Reasoning over Hierarchical Abstractions for Long-Horizon Planning in Robotics

by

Christopher P. Bradley

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN AERONAUTICS AND ASTRONAUTICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

Authored by: Christopher P. Bradley
Department of Aeronautics and Astronautics
December 17, 2024

Reasoning over Hierarchical Abstractions for Long-Horizon Planning in Robotics

by

Christopher P. Bradley

Submitted to the Department of Aeronautics and Astronautics
on December 17, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN AERONAUTICS AND ASTRONAUTICS

ABSTRACT

We aim to enable robots to act intelligently in complex environments not explicitly designed around them. In order to do so, robots can simplify decision making by forming hierarchical abstractions of their world, and planning within those representations. However, in reality, the types of abstractions robots are able to build are often poorly aligned with the planning problems they must solve, which limits how useful those abstractions can be in efficient decision making. For example, autonomous agents struggle in many real world scenarios, particularly when their environments are large, cluttered with obstructions, or beset by uncertainty. These factors often imply that decisions made at higher levels of abstraction may not be easily refined to low level plans, leading to backtracking during either search or execution. In this thesis, we consider contributions which improve the efficiency and quality of long-horizon hierarchical planning in robotics. Specifically, we propose approaches which explicitly reason about the imperfections of the abstractions available to robots during planning, and show how those methods can improve performance on a variety of tasks and environments.

There are three primary settings for which we make contributions in this thesis. First, we will consider the problem of solving tasks in partially revealed environments, wherein our abstract plans cannot be known to be feasible until we attempt execution because the world is not fully known at planning time. To solve this problem, we first develop a high level planning representation which recognizes that actions that enter unknown space can either succeed or fail with some probability. The first contribution of this work is then to learn to predict the feasibility and cost of actions within that abstraction from visual input. We also describe a method for planning which uses these predictions, and we are able to show that our approach can generate plans that are significantly faster at completing tasks in unknown environments experimentally when compared with heuristic driven baselines. Next, we will discuss work in Task and Motion Planning (TAMP), where the world is fully known, but the problems require complex interaction with the environment to the point that we must intelligently guide search in order to find plans efficiently. We build upon our work in the first setting by once again learning to predict the outcome and cost of different sub-tasks within a TAMP abstraction. We further contribute a novel method to guide search in this setting for plans which minimize cost given our learned predictions, and demonstrate the ability to find faster plans than established TAMP approaches both in simulation, and on real world robots. In our final problem setting, we consider attempting to solve TAMP problems in real world, large-scale environments. To do this, we define an approach for constructing tractable planning abstractions from real perception using hierarchical scene graphs, ensuring that when we refine our abstract plans within these representations, the low-level trajectories still satisfy the given task's constraints. A major contribution of this work is an approach for planning efficiently in these domains by pruning provably superfluous information from the world model. The unifying aim of the work in this thesis is to develop approaches which enable robots to solve complex tasks in large-scale, real world environments without human intervention. To that end, across all contributions, we demonstrate experimentally on real robots the importance of accounting for imperfections in hierarchical abstraction during planning.

Thesis supervisor: Nicholas Roy
Title: Professor of Aeronautics and Astronautics

Acknowledgments

The work in this thesis, and my completion of the PhD program at MIT, would not have been possible without the help of many others. This section will attempt to highlight some of those contributions. However, if were to accurately acknowledge, or even just mention, everyone who has had a hand in my development as a researcher and roboticist, I would need another 200 pages.

First, I would like to thank the members of my committee for their guidance over the past few years. Luca Carlone, Pulkit Agrawal, and George Konidaris each provided a different and invaluable perspective on my research, and I am grateful for their insights. Most notably, I would like to thank Nick Roy, my advisor and chair of my committee. When I was accepted to graduate school at MIT back in 2017, I was relatively new to robotics as I only started taking related courses in my junior year of undergrad. Most the research I had done was related to fluid mechanics (a brief thank you to Guillaume Blanquart for giving me my first opportunity to join a research lab as a freshman at Caltech). Despite my inexperience, Nick welcomed me to the Robust Robotics Group (RRG) with open arms (after confirming that I would answer "yes" to the question "are you good at math"). Since that time, Nick has contributed to my growth as a researcher in countless ways. From personally helping to guide my research direction, to pointing the way to interesting collaborations (both in and out of MIT), to connecting me with opportunities in industry, Nick has ensured that I had every chance to learn from the best at all stages of my PhD. For a small example of Nick's commitment, few advisors would take the time to meet with their junior students in person to prepare them for their qualifying exams on multiple occasions, yet Nick did exactly that, and much more.

