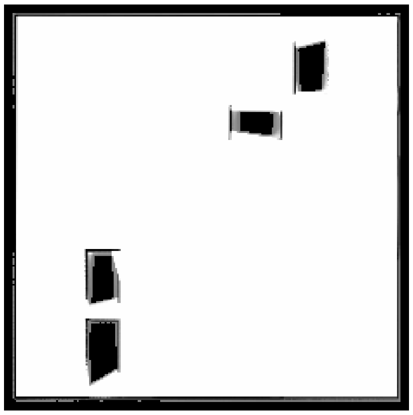
To use the LRPP solver, we require an appropriate map format to store experiences and to compare them to each other in the map filter (see Section 4.3.5). This section discusses the challenges of using practical map representations such as an occupancy grid and how those will be addressed.

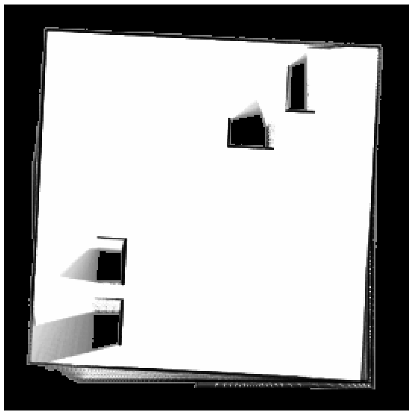
5.2.1 Map Agreement

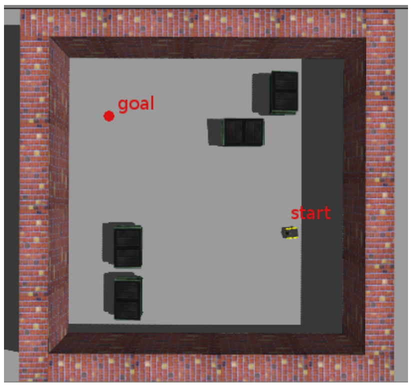
A key concept in learning the environment structure was map agreement, introduced in Section 4.3.1. The robot compares its map of the environment in its current task execution with past super maps to determine if it has previously encountered this same environment. Thus it is important to be able to decide if two maps, collected during two different task executions, agree or disagree with each other.

In the simulation results in chapter 4, the robot operated in an occupancy grid with perfect localization and observations, where each grid cell of the robot's map was either blocked, unblocked, or unknown. Two maps were in agreement with each other if none of the corresponding grid cells had conflicting states, otherwise they were in disagreement. This appears to be a smooth transition to using an occupancy grid generated by the multitudes of SLAM algorithms available, but reality is not so kind.

The issues with using a similar measure of agreement becomes apparent with figure 5.1. The robot starts at the same location (bottom right) and moves to the specified goal point using the same path during two separate executions. The robot is equipped with a single SICK LMS111 LiDAR and the occupancy grid is generated by the Cartographer algorithm [22]. During each execution a new occupancy grid would be generated. To human eyes, figures (a) and (b) would agree, but using the measure that was just described, the two maps would disagree. It is clear that there are regions that are unblocked (white) in one map but blocked (black) in the other, even if (b) were rotated and scaled to align with (a).


(a)


(b)



(c)

Figure 5.1: Occupancy grid agreement example.

One option is to coarsen the generated occupancy grid, but this could unintentionally block narrow corridors due to displacement errors. There have been multiple techniques developed by the computer vision community for measuring image similarity such as histogram comparison [67] or feature matching [4, 47] that are commonly used in robotics.

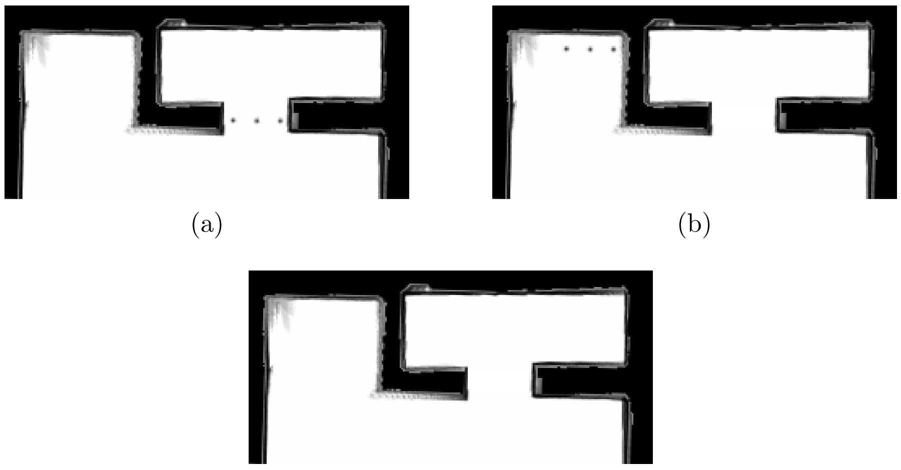

(a) (b) (c)

Figure 5.2: In (a), the small obstacles obstructs passage to the upper room, while it does not in (b). Image similarity techniques would say that both (a) and (b) either agrees or disagrees with (c). The desired outcome is for (a) to disagree with (c) and for (b) to agree with (c).

Alternatively, some work has also been done for merging multiple occupancy grids collected by a team of robots into one occupancy grid, which involves finding regions of similarity [2, 6]. Unfortunately, while these techniques may improve map agreement if we relax the thresholds, that proves detrimental when the maps actually disagree. Take figure 5.2 as an example, where there is a narrow corridor at the top that is blocked by small obstacles like cage bars. In an occupancy grid with a resolution of 5 cm, these obstacles may only take up one cell each, equivalent to one pixel in an image. And while it clearly prevents the robot from reaching the room (assuming the robot is wider than the cage bars), the above techniques may claim that the occupancy grids agree. This highlights that map agreement is not the same as image similarity. Instead of comparing if two maps look like each other, the goal of map agreement is to decide if the paths a robot can take in an environment are similar.

Since our original algorithm operated on graphs, perhaps we can use sampling to create a graph, and compare the edges. The probabilistic road map (PRM) [28] is a technique that randomly samples the free space of an environment to generate a graph. There are multiple variations [19, 51] to account for noise or difficult environment configurations like narrow gaps.

Figure 5.3 shows an attempt at creating a graph of the same environment as in figure 5.1. Halton sampling [39] was used to generate vertices, removing samples that were on or too close to obstacles. An edge between two vertices were added if their euclidean distance was less than a threshold, and if all of the cells in a rectangular swathe along the edge were free. This graph was generated on the left map, while it was superimposed on the right map. Collision checking for the edges would be re-run on the superimposed graph, removing edges if their swathe contained occupied cells in the underlying occupancy grid. The two maps are in agreement if the resulting graphs are identical.

It can be seen in Figure 5.3 that even for such a simple environment, the two graphs do not agree. Conflicting edge states are prevalent when samples are close to obstacles (top right quadrant). Of course, agreement could be reached if we decreased the number of samples or used a better sampling algorithm. However, in the event of unexpected obstacles appearing, the former would reduce the flexibility of the robot to navigate and the latter would not account for the unexpected obstacles when selecting better sampling locations.

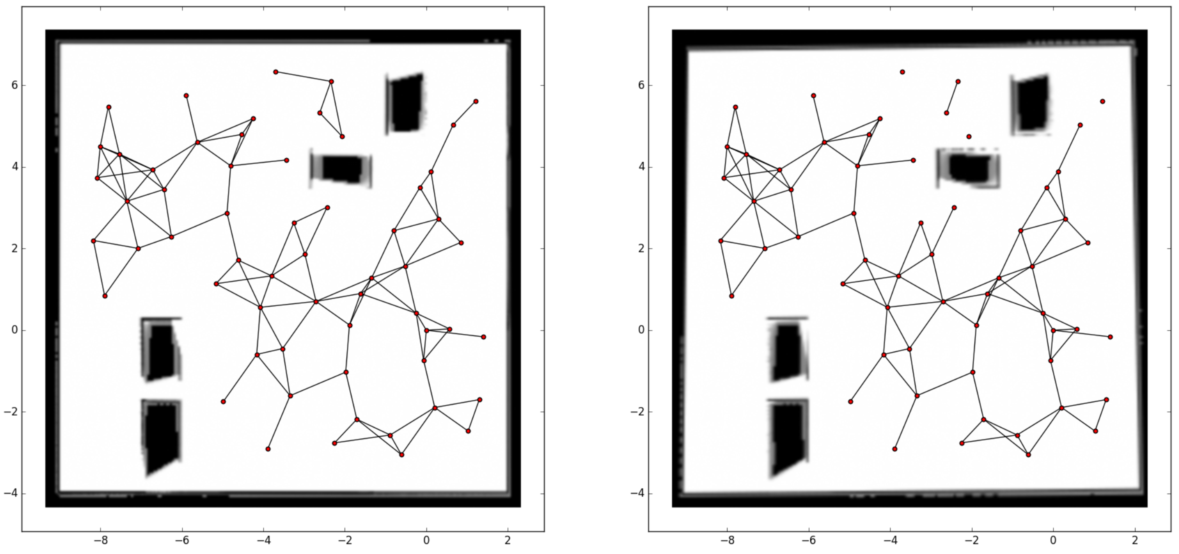


Figure 5.3: PRM agreement example

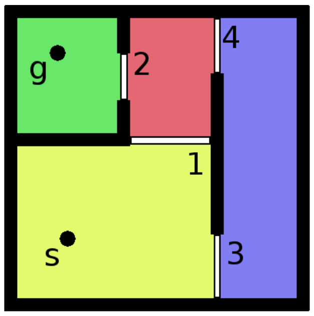
Even if the graphs in figure 5.3 did agree, what if one of the obstacles were moved one meter to the right? Should the maps be in agreement then? For maps to be in agreement, the paths the robot can take in the environment must be similar enough, but does not need to be exactly identical.

5.2.2 Navigation Graph

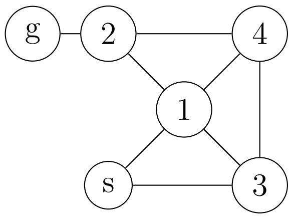
In this section, we propose our solution for map agreement using occupancy grids generated by robots. Because we are concerned with road blockages that may significantly alter the route the robot may take, we propose converting a base occupancy grid (such as a scaled floor plan where only permanent obstacles are marked) into a hybrid topological-metric map. A hybrid map combines a topological graph and an occupancy grid. The topological graph is different from a PRM generated graph because the vertices represent points of interest like rooms or hallways, while the edges may not be exact representations of the path the robot will follow. Constructing a topological graph from an occupancy grid is not a new idea, algorithms have been proposed by Thrun [70] and by Liu et al. [46]. Blöchliger et al. [7] proposed an algorithm to construct a topological graph from a 3D map. While we will not be proposing a new algorithm for decomposing an occupancy grid, we will briefly cover the process and borrow some terminology from [7].

The base occupancy grid is decomposed into convex regions of free space, which we will refer to as **submaps**. Using the aforementioned algorithms may result in rooms or hallways being divided into multiple convex regions, but that is not a problem. In Fig 5.4, these submaps are represented by different coloured areas. The submap boundaries where the robot can cross from one region to another are referred to as **portals** (or *critical lines* in [46, 70]), represented by the white rectangles in Fig 5.4. The topological graph is referred to as a **navigation graph** in this work. The navigation graph is a weighted, undirected graph and is defined by $G = (V, E)$ with a cost on each edge $c : E \to \mathbb{R}_{\geq 0}$. The vertices and edges are defined by the following:

- **Vertices** are the center points of each portal. Thus, each $v \in V$ represents a point $v \in \mathbb{R}^2$.
  Additional vertices that represent an area of interest (ex. start, goal) can be added to the graph.
- **Edges** are abstract representations of the connectivity between vertices. If there exists a path from one portal to another in the base occupancy grid without crossing any other portals, then we add an edge between the two portals. We initialize the cost with the minimum distance between the two vertices.



(a)



(b)

Figure 5.4: Example of occupancy grid decomposition (left) and the resulting navigation graph (right).

Localization is performed using existing algorithms such as AMCL [13] on the base occupancy grid. These algorithms are robust and can still localize reasonably well in the presence of unexpected obstacles. However, perfect localization is not required as edges are an abstract concept and Section 5.9 will allow for flexibility in vertex definitions, expanding it from a single point. The set of super maps will contain copies of the navigation graph with modified edge information. Therefore, the robot will only store the base occupancy grid, submap divisions, and disagreeing copies of the navigation graph instead of storing multiple copies of the occupancy grid. Another advantage of this approach is that small changes to the environment (like an additional chair being added to a room) that do not significantly alter the path a robot would take to reach the goal are not stored in memory.

One concern is the possibility of creating a multigraph where multiple edges share the same endpoints, such as in figure 5.5. While this scenario does not need to be avoided, if an implementation allows for the construction of a navigation multigraph, then each edge will need an explicit label instead of representing it with the endpoints $(u, v) \quad u, v \in V$, as is done traditionally. This enables the robot to differentiate the edges during planning and traversal.