Possibly the most significant impact Nick made on my time in the lab was his hand in forming the social and collaborative culture of the Robust Robotics Group. I am extremely fortunate to have been surrounded by people as brilliant and kind as those who I am able to call my lab-mates. In the RRG, I have shared a space with some of the smartest people I have ever met, and the chance to learn from them has been invaluable. Not only is our lab filled with excellent roboticists, but the RRG is also well known as a uniquely fun place to spend a PhD, especially when we have gotten the opportunity to travel to conferences together. Most importantly, our lab is extremely open with our research, and this collaborative nature is a major factor in my growth as a roboticist.

In particular, I would be remiss if I did not highlight the contributions of Greg Stein, both as a collaborator and as a mentor. Officially, Greg and I published three papers together over four years, however over that time I learned more from him about how to conduct research as a roboticist than could be contained in those publications. Greg taught me how identify interesting research directions, how to refine those directions in the face of disappointing experimental results, how to tactfully respond to unreasonable reviewers, and how to effectively present my ideas, both on paper and verbally. Greg unsurprisingly continued on to become a professor after he graduated, and I am lucky to have been his first "student." I would also like to thank a few other collaborators by name. Two professors I've had the good fortune of working with are Hadas Kress-Gazit at Cornell and Luca Carlone at MIT. Victoria Preston, Sebastian Castro, and Adam Pacheck, also all contributed in some part to the development and writing of different publications I've been a part of. Finally, Aaron Ray in particular has been a huge contributor of the work that makes up the final chapter of this thesis, and I look forward to continuing our collaboration.

Of course, I could not survive the challenges and stresses of MIT without the support of my friends (outside of lab). Thank you to my grad student cohort: Mike, Brad, Liam, Colin, Jess, Cadance, Patrick, et al. for going through this journey with me. Thanks to my friends at CABLE: Kris, Sasha, Albert, Jaffe, Sid, Shelly, Nick, and beyond for putting up with me even after I outlived my usefulness at IM waterpolo. And finally my people from Caltech: Harrison, Parth, Nick, Mojo, Sandia and many more for staying in touch over these years. It is in no small part because of these folks that I've genuinely enjoyed my time as a graduate student, which is more than most can say.

Most importantly, I would like to thank my family, my parents and three sisters, for their love and inspiration. Michelle made it all but impossible to give up during my PhD, how could I quit where she succeeded? Victoria has consistently been a much needed source of healthy, good natured debate from across the country to help take my mind off my work. And Cassandra, nearly four years my junior, almost earned her degree before I did, and may have been the push I needed to reach the finish line. Finally, I thank my parents. In so many different ways I could not be where I am without them. I will be forever grateful for all you have sacrificed for me.

Contents

Title page **1**

Abstract **3**

Acknowledgments **5**

List of Figures **13**

List of Tables **15**

1 Introduction **17**

1.1 Reasoning Abstractly 19

1.2 Making Decisions Within an Abstraction: Plans vs. Policies 21

1.3 Hierarchical Planning and Downward Refinement 23

1.4 Contributions 26

2 Background and Related Work **31**

2.1 Hierarchical Planning 31

2.1.1 Top-Down Planning 33

2.1.2 Task And Motion Planning 35

2.1.3 Learning to guide TAMP 37

2.2 Building Hierarchical Planning Abstractions 38

2.2.1 Metric Representations 39

2.2.2 Hierarchical Mapping 40

2.2.3 Planning in Large Scale Environments Using Scene Graphs 42

2.2.4 Deriving Abstractions over Actions . . . 43
2.3 Planning in Partially Revealed Environments . . . 43
2.3.1 Modeling Planning Under Uncertainty as a Partially Observable Markov Decision Process (POMDP) . . . 44
2.3.2 Solving POMDPs Exactly and Approximately . . . 47
2.3.3 Navigation In Partially Revealed Environments . . . 48
2.3.4 Solving Complex Tasks Under Uncertainty Using Temporal Logic . . 49

3 Hierarchical Decision Making in the Presence of Uncertainty 53
3.1 Planning over Subgoals for Complex Tasks . . . 57
3.1.1 Defining our Problem . . . 58
3.1.2 Defining an Abstraction over Actions and Outcomes . . . 63
3.1.3 Estimating Future Cost using High-Level Actions . . . 66
3.1.4 Planning with High-Level Actions using PO-UCT . . . 70
3.2 Learning Transition Probabilities and Costs . . . 73
3.2.1 Encoding a Transition . . . 73
3.2.2 Network Architecture and Training . . . 74
3.2.3 Collecting Training Data . . . 76
3.3 Experiments . . . 77
3.3.1 Firefighting Scenario Results . . . 77
3.3.2 Delivery Scenario Results in Simulation . . . 79
3.3.3 Delivery Scenario Results in the Real World . . . 80
3.4 Discussion . . . 81

4 Learning Feasibility and Cost to Guide Task and Motion Planning 87
4.1 An Abstraction for Task and Motion Planning . . . 92
4.2 Reformulating the Task and Motion Planning Problem . . . 95
4.3 Computing the Cost of Solving and Executing an Action Plan . . . 99
4.4 Learning to Model External Functions . . . 101
4.5 Planning with our Models . . . 104
4.5.1 Searching for Abstract Stream Plans . . . 105

4.5.2 Searching within Stream Plans 105
4.5.3 Guiding Search with Learned Cost 106
4.5.4 Progressive Widening with Learned Cost 107
4.6 Experimental Results 109
4.6.1 Kitchen Domain 109
4.6.2 Unpack Domain 111
4.6.3 Real World Experiments 112
4.7 Discussion 114

5 Building and Planning within Large-Scale Hierarchical Abstractions 119
5.1 Task and Motion Planning in 3D Scene Graphs 123
5.1.1 Task and Motion Planning Preliminaries 123
5.1.2 Building 3D Scene Graphs from Perception 125
5.1.3 Inferring the Planning Domain from Scene Graphs 126
5.2 Scalable Scene Graph Planning 129
5.2.1 Hierarchical Planning 130
5.2.2 Removing Redundant Symbols 133
5.2.3 Execution Consistency 135
5.2.4 Ignoring Irrelevant Objects 139
5.3 Evaluation 142
5.3.1 Scene Graph Size 142
5.3.2 Goal Complexity 143
5.3.3 Object Obstruction 144
5.3.4 Real-World Manipulation 146
5.4 Discussion 147

6 Conclusion 153
6.1 Incorporating Foundation Models into Planning 155
6.2 Ongoing Work Augmenting Hierarchical Planning with Language 156

A PDDL Domain Encodings **159**

A.1 Unpack Domain 159

A.1.1 Unpack Domain File 159

A.1.2 Unpack Stream File 162

A.1.3 Kitchen Domain File 164

A.2 Kitchen Domain 164

A.2.1 Kitchen Stream File 167

A.3 Relaxed Encoding of Inspection Domain 168

A.3.1 Relaxed Domain File 168

A.3.2 Relaxed Stream File 173

B Deferred Proofs **177**

B.1 Proof of Prop 1 177

B.2 Proof of Prop 2 177

References **179**

List of Figures

1.1 Building abstract planning representations 21
1.2 Downward refinement in robotics tasks 25

3.1 Overview of POTLP 56
3.2 Forming product automata from a task specification and labeled transition system 60
3.3 Possible orientations of unknown space 63
3.4 Enumerating high-level actions 66
3.5 Visualization of a POTLP search tree 70
3.6 Neural network architecture 73
3.7 POTLP training data generation process 76
3.8 Fire-fighting experimental results 77
3.9 Visualization of Delivery scenario in simulation 80
3.10 Real world delivery experiment 81