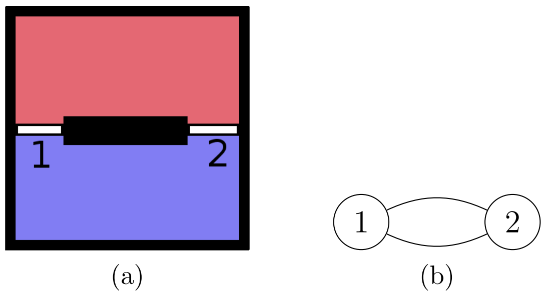


Figure 5.5: Example of a decomposition that could form a multigraph.

5.2.3 Edge Cost Update

The cost of edge (u, v) of the navigation graph is initialized with the Euclidean distance between u and v. This underestimates the true distance travelled by the robot, especially if the robot needs to turn or if the submap is cluttered. However, during every task execution, a moving average length of a valid path between u and v in $S(u, v)$ is maintained, resulting in the average path length for that particular task environment. At the end of the task, the averaged cost is input into another moving average over all the tasks executed thus far. Blocked edges are given a weight of infinity and are not included in the moving average. Edges that were not observed to be unblocked during the task execution are also not updated. The averages are updated using

$$\bar{c}_{\text{new}}(u, v) = \alpha|\bar{P}(u, v)| + (1 - \alpha)\bar{c}_{\text{old}}(u, v) \qquad \text{where } 0 \leq \alpha < 1 \tag{5.1}$$

where $\bar{c}(u, v)$ is the average, $\bar{P}(u, v)$ is the average path length of edge (u, v) in the recently finished task, and α is the filter constant. This approach is commonly known as an exponential moving average (EMA) [55] because the weight on older data is exponentially decreasing.

EMA was selected because of its simplicity and because it does not require storing any previous values aside from the current average. We chose to maintain a single weight for each edge instead of keeping a weight for each super map for the following reasons:

- Not all the edges of an environment will be observed by the robot during a single task execution and a weight is only updated if the edge state is unblocked by the end of the task execution. Therefore, for any given task, only a subset of edge weights will be known.
- We assume that if an edge is unblocked, its weight will stay relatively constant with small fluctuations for all the super maps.

5.3 Resolving Edges

In each task execution, the robot requires a method to determine if an edge is blocked or unblocked, i.e., to resolve the edge. Resolving an edge can be done via two different mechanisms: direct inference or indirect inference.

There is not much literature for verifying existing edge states. In papers that focus on constructing topological graphs, once a graph is constructed, the planner will assume the edges are unblocked. In papers that propose solutions for the CTP, it is assumed that the edge state can be determined at an edge endpoint. While this may be a reasonable assumption for driving (usually barriers are erected at an intersection if a road is impassable), this may not frequently be true while navigating indoors. For example, the robot may not realize a door on the side of a hallway is closed or locked until it gets closer to the door.

5.3.1 Direct Inference

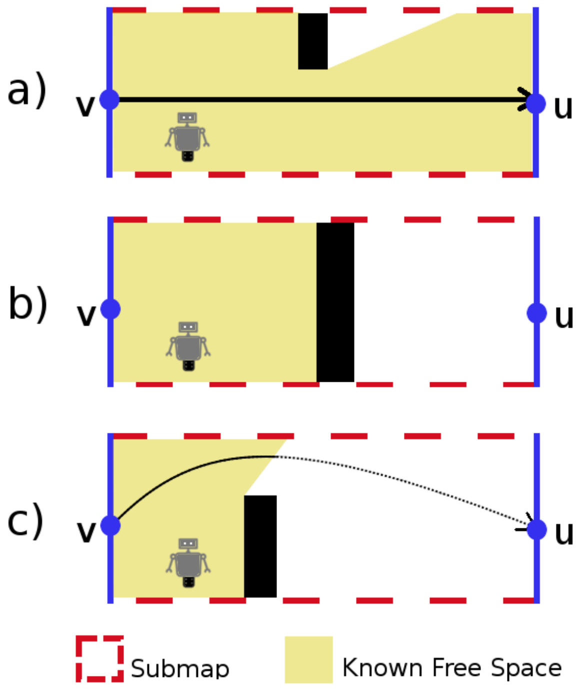


Figure 5.6: a) Example of unblocked edge; b) blocked edge; c) unknown edge

Direct inference uses the current costmap and the current location of the robot to update the state of the edges of the navigation graph during a task execution. It is continuously run at a user-specified rate. An edge can have one of three possible states: blocked, unblocked, or unknown. Resolving an edge means to set the state to be blocked or unblocked. The edges of the navigation graph are set to unknown at the beginning of every task.

While the robot is guaranteed to resolve an edge when it has traversed the entire length of the edge, a few properties of a 2D environment can be exploited to resolve the edge without traversing the entire edge.

Let the *true free space* be denoted by C where $C \subseteq \mathbb{R}^2$. This is the area of the environment the robot can occupy without colliding with an obstacle. We assume a disc-shaped robot along with a disc-shaped safety region so that orientation does not have to be considered. We also assume C does not change throughout the task execution. Finally, keep in mind that C is hidden from the robot.

Let the *known free space* be denoted by C_k where $C_k \subseteq C$. This is the area of the environment the robot has observed to be free space since the beginning of the task.

Let the *unknown space* be denoted by C_u where $C_u \subseteq \mathbb{R}^2$. This is the area of the environment that the robot has not observed yet, but is marked as free space by the base occupancy grid. Note that $C_u \cap C_k = \emptyset$.

Let the *optimistic free space* be denoted by C_o where $C_o = C_k \cup C_u$. This is the area of the environment the robot believes it can occupy, assuming unknown space is also free. We will also assume that $C \subseteq C_o$, which means obstacles in the base occupancy grid will be assumed to always be there.

Remark 3 (Implementation of C_k and C_u) *In practice, C_k and C_u can be obtained by using a layered costmap [48]. A costmap is an advanced occupancy grid where instead of each cell being either occupied, free, or unknown, a cost is assigned to each cell instead. Cells with a cost above a threshold are considered occupied. The cost can be affected by the distance a cell is from an occupied cell, therefore the costmap can set narrow spaces that the robot cannot fit through as occupied. With a layered costmap, it is possible to separate C_k and C_u into different layers.*

Let S denote a submap, then $C(S), C_k(S), C_u(S), C_o(S)$ is the true free space, known free space, unknown space, and the optimistic free space respectively, in submap S.

Let $\mathbf{x}_R \in \mathbb{R}^2$ be the current location of the robot. Now a formal definition of an unblocked and blocked edge can be given:

Definition 5.3.1 (Unblocked and Blocked Edges) *Given an edge $(u, v) \in E$ where $u, v \in V$, and $S(u, v) \subset \mathbb{R}^2$ is the submap associated with edge (u, v), the edge is* ***unblocked*** *if and only if there exists a path, $P \subseteq C(S)$ from u to v. Otherwise the edge is* ***blocked****.*

Lemma 1 *An edge (u, v) is unblocked if there exists a path from u to v in $C_k(S)$.*

Proof: Let $P(u, v)$ denote the path in $C_k(S)$. Since $C_k(S) \subseteq C(S)$, then it follows that $P(u, v) \subseteq C(S)$. □

Lemma 2 *An edge (u, v) is blocked if there does not exist a path from u to v in $C_o(S(u, v))$.*

Proof: By the definition of an unblocked edge, suppose there is a path $P(u, v) \subseteq C(S(u, v))$ but there does not exist a path from u to v in $C_o(S(u, v))$. This implies there exists $\mathbf{x} \in C(S(u, v))$ where $\mathbf{x} \notin C_o(S(u, v))$. But that is a contradiction since $C(S) \subseteq C_o(S)$. Therefore, $P(u, v) \subseteq C_o(S(u, v))$. □

Algorithm 4 exploits these properties to resolve edge states. This algorithm is continuously run at a user-specified rate because $C_k(S)$, $C_u(S)$ and $C_o(S)$ are dependent on the time passed since the beginning of the task. In practice, the maximum rate of the algorithm depends on the computation time of the algorithm used to find a path in lines 7 and 9.

To avoid checking edges that are not immediately relevant to the robot, an edge has to meet at least one of the following conditions to be observed (line 4):

1. The edge is in the same submap that the robot is currently positioned in.
2. At least one of the endpoints of the edge are within the maximum observation range of the robot at $\mathbf{x_R}$.

The maximum observation range of the robot is a circle with a radius of $r_{\max}$, which is based on sensor parameters and limitations of the obstacle detection. The motivation behind the dual conditions is to address the issues of having big submaps or small submaps.

Condition 1 forces the algorithm to check the edge states of all the edges in the current submap. If this condition was not in place, the robot would not resolve edges in submaps that were larger than the maximum observation range. If the submap is smaller than the maximum observation range, then only checking edges in that submap would be rather short-sighted if the robot is capable of resolving edges in other submaps that are close by. Condition 2 takes advantage of the robot's maximum observation range if possible.

Algorithm 4: directInference

```
   Input: $C_k$, $C_u$, $r_{\max}$, $\mathbf{x_R}$
1  $C_o = C_k \cup C_u$;
2  $S_{\text{curr}} = \text{currSubmap}(x_R)$;
3  for each $(u, v) \in E$ do
4      if $||\mathbf{x_R} - u||_2 \leq r$ OR $||\mathbf{x_R} - v||_2 \leq r$ OR $S(u, v) = S_{curr}$ then
5          if $u$ or $v \notin C_o(S(u, v))$ then
6              Set $(u, v)$ as blocked;
7          else if there exists $P(u, v) \subseteq C_k(S(u, v))$ then
8              Set $(u, v)$ as unblocked;
9          else if there does not exist $P(u, v) \subseteq C_o(S(u, v))$ then
10             Set $(u, v)$ as blocked;
```

Line 5 checks if either of the endpoints of edge (u, v) are in an obstacle, if they are in an obstacle, then the edge is marked as blocked. In Section 5.9, we present an extension of direct inference to be more robust by considering the entire length of the portal instead of just the center point.

Line 7 exploits Lemma 1 to set (u, v) as unblocked. In line 9, a path planner is run on $C_o(S(u, v))$ to find a path from u to v. If no path is found, then the edge is blocked. When selecting a path planner, it only needs to return if a path exists or not, thus optimality is not a concern. Minimizing the computation speed should be the primary objective. Planners that use heuristics (like A*) to speed up computation are good options.

If a path is found, but u and v do not satisfy the conditions in lines 5 or 7, this implies the robot still does not know whether the edge is blocked or unblocked, since the path could be blocked by an unseen obstacle, as illustrated by (c) in Fig 5.6. Then edge (u, v) cannot be resolved and direct inference moves on to the next edge to be checked.

Edges resolved by direct inference during task t are used to update the current map $M_t = (E_t^b, E_t^u)$ where $E_t^b, E_t^u \subseteq E$ and $E_t^b \cap E_t^u = \emptyset$, which will be tested for map agreement and stored as a super map (recall that E_t^b is the set of edges that are blocked, while E_t^u is the set of edges that are unblocked. Refer to Section 4.2.2 for the formal definition of a map). Computationally fast path planning is crucial to running direct inference at the desired rate. If the path finder (lines 7 and 9) is slow, the robot may resolve very few edges because during each *for* loop, the robot only resolves edges that satisfies either condition 1 or 2 at its current location, possibly resulting in missed edges because the robot has moved far away by the time direct inference is called again. Also large submaps may slow down the the rate of direct inference as most path finders will take longer to run on large maps.

Remark 4 *For ease of explanation, we have represented M as the tuple (E^b, E^u). However, in practice M is stored as an adjacency table representing a graph with edges marked as blocked, unblocked, or unknown. Therefore, we will also use M to represent the navigation graph induced by (E^b, E^u).*

5.3.2 Indirect Inference

Indirect inference makes assumptions about edge states that have not been resolved by direct inference. It infers an edge state based on historical information about the environment from previous task executions and the information that it has collected about the current environment. Since edge states are only assumed from historical data instead of inferred from data about the present environment, edges resolved by indirect inference do not update the current map, but the assumed edge state may affect how the robot navigates.

Indirect inference updates the belief the robot has about its environment. It is run at the same rate as direct inference is resolving edges. At the beginning of task t, the belief Y is set to all the super maps the robot has collected since the first task, $\mathcal{M}_{t-1}$. As the robot executes the task, it uses direct inference to resolve edge states and saves it in M_t. As M_t is updated, it is compared with the other super maps in $\mathcal{M}_{t-1}$. If super map $M_i \in \mathcal{M}_{t-1}$ *disagrees* with M_t (formal definition in Section 4.3.1), then i is removed from Y. Formally,

$$Y = \{i : M_i \text{ agrees with } M_t, M_i \in \mathcal{M}_{t-1}\}. \tag{5.2}$$

Let $M_k = (E_k^b, E_k^u)$ denote the *known map*, which stores all the edges that the robot believes it knows the state of in the current task t. The known map contains the edges that the robot has resolved with direct inference since it started executing task t, and all the edges that have a consistent state across all the maps in Y.