4.1 Visualization of Panda solving a manipulating problem 91
4.2 Example of sub-problem tree of two different high-level plans 98
4.3 Network Architecture using GNN 102
4.4 Planning and learning to compute cost 104
4.5 Kitchen domain experimental results 110
4.6 Unpack domain experimental results 112
4.7 Real world Panda manipulation examples 113

5.1 Illustration of how we derive and encode tasks from scene graphs 121

5.2 Examples of hydra scene graphs at different levels of abstraction 125
5.3 Planning in a direct, dense encoding of a planning problem derived from a Hydra scene graph 129
5.4 Planning in a Hydra scene graph using our hierarchical abstraction (without pruning) 131
5.5 Pruning our high level planning domain 136
5.6 Our middle level planning abstraction 137
5.7 Visualization of our tri-level planning abstraction, and grounded plan output 139
5.8 Evaluation environments for planning in scene graphs 143
5.9 Experimental results for our planning approach 144
5.10 A scene graph built from KITTI, highlighting our incremental object adding approach 145
5.11 Images captured during our real world Spot experiments in building 45 at MIT 147

6.1 Hydra scene graph built in a large, outdoor environment 157

List of Tables

3.1 POTLP Firefighting Results 79

4.1 TAMP Experimental Results 109

Chapter 1

Introduction

The field of robotics research has made significant progress toward the goal of producing autonomous agents which can move within and influence the world around them. Automation in manufacturing has spurred massive gains in efficiency in our ability to produce products as varied as automotive parts and consumer electronics. Beyond the factory, robotics has enhanced our ability to explore our universe, from the bottom of Earth's oceans to the surfaces of other planets. We use robots to help doctors conduct complex surgeries, search and rescue teams find people lost in the wilderness, and large companies move billions of dollars of products around massive warehouses. Our roads are now populated by self-driving cars, improving the safety and efficiency of travel for millions of people.

There are, however, limitations to what modern robotics is currently capable of, and we can see this in the common characteristics of commercial robots. For the most part, a robot operating in the real world (i.e., not in a research lab) falls into one of three categories: 1) the robot is constrained to act in an environment carefully tailored to its design; 2) it is tasked with solving a relatively specific, well-scoped task; or 3) a human is responsible for its remote operation. Stated differently, if a robot is capable of making decisions for itself, either the robot's environment is simple and well understood, or it is specialized to one specific problem. Often both are true. These restricted operating conditions are necessary for a number of reasons, but in general, as the variability of a robot's environment and the complexity of its assigned task increase, the more difficult it becomes for an autonomous agent to make and execute decisions. As a result, when we want to use robotics to solve problems in the real world, we often either need to take great care to control the environment, make sure the problem is simple (and ideally repetitive), or make the decisions for the robots ourselves.

There are many cases where these limitations are reasonable, and still enable real-world robots to provide value. There are classes of problems where robust, repetitive motion is all that is needed (like manufacturing), and areas where careful supervision is required (like surgery). There may always be domains where we want a *human in the loop* to aid in decision making, not because the robot is incapable of reasoning, but because the human may want to express particular desires. However, the reality expressed in the previous paragraph is obviously quite limiting in the types of tasks we can solve with robots. Some problems can be constrained to tailored environments or simple tasks, but many more can not. Furthermore, there are many instances where it is undesirable, or impossible, for a human to control a robot directly. For tasks like cooking or cleaning, arguably the whole goal of having a robot do these things is to free the human to spend their time on more intellectually stimulating, or otherwise enjoyable activities[1]. If a human is required to operate or oversee the robot to complete these tasks, we have failed to accomplish the primary purpose of using a robot in the first place. Similarly, there are situations, like an underwater rescue mission, where communication with a robot is infeasible, and the agent must be fully autonomous to accomplish its goal. Finally, there are tasks that we would hope a robot could perform better than any human ever could, like working in a mine, putting out a fire, or constructing a building.