For example, in Figure 5.7, $Y = \{1, 2\}$, and all the relevant maps are provided. A few of the edges in M_k are set according to the following:

- Edges (2,3) and (1,4) are either unblocked or unknown in M_t, M_1, and M_2, therefore it is unblocked in M_k.
- Edge (1,3) is unknown in M_t, while it is unblocked in M_1 and blocked in M_2, therefore the edge is unknown in M_k.
- Edge (2,4) is blocked in M_t, so while it is unknown in both M_1 and M_2, the edge is also blocked in M_k.

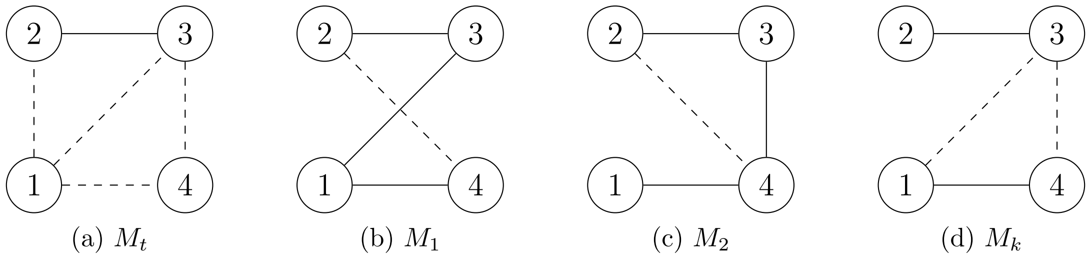


Figure 5.7: Example of how known map M_k is formed.

Formally,

$$E_k^b = E_t^b \cup \Big(\bigcap_{i \in Y} E_i^b \Big), \tag{5.3}$$

$$E_k^u = E_t^u \cup \Big(\bigcap_{i \in Y} E_i^u \Big), . \tag{5.4}$$

The current map M_t and the known map M_k differ in that the current map only has the edge states that the robot has 'seen' with its sensors in the current task, with all other edges being unknown (thus direct inference), while the known map also contains state information about edges that the robot has not yet seen but should be true based on the belief (indirect inference).

5.4 Robot Actions

Here we formally define the actions that the robot can take when navigating. Resolving edges is not considered an action because it is being constantly performed through direct and indirect inference. The following are actions that the robot will decide to take depending on the results of direct and indirect inference.

The robot can *move* from vertex v to vertex u if $(v, u) \in E$ and the state of the edge is unblocked or unknown. A path planner such as A* with smoothing will compute a detailed path from v to u in the corresponding optimistic free space of the submap for the robot to follow. Suppose the robot is moving from v to u, then a move is complete if $||\mathbf{x_R} - u||_2 \leq \epsilon$ for some predefined threshold $\epsilon \in \mathbb{R}$. It is possible for the robot to be unable to complete a move because an unknown edge turns out to be blocked. In that case the policy will decide where it will go next. A *leg* is a sequence of vertices that defines the moves the robot will make when following the policy.

Definition 5.4.1 (Leg) *A sequence of vertices $v_1, v_2, v_3, \ldots, v_n$ that represents the sequence of moves v_1 to v_2, then v_2 to v_3, etc, up to v_{n-1} to v_n where n is the number of vertices in the leg.*

The robot can *observe* an edge (v, u) by resolving it via direct inference. This is a combination of intentionally following a leg to one of the end points of the edge, say $v_n = v$, then moving from v to u until the edge has been resolved. An observation is not complete until (v, u) has been set as blocked or unblocked.

Definition 5.4.2 (Observation $O = (e, v)$) *A tuple where $e = (v, u) \in E$ is the edge to be resolved by direct inference. The robot will move to v, then move from v to u until the edge has been resolved.*

The robot can *terminate*, or end the task. It will only do so when it reaches its goal. Thus, the robot can choose to do one of the following three actions at any given moment during a task execution: move, observe, and terminate.

5.5 Policy Construction

In this section, we will discuss three different motion policies for navigating to a goal: optimistic, offline, and online. These methods will be compared through experiments in Section 5.7.

5.5.1 Optimistic Policy

The optimistic policy is one of the most prevalent policies in the literature for navigating uncertain environments (see Section 2.2). In the context of this chapter, the optimistic policy does not use the observe action. Informally, it uses a shortest path algorithm such as A* or Dijkstra to compute the shortest path from its current location to the goal on the navigation graph. If the move cannot be completed because of an unexpected blocked edge, then it recomputes and follows the shortest path in the updated map.

When the robot discovers a blocked edge, we insert a vertex v_R containing the robot's current position $\mathbf{x_R}$ into the navigation graph, and run the shortest path algorithm with v_R as the starting vertex. Let v be the last vertex the robot was at. Then for every vertex u corresponding to a portal in the submap the robot is currently in, if (v, u) is not blocked, an edge is added from v_R to u. An edge is also added to v. The cost of each edge is $||\mathbf{x_R}, u||_2$.

Similar to the previous chapter, the optimistic policy is also a contingency plan for the following two policies that will be introduced. If the robot encounters an unexpected obstacle in the environment, it will switch to the optimistic policy to reach the goal. Therefore, the robot is guaranteed to terminate at the goal.

5.5.2 Offline RPP Policy

The offline RPP policy is constructed by calling the Reactive Planning Problem (RPP) solver (Algorithm 1) before every task execution, like the policy builder in the chapter 4. As a reminder, it accepts the set of super maps and the estimated pmf as input and outputs a policy for the robot to follow. The policy is encoded as a binary tree that guides the robot on what actions it should take based on observations it has made since the beginning of the task execution. If an edge is unexpectedly blocked (i.e. move cannot be completed), the robot will switch to the optimistic policy. This policy is considered offline because it is pre-computed before the task begins. It predicts how the known map M_k will change and pre-determines actions accordingly.

Since Algorithm 1 was designed to compute a policy for graphs with the assumption that the robot can resolve an edge at an endpoint, the following minor modification needs to be made. Given an observation (e, v) where $e = (u, v)$, u needs to be appended to the leg to v. The robot does not need complete the move on e, it merely needs to resolve e with direct inference to complete the observation, but u needs to be appended to guarantee that the observation will indeed be completed.

5.5.3 Online RPP Policy

A weakness of the offline policy is the need to make assumptions about where edges can be resolved to predict not only the next best action, but subsequent actions after. Since the offline policy assumes edges can be resolved at an endpoint when planning observations, it may lead to some poor decisions as illustrated in figure 5.8.

In panel 1, the robot observes edge (1,6) as unblocked. However, in panel 2, the next observation the offline policy selects is (2,7) because the policy assumed the robot would be at portal 1 when it completes its observation, but in reality the robot was more than halfway to portal 6. It completes observing (2,7) in panel 4, and in panel 5 the policy chooses to observe (2,3) (in this environment portal 3 is always blocked at the same time as portal 12). Again it assumed the robot would be at portal 2 when the observation is completed. The robot finally makes its way to the goal at panel 6, after observing (2,3).

The robot ended up observing edges (1,6), (2,7), and (2,3), but it did not complete the observation until it was at least half way to the next vertex for all of them. This resulted in a significantly worse travel distance than even the optimistic policy.

Another weakness is that the offline policy is not flexible if the belief changes in a way it did not expect. The robot will rigidly follow the policy until it either reaches the goal or a move fails and it switches to the optimistic policy.

Instead of pre-calculating the full policy, an alternative approach is to decide the next best action based on the current position of the robot, the current knowledge of the environment configuration, and the belief. The next-best action is re-calculated only when the belief over the super maps changes. As a reminder of how actions are selected by Algorithm 1, assuming the robot starts the task at v, it first decides between exploration and exploitation by evaluating the following expression for all vertices u,

$$c_{M_k}(v, g) \leq c_{M_k}(v, u) + \mathbb{C}_Y(u, g), \tag{5.5}$$

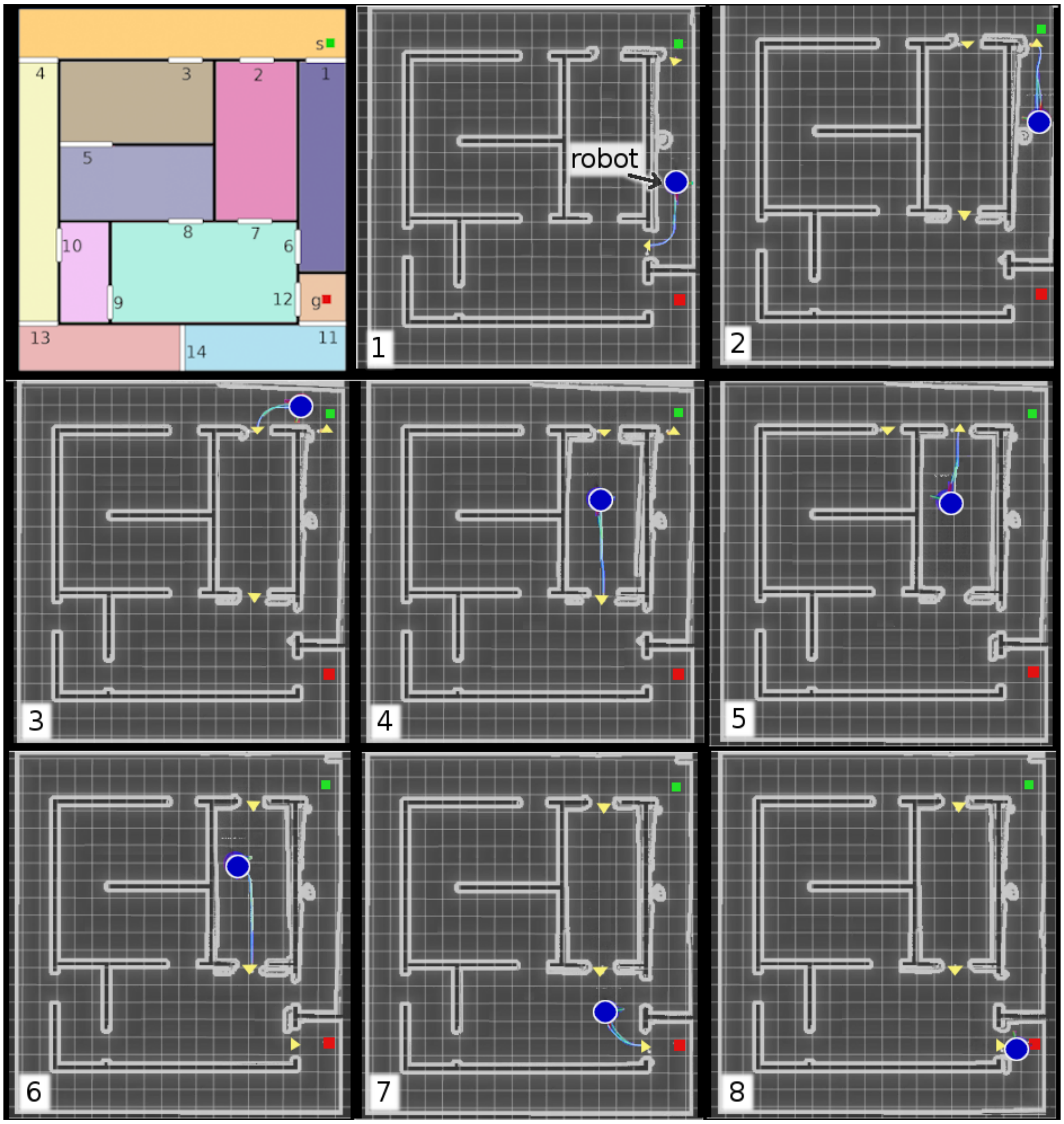


Figure 5.8: This is a sequence of actions the robot took in 19th task of one of the simulation trials following the offline policy. The robot starts at the green square and the goal is the red square. The yellow triangles indicate the current leg the robot is following.

where M_k is the known map and $\mathbb{C}_Y(u, g)$ is the expected cost from u to g given the current belief Y. If every u satisfies (5.5), then exploitation is chosen, meaning the next best action is to move to the goal. Otherwise exploration is chosen, meaning the next best action is an observation. Let $\mathcal{O}$ contain all the (e, u) pairs that do not satisfy (5.5). The best observation is selected by evaluating the following:

$$O_{min} = \underset{(e,u)\in\mathcal{O}}{\operatorname{argmin}} \Big[(c_{M_k}(v, u) + \mathbb{C}_Y(u, g))\mathbb{E}[H(X_Y|e)]\Big]. \tag{5.6}$$

The expected entropy after observing e is denoted by $\mathbb{E}[H(X_Y|e)]$. For every observation, two actions must be selected, one action for when the observed edge is blocked, and a different action if the observed edge is unblocked. This is repeated until every branch in the policy tree terminates at the goal. For details, please refer to the background chapter, Section 3.4.

In the online policy, we modify Algorithm 1

- to only return the next best action instead of the full policy,
- to temporarily add a new vertex encoded with the robot's current position in the belief's super maps, and
- to modify the *distance score* $(c_{M_k}(v,u)+\mathbb{C}_Y(u,g))$ of the observation e to better reflect the new reality that the edge state may not be resolved at an endpoint without first traversing the edge.

Algorithm 5 below reflects the first two modifications. We extract the inside of the *for* loop that generates each child of the policy tree in Algorithm 1 to only calculate the next best action (lines 8-14). In lines 1-7, the new vertex encoded with the robot's current position is added to each super map in the belief and M_t, in the same way when a new vertex is added to M_t in the optimistic policy. Line 8 finds the (e,u) pairs whose distance score $D(e,u) < c_{M_k}(v_R, g)$, resulting in a list of (e,u) pairs $\mathcal{O}$ and their corresponding distance scores in D. If no observations are in $\mathcal{O}$, the robot will proceed to the goal. Line 13 then multiplies each distance score by the expected entropy of their respective edge e, selecting the next observation to be the (e,u) pair that has the minimum product.

Algorithm 5: onlineRPP

```
   Input: current map $M_t$, super maps $\mathcal{M}_t$, robot location $x_R$, last vertex $v$, belief $Y$,
          estimated probabilities $\hat{P}$
   Output: Observation $O$, Leg $L$
 1 $S_{\text{curr}}$ = currSubmap($x_R$);
 2 $v_R$ = vertex with position $x_R$;
 3 Add $v_R$ to $M_t$ and $(v_R, u)$ for each portal $u$ in $S_{\text{curr}}$;
 4 for $i$ in $Y$ do
 5     Add $v_R$ to $M_i \in \mathcal{M}_t$;
 6     Add edge $(v_R, u)$ if $(v, u)$ is an unblocked edge and $u$ is a portal in $S_{\text{curr}}$;
 7     Compute $c_{M_i}(v_R, g)$;
 8 Compute $(\mathcal{O}, D)$ =PossibleObservations($Y, M_t, \mathcal{M}_t, \hat{P}$);
 9 if $|\mathcal{O}| = \emptyset$ then
10     $O = \emptyset$;
11     $L$ = shortestPath($v_R$,$g$);
12 else
13     $O = \text{argmin}_{(e,u)\in\mathcal{O}} D((e,u))\mathbb{E}[H(X_Y|e)]$;
14     $L$ = [shortestPath($v_R$, $u_1$),$u_2$];
15 return $O$, $L$;
```

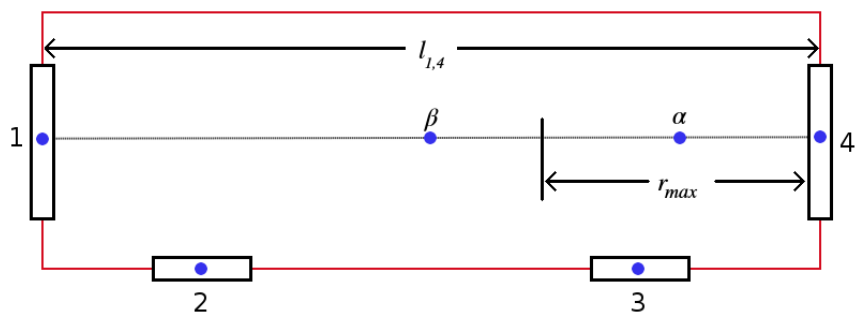


Figure 5.9: Illustration to help explain the modified distance score. The maximum observation range is specified by $r_{\max}$, while the red rectangle represents a submap.

To modify the distance score, we first look at Figure 5.9. Consider the following scenario: Suppose the robot is considering the observation ((1,4),1). The distance score of this observation would be

$$D((1,4),1) = c_{M_k}(v_R, 1) + \mathbb{C}_Y(1, g). \tag{5.7}$$

However, this score fails to account for the distance the robot may have to travel to complete the observation and the distance it will have to travel from there to the next vertex, which could be significant if the submap is large. The expected cost

$$\mathbb{C}_Y(1, g) = \sum_{i \in Y} p(X_Y = i) c_{M_i}(1, g) \tag{5.8}$$

only calculates the shortest path from 1 to g in each super map of the belief. The probability of the robot being in map i given the belief is denoted by $p(X_Y = i)$. We propose an alternative score that predicts where an observation will be completed to better estimate the expected cost the robot will have to travel after completing an observation. Assuming the robot can complete an observation at 1 is the best case scenario, however, the expected case can be divided into two cases:

A. If (1,4) is unblocked, the robot must be at least $r_{\max}$ away from 4 to resolve the edge as unblocked, so the expected location would be at α, $r_{\max}/2$ away from 4.

B. If (1,4) is blocked, the robot could resolve the edge as blocked from anywhere between 1 and 4, so the expected location would be in the middle, at β.

Combining these two cases and generalizing to any observation (e, u), the new distance score is

$$\begin{aligned} D(e, v) = \quad & c_{M_k}(v_R, v) \quad + \\ & p(e \in E^u|Y)(||v, \alpha||_2 + \mathbb{C}_{Y^u}(\alpha, g)) \quad + \\ & p(e \in E^b|Y)(||v, \beta||_2 + \mathbb{C}_{Y^b}(\beta, g)), \end{aligned} \tag{5.9}$$

where $p(e \in E^u|Y)$ and $p(e \in E^b|Y)$ are the conditional probabilities of e being unblocked and blocked respectively given Y. In equation 5.9, Y^u is the partition of the belief where

e is unblocked and Y^b is the partition of the belief where e is blocked. The expected costs $\mathbb{C}_{Y^u}(\mu, g)$ and $\mathbb{C}_{Y^b}(\beta, g)$ can be calculated using

$$\mathbb{C}_{Y'}(x, g) = \min \left\{ ||x, v'||_2 + \mathbb{C}_{Y'}(v', g) \right\} \qquad \forall v' \in V(S(e)) \tag{5.10}$$

where $V(S(e))$ are all the vertices of the submap associated with e. Equation 5.10 finds the portal, or the vertex v' that has the minimum expected cost from $x \in S(e)$ and returns the expected cost if the robot were to move from x to v' to g.

Remark 5 *The accuracy of the updated distance score depends on the assumption that the submaps are convex. While a distance score will still be calculated if the submaps are not convex, it may not be able to accurately reflect the real costs of taking the observation.*

5.6 LAMP Framework

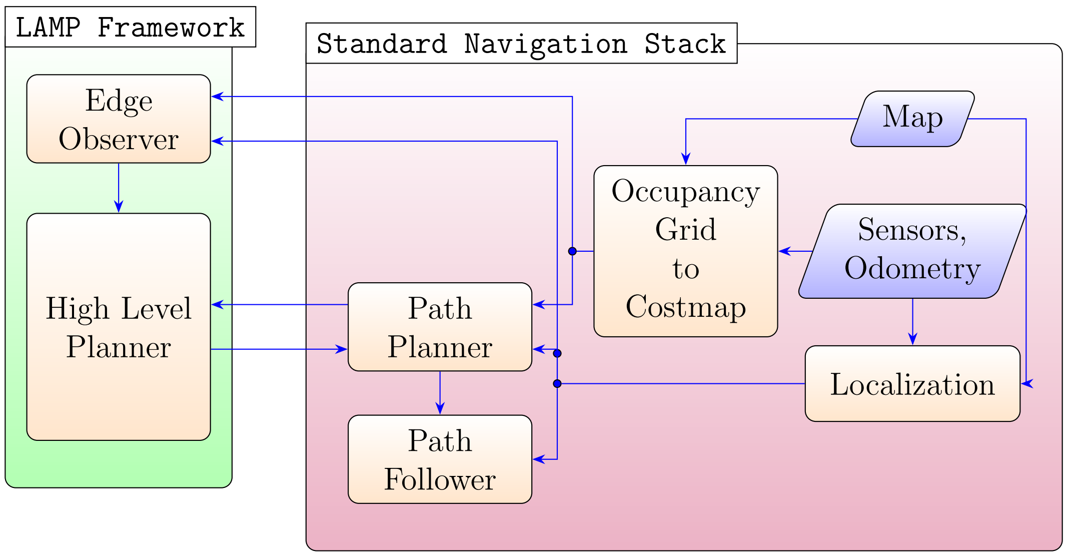


Figure 5.10: Overall LAMP architecture

The architecture of the LAMP framework is shown in figure 5.10. The blue arrows represent the flow of data between the nodes. The framework is meant to be added to a standard navigation stack. Although the components in the navigation stack in figure 5.10 are based on the ROS navigation stack [50], others have a similar structure [23].

The edge observer takes the costmap and the estimated robot's pose from the localization system to resolve edge states via direct inference. The edge states are given to the high level planner, which uses it to build maps of the environment. The high level planner implements the policies mentioned in Section 5.5 and instructs the path planner on which vertex to go to next.

The path planner is responsible for planning a smooth path from the robot's current position to the next vertex dictated by the high level planner. The path must be within the submap the robot is located in. If no such path is found, the high level planner will set the current edge state to blocked. In our implementation, we integrate our algorithm with the ROS navigation stack, which only uses laser range data and odometry. However, other navigation stacks may be able to incorporate other data such as camera feeds to aid localization. The provided map is expected to be an occupancy grid. The occupancy grid to costmap component takes the provided map and combines it with sensor data to generate a costmap. The ROS navigation stack uses a layered costmap so permanent and semi-static obstacles can be differentiated, in addition, semi-static obstacles from previous task executions in the costmap are cleared at the beginning of each new task. To minimize computation, one could limit the output of the costmap to the submap that the edge observer and path planner currently need instead of passing the full costmap.

The remainder of this section will detail how the policies detailed in Section 5.5 are executed in the high level planner.

5.6.1 Task Execution with Offline Policy Construction

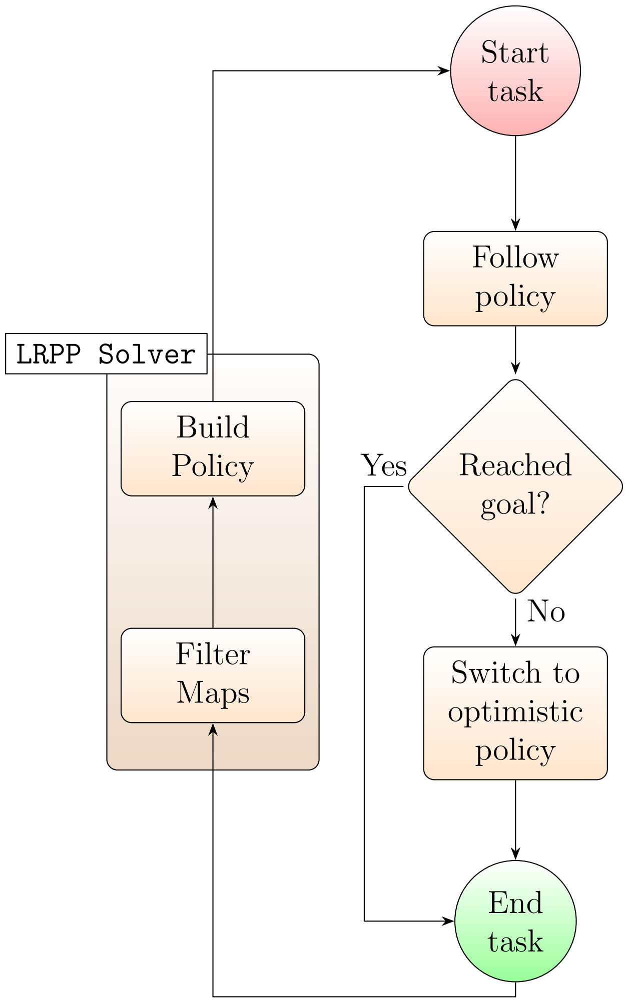


Figure 5.11: Structure of High Level Planner with offline policy construction.

The task execution is the same as the Sequential Task Completion algorithm in chapter 4, which is summarized in figure 5.11. The black arrows represent the order which processes are executed. The high level planner maintains a filtered archive of maps of previous task executions, and creates a motion policy based on past runs that minimizes the expected travel cost. A map of the environment, M_t, is collected at the end of each task execution with edge states set as blocked, unblocked, or unknown. These edges are resolved by the edge observer. This map is then compared with previously saved super maps by the map filter (see Section 4.3.1) and updates the set of super maps and their estimated probabilities of being encountered in the next task execution. The policy builder (see Section 4.3.5) accepts the set of super maps as input and outputs a policy for the robot to follow in the next task execution. If an edge in a leg of the policy is observed to be blocked, the robot will switch to the optimistic policy. The belief of the robot only changes when an intentional observation (planned by the policy) is made, other observations are ignored.

However, in this new environment model, two changes need to be made in how the robot executes the policy. First, given an observation (e, v), the policy assumes the robot will be located at v when it completes the observation, so the next leg will begin with v. Suppose the leg is $[v, v_2, v_3, \ldots, v_n]$. If the robot is in the same submap as (v, v_2), then the robot will backtrack to v before going to v_2, which is undesirable. On the other hand, the robot cannot always skip v and move to v_2 because there is the possibility that (v, v_2) is in a different submap than the robot is currently operating in. In that situation, the correct move is to proceed to v first, then move to v_2. Thus before following a leg, the robot must first compare the submap it is in with the submap of (v, v_2), if they are the same then move to v_2, otherwise move to v.