Imagine, for example, a robot tasked to make coffee in an unfamiliar building. This is a common example in robotics research (popularized in part by Leslie Pack Kaelbling), and is difficult for an autonomous agent for a number of reasons. First, the robot must build a representation of its environment from different sources of perception. This model of the world must be accurate, while at the same time be useful for decision making. Second, it must use that representation to reason about how it should explore its environment in order to find the ingredients required for making coffee. Once it has found these ingredients, the agent needs to solve the multi-modal problem inherent in manipulating multiple objects in a

[1]Some would argue that cooking itself is enjoyable, but that is outside the scope of this thesis.

potentially cluttered environment. Finally, it must do all of this quickly, both in terms of time spent thinking and time spent acting, or risk becoming obsolete as a human decides to make coffee themselves. This motivating task combines elements of reasoning in the presence of uncertainty, executing complex manipulation plans, and acting in real-world environments. The goal of the work in this thesis is to develop approaches which enable robots to solve complex tasks in large-scale, real world environments without human intervention.

1.1 Reasoning Abstractly

Often times, when we are considering how we might improve how a robot behaves in certain scenarios, it is instructive to imagine how we as humans think and act in the same settings. As a human, when faced with the task described above, it is intuitive to break down the problem into a set of possible actions, which can reduce the space of everything one could do into a few discrete options. In this case, we would expect that a human would reasonably conclude that to make coffee, they should first travel to the kitchen. Upon arrival, they might then look in different cupboards in order to find coffee beans, considering actions like "open the cabinet" or "pick up an object", and so on.

Making discrete decisions in this type of problem relies on the idea of an "Abstraction", and this comes naturally to humans. We can define an abstraction as a representation of reality which does not contain all possible information about the world, but instead trades some of that information for a simpler structure that is efficient for planning and is—hopefully—still a useful model of the real problem. Abstractions exist hierarchically, and for any domain there are low levels of abstraction which contain more fine-grained information, and higher levels which discard some of that information. In our coffee making example, reasoning at a low-level of abstraction might require thinking in terms of placing one foot in front of the other, or even considering the continuous path through space our body takes, and the muscle movements required to follow that trajectory. Thinking exclusively at a low level presents a challenging problem for a robot, particularly as both the complexity of the task and scale of the environment grow. For a high degree-of-freedom robot, finding a fine-grained trajectory which takes the agent from its current position to one where it has done everything needed to make coffee is a very long-horizon problem, with a very high branching factor. In other words, it has to make many decisions, each of which has many (possibly infinite) options. These challenges compound when the world is partially observed. Attempting to produce a low-level trajectory through unseen space requires reasoning about every possible configuration of the world, which is generally intractable.

Alternatively, if we were to consider the walk to the kitchen as one distinct action, as opposed to a chain of many, we can greatly reduce the number of decisions needed to solve the full problem. Under the assumption that our robot has the ability (either learned or programmed) to navigate hallways safely, how exactly we plan to reach the kitchen is not critical to the feasibility and cost of our overall plan. Moreover, by thinking in terms of high level decisions first, we can avoid having to consider finding a low-level plan in unknown space until it has been observed. Humans often reason at this high level, only refining our abstract decisions to full trajectories when we execute a plan, thus simplifying our decision making.

Reasoning at a high level of abstraction in this way has a long history in artificial intelligence research. As an example, let us consider the classic AI problem of a robot playing chess. To play chess in the real world, a robot needs some mechanism of observing the board state, picking up game pieces, and placing them in legal positions. However, given a robot which can reliably do these things, it is a waste of time to consider the act of physically moving pieces around the board when deciding what moves to make. If we want to design a chess playing agent, it is more efficient to describe the world as an 8x8 grid, with 32 total pieces and known dynamics (see Figure 1.1) [1]. This clearly is a useful abstraction for chess as it is simple, and perfectly describes the relevant components of the problem. Great progress has been made in AI in these exact types of scenarios (e.g., the Alpha Zero line of work from Google Deepmind). Modern solutions to decision making problems like chess generally involve collecting a large amount of data in simulation, then using a few different models trained on this data to guide decision making. Specifically, Alpha-Zero uses experience from self-play to learn both a value function over board states, as well as a policy for the optimal action in a given state. The agent can then use these learned functions to guide search within the chess planning abstraction, leading to very compelling results [2].