Second, given the observation (e, v), the robot must follow the leg to v before it can begin execution of the next action in the policy. The reason for this additional rule is because it is now possible for direct inference to resolve edges that are not incident to the robot's current vertex. Since the offline policy assumes the robot will complete the observation at v, we enforce this rule so that executing the next step in the policy is simple.

Remark 6 (Using other CTP policies) *It is straightforward to modify the Build Policy node to construct different policies other than the RPP policy. The collected maps can be used to create a probabilistic graph, and other CTP policies can be constructed using the graph as an input. No other nodes will need to be modified since most other CTP policies also assume that edge states can be observed from their endpoints.*

5.6.2 Task Execution with Online Policy Construction

Figure 5.12 summarizes the procedure in the high level planner if it is using an online policy. Similar to the offline execution, at the end of every task execution, the current map is used to update the set of super maps that onlineRPP will use to help determine the next best action. In the online policy, the belief is updated as the current map is updated, and if the belief changes, then onlineRPP is called. As with the offline policy, if the current experience does not agree with any of the super maps, the robot will switch to the optimistic policy. Unlike the offline policy, the online policy can react to unexpected information that does not affect its current plan, but changes the belief (such as an edge that was expected to be blocked but is unblocked). Also unlike the offline policy, no special treatment needs to be given to the legs that the onlineRPP outputs, the robot can follow them as is, ignoring the first vertex which is always the robot's current location.

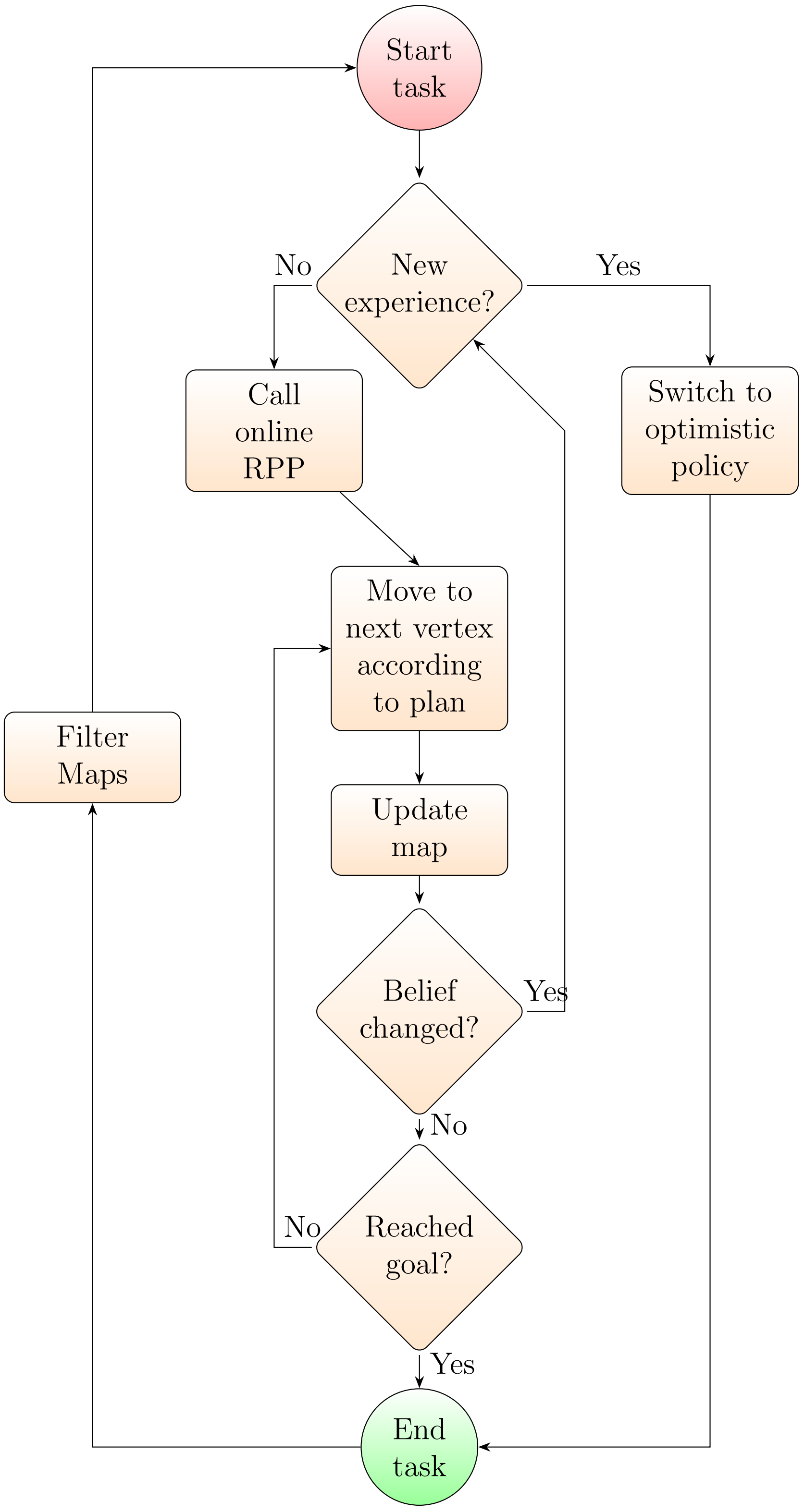


Figure 5.12: Online flow

5.7 Results

Two experiments were run to test the LAMP framework. The first was a simulation in an environment that has similar structure to an office or a small warehouse. The second was an experiment on a real robot that was run in a maze-like environment constructed indoors. In both cases the navigation graph was not a multigraph and in each task execution there existed a path to the goal.

All of the code for the high level planner and edge observer is in Python 2.7, which is integrated as a package for ROS Kinetic. The LAMP framework was built on top of the ROS navigation stack, which already includes all the components in the standard navigation stack in figure 5.10. In the ROS navigation stack, the path planner is referred to as the global planner and the path follower as the local planner. Table 5.1 outlines which ROS packages and therefore which algorithms were used for each component of the navigation stack in our implementation. Our high level planner uses the move_base action library to interact with the ROS navigation stack.

Component	ROS Package	Algorithm
Path Planner	global_planner	A*
Path Follower	base_local_planner	Dynamic Window Approach [14]
Localization	amcl	Augmented Monte Carlo Localization [13]
Costmap	costmap_2d	Layered costmap [48]

Table 5.1: Existing algorithms were used in the navigation stack for the experiments.

The Visilibity library [59] was used to estimate the known free space for the edge observer. Converting the occupancy grid into a hybrid map were done manually for each experiment.

5.7.1 Simulation

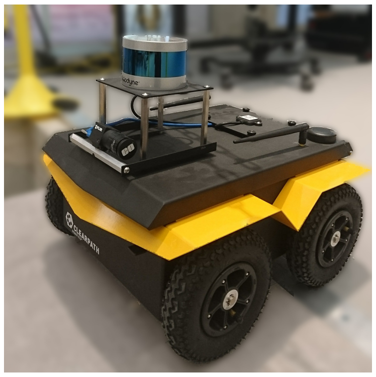
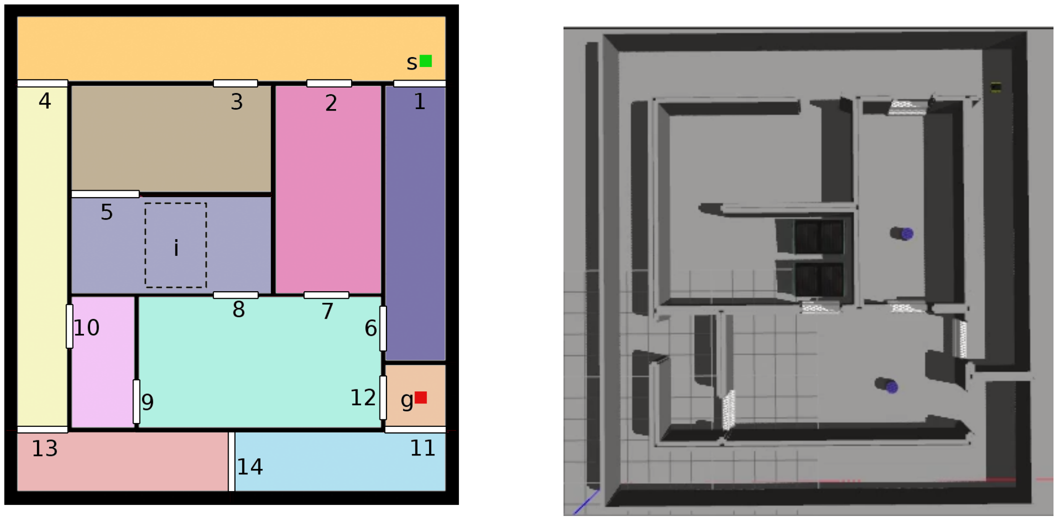


Figure 5.13: Clearpath Jackal that was used in both the simulation and real world experiment.

The simulations were run on a PC with a 4.2 GHz Intel Core i7 processor with 32 GB memory and a GeForce GTX 1060 GPU with 6 GB of VRAM, using the ROS Gazebo



(a) (b)

Figure 5.14: (a) Base occupancy grid of the Gazebo test environment (20 m $\times$ 20 m) showing submap decomposition, labelled portals, and a potential obstacle i. The start and goal vertices are labelled as s and g respectively. (b) Example of Gazebo environment configuration with obstacles. In this example, obstacles appeared in 2, 6, 7, 8, 9, and i, along with some random debris.

Obstacles	Probability
1	0.1
3,10,12	0.6
2,6	0.5
7,8,9	0.4
i	0.9

Table 5.2: Probabilities of obstacles appearing in the simulation.

simulator. The robot platform is a Clearpath Jackal, a small differential drive unmanned ground vehicle (0.5m $\times$ 0.5m footprint) that has been equipped with a SICK LMS111 LiDAR. Figure 5.14 depicts a top-down view of the environment the robot operated in and table 5.2 shows the probability of edge blocking obstacles appearing. If an obstacle appears on a portal, a white barrier blocks the portal like in figure 5.14b. If the region i is to have an obstacle appear, then two rectangular obstacles can appear anywhere in the submap, but in the same orientation as in (b). In addition, to simulate scattered debris, up to 10 blue cylinders will spawn in random locations throughout the environment.

The occupancy grid was converted into a graph with 16 vertices and 33 edges. The uncertainties in table 5.2 resulted in 32 possible configurations for the test environment, not accounting for all the different configurations the blue cylinders could be in.

The robot is tasked with repeatedly going from s to g. We ran 10 trials with 100 task executions each. The environment configuration for each execution was randomly selected according to the probabilities in table 5.2. We tested the following four policies for each trial: Simple, Optimistic, Offline, and Online. The latter three policies were explained in section 5.5. The Simple policy does not use the navigation graph, we simply send the goal position g to the navigation stack and the path planner guides the robot to the goal. This is the baseline we compare our policies to. The reasoning is that this is available out-of-the-box by ROS and presumably many robots will use this method of navigation.

Figure 5.15 compares the average distance travelled by each policy after performing a certain number of task executions, normalized by the average distance travelled by the simple policy. All the data is shown except for an outlier by the Optimistic policy that resulted in -21% after 5 tasks, most likely due to an incorrect edge resolution resulting in a unnecessarily long path.

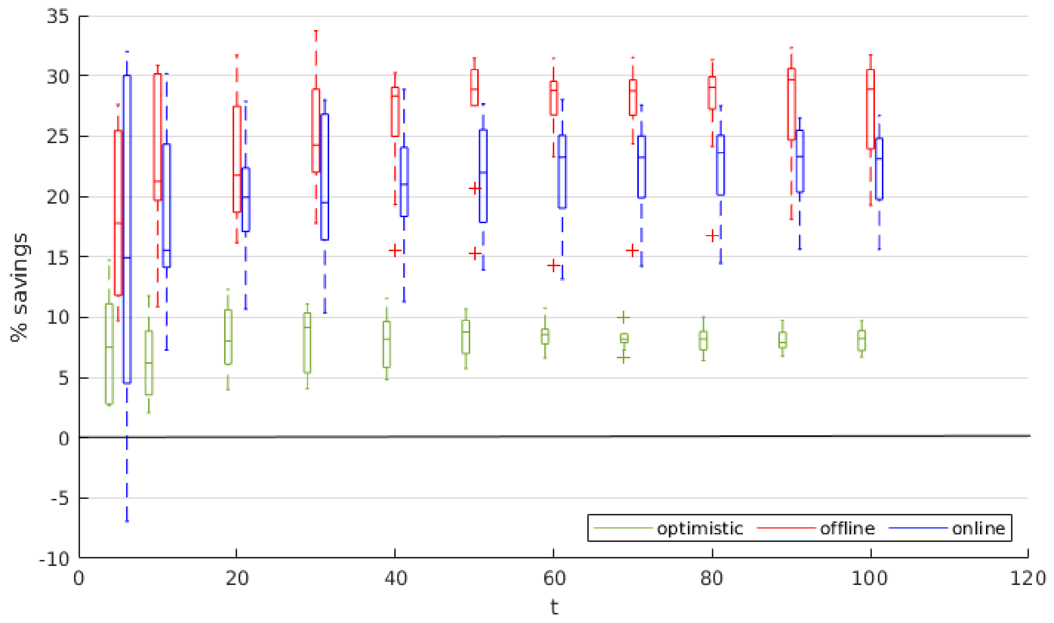


Figure 5.15: Cost savings of optimistic, offline, and online policy normalized to the simple policy.

Both the Online and Offline policies performed 10-20% better than the Optimistic policy, and 20-30% better than the Simple policy. The savings stabilize at 25% and 30% respectively, astonishingly similar to the performance of the solution in chapter 4. Unfortunately, the Online policy did not perform as well as the Offline policy, which was unexpected. From inspecting the paths the robot took, scenarios like the one in figure 5.8 did occur and the Online policy had better paths in those. One possible reason is because the Online policy reacts to changes in all edge states, rather than just the next assigned observation, there were instances when it was better to continue with the original plan. Yet the Online policy switched to the Optimistic policy because the environment no longer agreed with any of the super maps. However, this still does not explain why the Online policy does not eventually overtake the Offline policy since it should perform worse only at the beginning of the trial.

Another plausible explanation is that the edge observer sometimes incorrectly resolves an edge state because of error in the obstacle detection of the costmap. Particularly when the robot is positioned at an acute angle to the portal, the costmap sometimes reports no obstacles, resulting in the Online policy re-planning prematurely, possibly switching to the

Optimistic policy. The edge observer corrects itself later on, but the robot cannot switch back to the Online policy afterwards. Figure 5.16b seems to suggest that this does happen, since the Online policy consistently switches to the Optimistic policy more frequently than the Offline policy, even after 100 tasks despite the number of super maps being similar in both policies. As a side note, for both policies the rate of calls to the Optimistic policy decreases exponentially as the number of completed task executions increase, which is the desired behavior.

Although figure 5.15 shows the Optimistic policy outperforming the Simple policy, in the case of this test, it was by chance. The Simple algorithm always initially plans a path along the leg $[s, 2, 7, 12, g]$ (one smooth path from s to g, but it goes through those sequence of portals), whereas the Optimistic policy always sets the initial leg to be $[s, 1, 6, 12, g]$. This difference is due to the navigation graph edge costs being initialized to the euclidean distance between portal centers, not accounting for the robot's turn radius, while the path planner actually computes a smooth path in the cost map. Notice that in table 5.2, the barriers on portal 2 and 6 are correlated, but 7 is not. When portal 6 is not blocked, neither is portal 2, but portal 7 may be blocked. By going to 6 first, the Optimistic policy can view whether 7, 9, or 12 is blocked, but the Simple policy may have to backtrack from edge (2,7) to portal 6 before learning this information. All this to say, if the edge costs were initialized with a more accurate value, the Optimistic policy's average travel distance compared to that of the Simple policy's should be close to 0%.

The map filter is performing as expected, figure 5.16a shows the number of super maps plateauing as the number of completed task executions increases. Out of all the trials, the maximum number of super maps that were stored after 100 tasks was 20. Figure 5.17 contains two examples of different environments but their maps were in agreement, resulting in considerably fewer maps being stored than if we were to compare the occupancy grids directly.

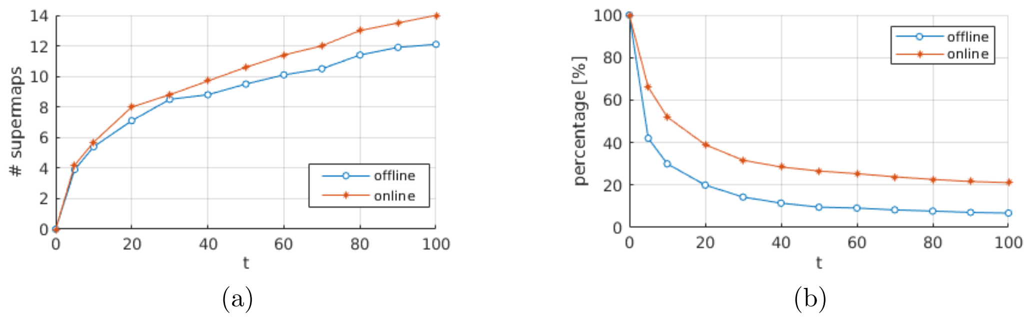
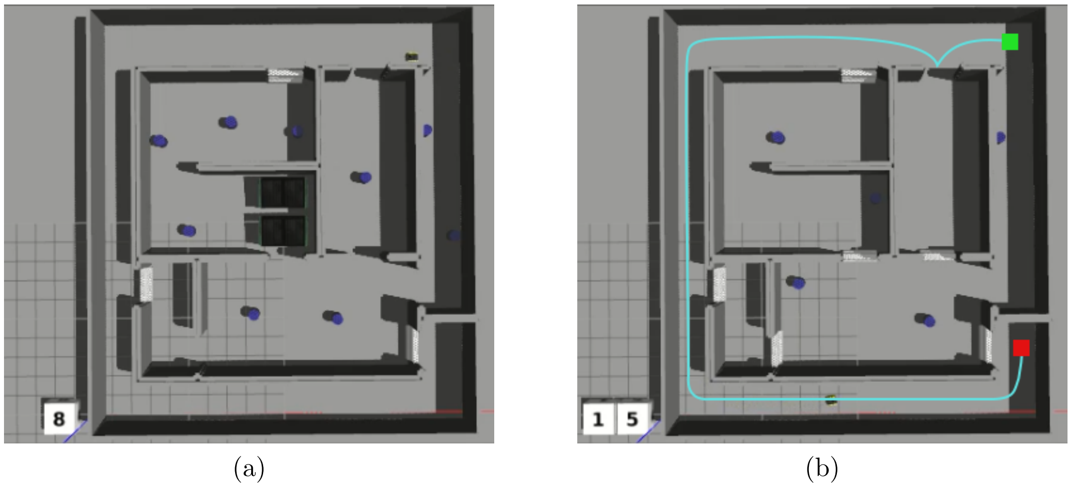


Figure 5.16: (a) Average number of super maps stored compared to the number of tasks that have been executed.(b) Percentage of executed tasks where the high level planner switched to the optimistic policy.

Although minimizing computation time is not the focus of this work, it should be noted that during the simulations, the edge observer was operating at a rate of roughly 3 Hz, or observing 3 edges per second. While this was slower than desired (resulting in some edges remaining unknown when they should have been resolved), the Offline and Online policies still performed better than the Simple policy. However, there is room for optimization in



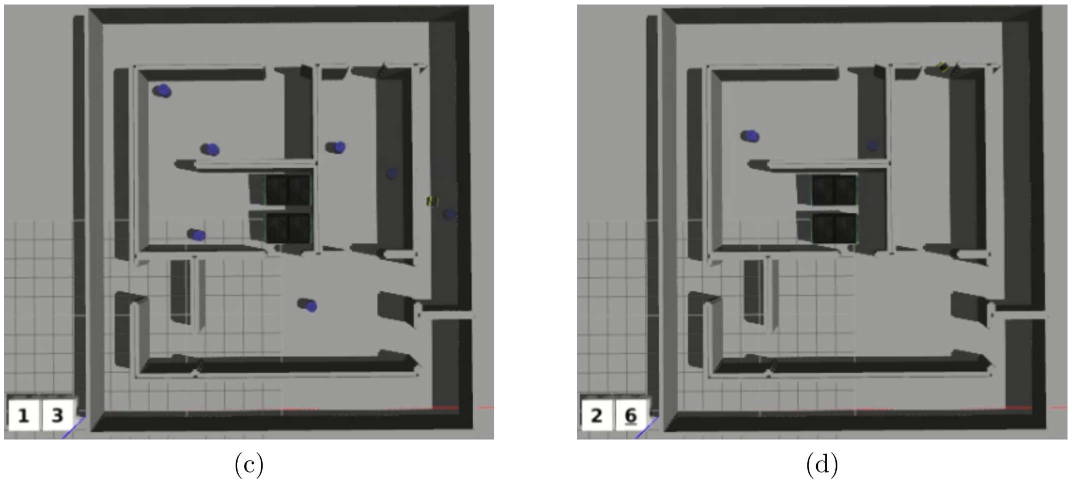


Figure 5.17: Two examples of different environment configurations that satisfy map agreement. The top environments agree because in task 15, the robot did not see the barriers in the center along its route (light blue). The bottom environments agree despite the blue cylinders being in different locations.

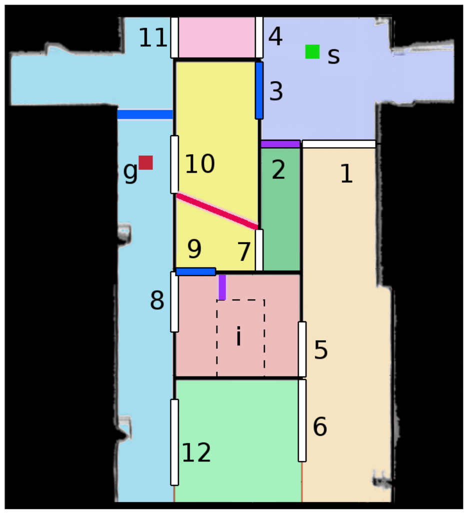
the direct inference algorithm and employing a faster path finder than A* would likely increase the edge observation rate.

5.7.2 Physical Robot Experiments

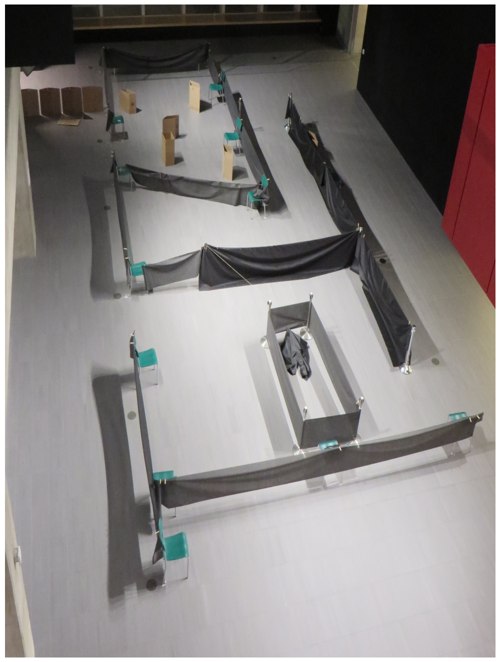
Obstacles	Probability
dark blue	0.6
purple	0.3
pink	0.9
i	1.0

Table 5.3: Probabilities of obstacles appearing in the environment.

Similar to the simulation, a Clearpath Jackal was used, this was equipped with a



(a)



(b)

Figure 5.18: (a) Base occupancy grid of real environment (20 m $\times$ 10 m) with submap decomposition, labelled portals, and potential obstacle locations. The start and goal vertices are labelled as s and g respectively. (b) Example of environment configuration with obstacles. In this example, i, the dark blue, and pink obstacles exist, along with some debris in the yellow submap.

Velodyne Puck, a 360°LiDAR. The environment that the robot operated in is shown in figure 5.18 and the probability of obstacles appearing in table 5.3. The occupancy grid provided to the robot in (a) was generated by running the Cartographer SLAM algorithm [22] in the empty area with the robot, and the permanent obstacles were later added with a drawing program.

This test environment has 14 vertices with 35 edges and 8 different configurations. We ran 2 trials with 10 task executions each, comparing only the Simple and Offline policies. Again, the environment configuration for each task execution was randomly selected according to the probabilities in table 5.3. However, because of time constraints, the simple policy was not run in the second trial. Instead, we averaged the distance travelled in identical configurations in trial 1 and used that value for the distance travelled in the same configurations in trial 2.

In trial 1, the average expected cost of the Offline policy was 5.06% less than the Simple policy, and in trial 2 it was 10.42%. While the results are positive for only 10 task executions, they are not conclusive of the potential of using the Offline policy. It can be seen that the Offline policy performed comparably or better in most of the task executions, but its gains were offset by a few very poor runs (tasks 3,6,9 in trial 1 and task 8 in trial 2). Its performance was severely hampered by errors in the edge observer, resulting in super maps with configurations that were impossible and wrong decisions being made. While these errors did happen in the simulation, it was rare and did not make much of a difference.

Here, although there were only 8 possible configurations for the environment, trial 1 had 9 super maps after 10 task executions, and 5 of the super maps were impossible (at least one of the following edges were resolved as blocked: (8,12),(8,g),(8,10),(10,g),(1,5)). In addition, in task 6 of trial 1, the offline policy observed edge (3,2) to be blocked when it was in fact, unblocked, as a result it also assumed edge (11,g) was blocked, leading it to take a much longer path to g.

The majority of these mistaken observations were due to the localization error exceeding 1 m. Since the vertex location is the center of the portal, if that pixel is occupied then the edge observer resolves any edges that are incident to that vertex as blocked. Although the portals were a minimum of 3 m wide, many edges were incorrectly observed as blocked. This highlights that only considering the center of a portal is not sufficient for accurate edge observations, and the importance of accurate localization when using the LAMP framework.

Trial	Task	Simple [m]	Offline [m]	Number of super maps	Switched?
1	1	74.54	60.41	2	Yes
	2	70.24	39.24	3	No
	3	10.28	32.83	4	Yes
	4	59.67	53.52	5	No
	5	60.77	52.99	5	No
	6	10.08	45.39	5	No
	7	23.80	13.39	6	No
	8	75.08	57.32	7	No
	9	22.15	42.84	8	No
	10	59.92	45.00	9	Yes
	Average	46.65	44.29		0.3
2	1	10.18	10.75	2	No
	2	74.54	67.40	3	Yes
	3	65.14	57.64	4	Yes
	4	10.18	10.79	4	No
	5	65.14	31.92	5	No
	6	10.18	11.36	5	No
	7	10.18	10.90	5	No
	8	65.14	87.77	6	Yes
	9	10.18	10.83	7	No
	10	74.54	54.82	8	Yes
	Average	39.54	35.42		0.4

Table 5.4: Results from experiment on real robot.

5.8 Discussion

The results show the potential the LAMP framework has to decrease the expected cost of a navigation task over time if there are any savings to be had. The test environments that the experiments were run in both had at least one key indicator of whether or not there was a dead end later on. In the simulation, if portal 2 was blocked, it meant portal 6 would also be blocked. In the experiment with a real robot, if portal 3 was blocked, it meant edge (11,g) would also be blocked. The RPP-based policies (Offline and Online) were able to learn these correlations and exploit them to decrease the cost to goal. They were also able to avoid routes with a large possibility of being blocked, such as edge (5,8) in the simulation.

This suggests that perhaps the savings from exploiting these correlations can be predicted and measured, given the correlation of edges and the distance saved from observing these indicators rather than only turning around when the path forward is blocked. However, the exact relationship is still unclear, and it would be beneficial to test the policies with more varied environments to better understand this. The difficulty in designing environments to test the potential of these policies is what motivates this part of the discussion. A different scenario, or even different probabilities for the obstacles may produce a different savings result. However, the design and construction of each scenario costs considerable effort, which is why we have only tested on two. Perhaps there is another scenario that better reflects the savings that the policies provide. We believe the Online policy may perform better in other environments, it's possible assuming that edges can be viewed from their endpoints just happened to be the better assumption for the simulation environment.

Although the submaps were all rectangular, this was intentional to keep the implementation simple. The only constraint is that the submaps should be as convex as possible for the observation scoring in the Online policy to be representative of the real value of an observation. Modeling the environment as a hybrid map combined with the map filter resulted in the robot remembering only major blockades, while relying on the path planner and path follower to navigate around smaller obstacles as it encounters them. It is important to note that the size of the submaps will affect the performance of the LAMP framework. Increasing the submap size can decrease the size of the resulting navigation graph and reduce the sensitivity of the learning to changes in the environment. But doing so would increase the robot's reliance on the path planning component to navigate, and it would increase the computation time of the edge observer. Conversely, decreasing the submap size can reduce the computation time of the edge observer to find a path between vertices, but it increases the robot's sensitivity to small changes in the environment, causing it to remember more super maps.

5.9 Portal Extension

As mentioned in Section 5.3.1, a weakness of our edge observation approach (direct inference) is that the midpoint of a portal is a vertex. If an obstacle blocks only the middle of the portal like in figure 5.19, direct inference will conclude the edge is blocked, yet the robot can clearly still move to the next submap.

A potential solution to this issue is to consider the length of the entire portal when checking for a path from one portal to another in the submap. However, to reduce the computation cost, Algorithm 6 is proposed to reduce the number of calls to the path finding algorithm. The data structure for the vertices needs modification to accommodate considering the full length of the portal instead of only the center. We let the vertex have

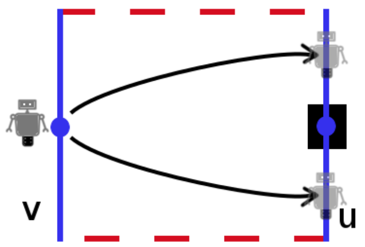


Figure 5.19: Example where direct inference will observe that edge (u,v) is blocked, but the robot can still pass through the portal on either side of the obstacle.

three attributes: name, portal, and position. The name is just an arbitrary label assigned to each vertex, the portal is a vector of cell locations of the portal in the full costmap, and the position is the coordinate we are using to represent the vertex for the path finder. Recall that the submap is just a portion of the full costmap, which is a type of occupancy grid. Therefore, the submap is an array of grid cells and the portals are a line from that submap. The vertex position can be modified depending on what portion of the portal is obstructed. It is initialized as the center of the portal at the beginning of every task execution. FindFreeSegments (Algorithm 7) finds the free segments of a portal and returns a list of cells, F, selected from each segment, prioritized by the following:

$$\text{priority} = \text{length of segment} - (\text{cell cost} + \text{distance from midpoint of portal}). \tag{5.11}$$

The cell cost is a number assigned to the cell by the costmap that usually depends on proximity to an obstacle, but it depends on the costmap configuration (see [48]). The cell with the highest priorities are checked first in lines 9 and 10 of Algorithm 6. Calculating the priority in this way dictates that the robot will prefer heading towards the middle of the portal, while balancing distance from obstacles and how much space there is for the robot to pass through. If the entire portal is blocked by an obstacle and the robot cannot pass, then line 7 will set the edge as blocked.

The advantage of the augmented direct inference is that instead of running the path finder for every combination of free cells in each portal, we only run it for each combination of free segments (lines 10 and 11). If we were to check for a path between each combination of cells in v and u until one was found, then the worst case is where both u and v are completely free, but there is no path between them. Suppose the portals for v and u had the same number of cells, n, then we have $O(n^2(O(\text{pathFinder})))$ for this worst case scenario. For the augmented direct inference, the theoretical worst case would be if every alternating cell was free, but there is no path between u and v, which will result in a time complexity of $O(\frac{n}{2}^2 O(\text{pathFinder})) + 2n)$. Although this does not appear to be much of an improvement, consider that the first worst case would happen frequently as we are expecting edges to be blocked, while the worst case for augmented direct inference is near impossible in practice. Even if there were obstacles that were perfectly aligned with the costmap, the robot would have to be smaller than the resolution of the costmap such that it can fit between those obstacles. In practice, the number of free segments would be several magnitudes lower than the number of cells in a portal.

Algorithm 6: AugmentedDirectInference

```
Input: $C_k$, $C_u$, $r_{\max}$, $\mathbf{x_R}$
1  $C_o = C_k \cup C_u$;
2  $S_{\text{curr}} = \text{currSubmap}(x_R)$;
3  for each $(u, v) \in E$ do
4      if $||\mathbf{x_R} - u.pos||_2 \leq r$ OR $||\mathbf{x_R} - v.pos||_2 \leq r$ OR $S(u, v) = S_{curr}$ then
5          $F_u = \text{FindFreeSegments}(u.portal)$;
6          $F_v = \text{FindFreeSegments}(v.portal)$;
7          if $F_v$ or $F_u$ is empty then
8              Set $(u, v)$ as blocked;
9          else
10             for each $v_F$ in $F_v$ do
11                 for each $u_F$ in $F_u$ do
12                     $v.pos = v_F$;
13                     $u.pos = u_F$;
14                     if there exists $P(u, v) \subseteq C_k(S(u, v))$ then
15                         Set $(u, v)$ as unblocked;
16                         Break, move to next edge in first for loop;
17                     else if there exists $P(u, v) \subseteq C_o(S(u, v))$ then
18                         Break, move to next edge in first for loop;
19             Set $(u, v)$ as blocked;
```

Algorithm 7: FindFreeSegments

```
Input: $\mathbf{p_v}$ vector of cells in the portal
Output: $F$ priority queue of selected cell in each free segment
1  $F = \emptyset$;
2  inSegment = False;
3  for each cell in $\mathbf{p_v}$ do
4      if inSegment = False AND cell is free then
5          start = cell;
6          inSegment = True;
7          priority = cell cost $+||c-$ mdpt$(\mathbf{p_v})||_2$;
8          chosenOne = cell;
9      else if inSegment = True AND cell is free then
10         priority = cell cost $+||c-$ mdpt$(\mathbf{p_v})||_2$;
11         if cellpriority < priority then
12             priority = cellpriority;
13             chosenOne = cell;
14     else if inSegment = True AND cell is occupied then
15         end = previous cell;
16         priority = $||end - start||_2$ - priority;
17         Add chosenOne to $F$ in order of priority;
18         inSegment = False;
```

5.10 Summary

This chapter first introduced and justified using a hybrid map as the environment model for real robots to plan with when utilizing past experience to improve navigation. We then proposed the direct and indirect inference mechanisms to resolve edges on the hybrid map so the robot can navigate to the goal using the hybrid map. We reviewed how the Optimistic and Offline RPP policies were constructed before introducing the Online RPP policy and its construction method. Finally, we put all the pieces together into the LAMP framework which can be easily integrated with existing navigation stacks to allow the robot to learn and improve its navigation from its past experiences navigating in the same uncertain environment. The potential of the LAMP framework to improve navigation was supported by our experiment results, which showed that with the added support to construct better policies, navigation stacks with the LAMP framework can reduce the average travel cost over time, even with imperfect localization and observations. Lastly, we proposed an extension to direct inference to more accurately resolve edges if obstacles partially obstruct the center of a portal.

Chapter 6

Conclusion

The goal of this thesis was to investigate the possibility of implementing a smarter navigation system to improve navigation in environments with structured uncertainties over time, that is, learning to predict these uncertainties and plan accordingly. In the first half, we presented the Learned Reactive Planning Problem (LRPP) which captured this problem in a theoretical manner. We then proposed a map filter algorithm to capture and store information, or super maps, about the environment after a completed task execution. We exploit these super maps to construct a policy using the RPP algorithm that minimizes the expected travel distance for a given navigation task over time. This policy is updated after every task execution. Furthermore, our simulation results verified that as the robot accumulates data about its past task executions, following the constructed policy reduced the average travel distance of the navigation task by around 30%. However, the experiments were done in an ideal world with perfect localization, control, and observations.

In the second half of this thesis, we consider using the same solution in a more realistic setting, where occupancy grids of the same area are noisy and the robot has imperfect localization. We proposed the LAMP framework which has two parts, an edge observer and a high level planner. Both use a hybrid map to model the environment. The edge observer uses direct inference to resolve edge states in the current task execution. The high level planner has several functions. First, it uses the results from the edge observer to update its map of the environment and at the end of a task execution it updates its set of super maps using the map filter from the first half of the thesis. Second, it uses indirect inference (which takes the results from the edge observer and past task executions) to form a belief about the environment. Third, it constructs a policy based on its set of super maps and guides the robot during task execution with the policy. We also introduce an algorithm that computes the policy during task execution which we call the Online policy.

We show how the LAMP framework can be integrated with existing navigation stacks, and present results from simulations and experiments on a real robot. Using the LAMP framework, we showed that following the Offline RPP and Online RPP policies reduced the expected travel distance by 20-30% as it collected more experience compared to just using the default ROS solution. In conclusion, the LAMP framework is a promising approach for smarter navigation systems that learn how to better navigate an environment the robot has experienced many times.

6.1 Future Work

Although the results were positive, this section highlights several limitations which present themselves as future directions of research for improving the algorithms and frameworks proposed in this thesis.

6.1.1 RPP Based Policies

While this thesis focused on utilizing the RPP algorithm to solve the LRPP problem in idealistic and realistic scenarios, the LRPP is closely related to the stochastic CTP, which has multiple solutions in the literature [1, 15]. It would be interesting to compare the performance of the RPP policy with the policies from these other solutions on various CTP instances. Furthermore, their performances in real world settings can now be tested by implementing these policies in the high level planner of the LAMP framework, using the same edge observer and mapping approach.

A challenge that was encountered while designing the experiments for this thesis was how to benchmark the performance of the policies we proposed. Our work and several others [45, 57] claimed that our policies perform better than the optimistic policy if the probabilities of edges being blocked were known or learned, but each author used different environments that may favor their own policy. Moreover, the benefits of using these policies over the other stochastic CTP policies depend on state correlations existing between edges instead of assuming they are independent. This raises several questions for the research community: How correlated are the environments that robots operate in? Do the savings outweigh the added complexity to implement such navigation policies?

Additionally, observing edges to draw conclusions about the rest of the environment has limited usefulness. A major improvement to the current solution would be to draw conclusions about the environment based on more general observations, such as identifying signs, specific objects, or even sounds that could indicate edge traversability. A difficulty that would arise is how to filter and remember only potentially useful information, similar to what we did in section 5.2 (i.e. compressing cost maps into edges states). The difference is that in our work, the robot knows that it needs to remember the edge states, yet not all of these edge states will aid the policy construction. Whereas in a general observation scheme, what constitutes as ‘potentially useful’ information and therefore should be remembered, is not so clear.

6.1.2 Edge Observer

As we mentioned in chapter 5, little work has been done to address the problem of resolving edges given a topological map (i.e. a graph) when we no longer assume that edges are unblocked. Our edge observer that used direct inference was functional, but it was still fairly sensitive to errors in the cost map. We introduced a theoretical extension to direct inference, called the augmented direct inference, to consider the entire length of the portals when searching for a possible path between two portals, instead of just the center pixel. Experiments should be done with this extension implemented to verify that it does improve the accuracy of the edge observer without adding significant computation time.

Another improvement to the edge observer is to measure its confidence in a reported edge state. Obstacle information in the costmap can be cleared by mistake if the robot is at an acute angle to the surface of the obstacle. This angle could be calculated and factored into the confidence measurement. Alternatively, the high level planner could consider multiple measurements of the edge state before setting the state in its current map.

6.1.3 LAMP Framework

Although the Online policy performed worse than the Offline policy in our experiments, it may have been due to inaccuracies in the reported edge state from the edge observer or the structure of the occupancy grid of the test environment. The performance of the Online policy needs further investigation by testing it with different environments, including ones that are intentionally designed to attempt to showcase its advantages over the Offline policy.

Furthermore, while the LAMP framework is currently implemented to run with the ROS navigation stack, it would be beneficial to test it with a simpler simulator to decouple the strengths and weaknesses of the policies from shortcomings in the navigation stack. We already noted that there can be errors in the resulting costmap, but some other issues that arose were the robot getting stuck because of poor choices made by the path follower algorithm, and the sheer amount of time required to collect data. The vast majority of the runtime of the experiments was spent on the robot moving from one vertex to the next. Running only one trial with 100 task executions in a 20 m x 20 m environment took close to 12 hours to complete. To bypass the need for localization and path following, the LAMP framework could be tested on large, game-like grid environments, similar to the simulation in chapter 4, where movement is constrained to 8 directions. Aside from reducing the time to execute policies, using these environments may help to better understand the performance of the Online policy and it would allow for testing on environments that do not guarantee a path to the goal, without including the possibility of the cost map incorrectly indicating that there is no path to goal because a route is blocked due to localization error.

Lastly, to improve the usability of the LAMP framework, we could add a component for automatically decomposing maps and testing the performance of the policies with submaps generated by existing 2D map decomposition algorithms [46, 70]. Quantifying the relationship between the submap size, the number of super maps recorded, and the computation time of direct inference may be of interest to the robotics community.

References

[1] Vural Aksakalli, O. Furkan Sahin, and Ibrahim Ari. An AO* based exact algorithm for the Canadian traveler problem. *INFORMS Journal on Computing*, 28(1):96–111, 2016.

[2] Y. Alnounou, M.J. Paulik, M. Krishnan, G. Hudas, and J. Overholt. Occupancy Grid Map Merging using Feature Maps. In *IASTED Technology Conference on Robotics and Applications*, 2010.

[3] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[4] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008.

[5] Jonathan Binney and Gaurav S. Sukhatme. Branch and bound for informative path planning. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2147–2154, 2012.

[6] Andreas Birk and Stefano Carpin. Merging occupancy grids from Multiple Robots. *Proceedings of the IEEE*, 94(7), 2006.

[7] F. Blochliger, M. Fehr, M. Dymczyk, T. Schneider, and R. Siegwart. Topomap: Topological mapping and navigation based on visual slam maps. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9, May 2018.

[8] Zahy Bnaya, Ariel Felner, and Solomon Eyal Shimony. Canadian traveler problem with remote sensing. *IJCAI International Joint Conference on Artificial Intelligence*, pages 437–442, 2009.

[9] Gino Brunner, Oliver Richter, Yuyi Wang, and Roger Wattenhofer. Teaching a Machine to Read Maps with Deep Reinforcement Learning. *32nd AAAI Conference on Artificial Intelligence*, abs/1711.07479, 2017.

[10] Adam Bry and Nicholas Roy. Rapidly-exploring random belief trees for motion planning under uncertainty. In *2011 IEEE international conference on robotics and automation*, pages 723–730. IEEE, 2011.

[11] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[12] Didier Devaurs, Thierry Siméon, and Juan Cortés. Parallelizing rrt on distributed-memory architectures. In *2011 IEEE International Conference on Robotics and Automation*, pages 2261–2266. IEEE, 2011.

[13] Fox Dieter, Sebastian Thrun, and Wolfram Burgard. *Probabilistic Robotics.* MIT Press, 2005.

[14] Fox Dieter, Burgard Wolfram, and Thrun Sebastian. The Dynamic Window Approach to Collision Avoidance. pages 137–146, 1997.

[15] Patrick Eyerich, Thomas Keller, and Malte Helmert. High-quality policies for the Canadian traveler's problem (extended abstract). *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SoCS 2010*, pages 147–148, 2010.

[16] Mark Fiala. Artag, a fiducial marker system using digital techniques. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 590–596. IEEE, 2005.

[17] Dror Fried, Solomon Eyal Shimony, Amit Benbassat, and Cenny Wenner. Complexity of Canadian traveler problem variants. *Theoretical Computer Science*, 487:1–16, 2013.

[18] Paul Furgale and Timothy D. Barfoot. Visual Teach and Repeat for Long-Range Rover Autonomy. *Journal of Field Robotics*, 27(5):534–560, 2010.

[19] Roland Geraerts and Mark H. Overmars. A comparative study of probabilistic roadmap planners. In *Springer Tracts in Advanced Robotics*, volume 7, pages 43–57. 2004.

[20] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *Journal of Experimental Algorithmics*, 13:2.2, 2009.

[21] Hengwei Guo and Timothy D Barfoot. The Robust Canadian Traveler Problem Applied to Robot Routing. *IEEE International Conference on Robotics and Automation*, pages 5523–5529, 2019.

[22] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-Time Loop Closure in 2D LIDAR SLAM. In *IEEE Conference on Robotics and Automation*, pages 1271–1278, 2016.

[23] Goran Huskić, Sebastian Buck, and Andreas Zell. GeRoNa: Generic Robot Navigation: A Modular Framework for Robot Navigation and Control. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 95(2):419–442, 2019.

[24] Daisuke Kakuma, Satoki Tsuichihara, Gustavo Alfonso Garcia Ricardez, Jun Takamatsu, and Tsukasa Ogasawara. Alignment of Occupancy Grid and Floor Maps Using Graph Matching. *Proceedings - IEEE 11th International Conference on Semantic Computing, ICSC 2017*, pages 57–60, 2017.

[25] Mrinal Kalakrishnan, Peter Pastor, Ludovic Righetti, and Stefan Schaal. Learning objective functions for manipulation. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1331–1336. IEEE, 2013.

[26] A. Kanezaki, J. Nitta, and Y. Sasaki. Goselo: Goal-directed obstacle and self-location map for robot navigation using reactive neural networks. *IEEE Robotics and Automation Letters*, 3(2):696–703, April 2018.

[27] Richard M. Karp. Reducibility Among Combinatorial Problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[28] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[29] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[30] Sven Koenig and Maxim Likhachev. D*Lite. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 15, pages 476–483, 2002.

[31] Sven Koenig and Maxim Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005.

[32] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[33] Amit Konar, Indrani Goswami Chakraborty, Sapam Jitu Singh, Lakhmi C Jain, and Atulya K Nagar. A deterministic improved q-learning for path planning of a mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(5):1141–1153, 2013.

[34] Kurt Konolige, Eitan Marder-Eppstein, and Bhaskara Marthi. Navigation in hybrid metric-topological maps. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3041–3047, 2011.

[35] Ioannis Kostavelis and Antonios Gasteratos. Semantic mapping for mobile robotics tasks: A survey. *Robotics and Autonomous Systems*, 66:86–103, 2015.

[36] Henrik Kretzschmar, Markus Spies, Christoph Sprunk, and Wolfram Burgard. Socially compliant mobile robot navigation via inverse reinforcement learning. *The International Journal of Robotics Research*, 35(11):1289–1307, 2016.

[37] Tomasz Kucner, Jari Saarinen, Martin Magnusson, and Achim J Lilienthal. Conditional transition maps: Learning motion patterns in dynamic environments. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1196–1201. IEEE, 2013.

[38] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001. IEEE, 2000.

[39] Steven M LaValle. *Planning algorithms.* Cambridge university press, 2006.

[40] Steven M LaValle and James J Kuffner Jr. Rapidly-exploring random trees: Progress and prospects. 2000.

[41] T. Lei and L. Ming. A robot exploration strategy based on Q-learning network. In *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 57–62, June 2016.

[42] Wenqi Li, Dehua Chen, and Jiajin Le. Robot patrol path planning based on combined deep reinforcement learning. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 659–666. IEEE, 2018.

[43] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime Dynamic A *: An Anytime , Replanning Algorithm. *Science*, pages 262–271, 2005.

[44] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[45] Zhan Wei Lim, David Hsu, and Wee Sun Lee. Shortest path under uncertainty: Exploration versus exploitation. In *Uncertainty in Artificial Intelligence - Proceedings of the 33rd Conference, UAI 2017*, 2017.

[46] Ming Liu, Francis Colas, Luc Oth, and Roland Siegwart. Incremental topological segmentation for semi-structured environments using discretized GVG. *Autonomous Robots*, 38(2):143–160, 2014.

[47] David G Lowe et al. Object recognition from local scale-invariant features. In *iccv*, volume 99, pages 1150–1157, 1999.

[48] David V. Lu, Dave Hershberger, and William D. Smart. Layered costmaps for context-sensitive navigation. *IEEE International Conference on Intelligent Robots and Systems*, pages 709–715, 2014.

[49] Ryan A MacDonald and Stephen L Smith. Active sensing for motion planning in uncertain environments via mutual information policies. *The International Journal of Robotics Research*, 38(2-3):146–161, 2019.

[50] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 300–307, 2010.

[51] Patrycja E. Missiuro and Nicholas Roy. Adapting probabilistic roadmaps to handle uncertain maps. *Proceedings - IEEE International Conference on Robotics and Automation*, 2006(May):1261–1267, 2006.

[52] Nikos C Mitsou and Costas S Tzafestas. Temporal occupancy grid for mobile robot dynamic environment mapping. In *Control & Automation, 2007. MED'07. Mediterranean Conference on*, pages 1–8. IEEE, 2007.

[53] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[54] Michael Montemerlo and Sebastian Thrun. Simultaneous localization and mapping with unknown data association using fastslam. In *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, volume 2, pages 1985–1991. IEEE, 2003.

[55] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015.

[56] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[57] Lorenzo Nardi and Cyrill Stachniss. Long-term robot navigation in indoor environments estimating patterns in traversability changes. *arXiv preprint arXiv:1909.12733*, 2019.

[58] Gergely Neu and Csaba Szepesvári. Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 295–302, 2007.

[59] K. J. Obermeyer and Contributors. VisiLibity: A c++ library for visibility computations in planar polygonal environments. `http://www.VisiLibity.org`, 2008. R-1.

[60] Stefan Obwald, Maren Bennewitz, Wolfram Burgard, and Cyrill Stachniss. Speeding-Up Robot Exploration by Exploiting Background Information. *IEEE Robotics and Automation Letters*, 1(2):716–723, 2016.

[61] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407. IEEE, 2011.

[62] Christos H Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.

[63] Jung-Jun Park, Ji-Hun Kim, and Jae-Bok Song. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems*, 5(6):674–680, 2007.

[64] James Parker, Alessandro Farinelli, and Maria Gini. Lazy max-sum for allocation of tasks with growing costs. *Robotics and Autonomous Systems*, 110:44–56, 2018.

[65] Mark Pfeiffer, Michael Schaeuble, Juan Nieto, Roland Siegwart, and Cesar Cadena. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1527–1533, 2017.

[66] Mihail Pivtoraiko and Alonzo Kelly. Efficient constrained path planning via search in state lattices. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, pages 1–7, 2005.

[67] J. Puzicha, J. M. Buhmann, Y. Rubner, and C. Tomasi. Empirical evaluation of dissimilarity measures for color and texture. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1165–1172 vol.2, Sep. 1999.

[68] Brent Schlotfeldt, Vasileios Tzoumas, Dinesh Thakur, and George J Pappas. Resilient Active Information Gathering with Mobile Robots. *arXiv preprint arXiv:1803.09730*, 2018.

[69] Anderson Souza and Luiz M. G. Goncalves. Occupancy-elevation grid: an alternative approach for robotic mapping and navigation. *Robotica*, 34:2592–2609, 2016.

[70] Sebastian Thrun. Artificial Intelligence Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[71] Florence Tsang, Ryan A. Macdonald, and Stephen L. Smith. Learning Motion Planning Policies in Uncertain Environments through Repeated Task Executions. *IEEE Conference on Robotics and Automation*, pages 8–14, 2019. ©2019 IEEE. Reprinted, with permission from authors.

[72] Sundar Vishwanathan. Randomized Online Graph Coloring. *Journal of Algorithms*, 669:464–469, 1992.

[73] K M Wurm, a Hornung, M Bennewitz, C Stachniss, and W Burgard. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. *Proc of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, 16(3):403–412, 2010.

[74] Delong Zhu, Tingguang Li, Danny Ho, Chaoqun Wang, and Max Q Meng. Deep Reinforcement Learning Supervised Autonomous Exploration in Office Environments. *Proceedings of the 2018 IEEE Conference on Robotics and Automation (ICRA)*, pages 7548–7555, 2018.

[75] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3357–3364, May 2017.