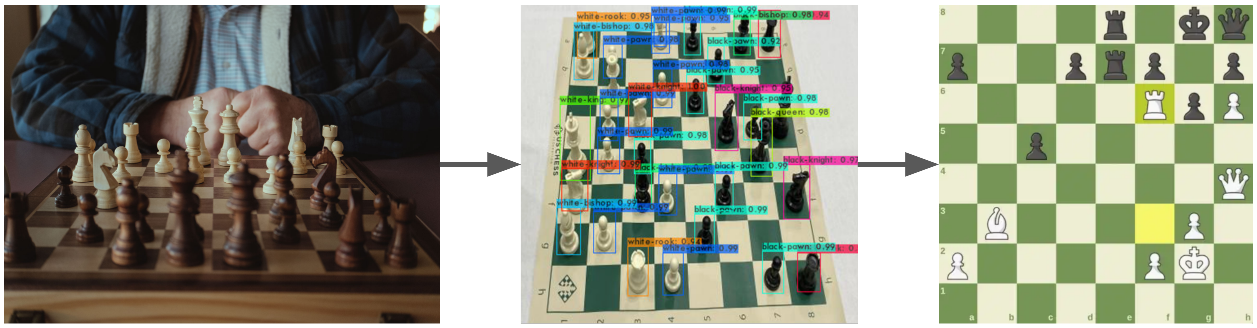


Figure 1.1: A visualization of how we might build an abstract planning representation for the problem of a robot playing chess. At the lowest level, our agent might take in an RGB image from its perspective. Using that input, we can detect individual pieces, along with their relative positions on the board, using modern computer vision techniques. From these detections we can construct our abstract representation, which ignores all information unrelated to playing chess, and organizes the information we keep into a form that enables efficient planning.

On its face, playing chess sounds a lot harder to most people when compared to walking down a hallway and making a cup of coffee. If I ask you to either make me a cup of coffee or win a game of chess against a grandmaster, which would you say is more difficult? Yet, while we have been able to produce agents that easily outperform the very best human chess players, these approaches have so far struggled to make it onto real-world robots. Why is that? To gain insight into this question, we should look closely at the strategies which have gained traction in chess-like domains. As mentioned above, one successful approach in classical AI involves a combination of policy learning and planning. Next, we will consider each of these approaches to decision making, and how they work in the context of robotics.

1.2 Making Decisions Within an Abstraction: Plans vs. Policies

In robotics literature, there are two basic strategies for how an autonomous agent might make decisions within an abstraction. Most of the work in this thesis is framed as a planning problem, where an agent has some model of the world in a particular state, and a set of actions that, when taken, change the state of that world in a predictable way. To find a plan, the agent searches for the sequence of actions that will put its representation of the environment in some desired state according to its models. This action sequence is referred to as a plan, and can be executed from the robot's current state.

An alternative approach to searching for a plan is to act according to some predetermined policy. Whereas a plan is a sequence of actions that transitions the world from a single initial state to a goal state, a policy is a function which maps each possible world state to an action which (hopefully) progresses toward the goal. Therefore, unlike a plan, a policy identifies which action is taken from any world state. Recently, there has been great progress in the space of learning policies to guide autonomous decision making. When applied to robotics problems, approaches like Reinforcement Learning (RL) [3]–[5] and Behavior Cloning (BC) [6], [7] have enabled robots to develop standalone skills such as locomotion of a legged robot or dexterous manipulation [8], [9]. Policy-based approaches can be widely applicable in that they do not always require a model of the world, meaning that an agent can reason about taking actions that are difficult to model or are partially observed. Moreover, once the policy is formed, the agent does not have to spend time "thinking" when attempting to execute the skill.

Unfortunately, while policy based methods can be very effective in shorter horizon tasks, they often struggle as the horizon length grows. Consider again the coffee problem, and imagine watching a human make coffee. At the outset, the human might start walking in a particular direction, which may or may not lead to the kitchen. As more actions are taken, it becomes difficult to identify what steps ultimately contributed to accomplishing the goal. How do we assign credit for taking a step vs. picking up the coffee beans? The best action at the outset may be to walk toward the kitchen, however, the robot might not appear to be materially closer to its goal after doing so if the kitchen is not in view. Chess is a very long horizon problem, yet policy based approaches can be effective in that setting. Notably however, for a chess playing agent, our abstraction perfectly matches reality, meaning we are able to easily acquire a mountain of data through self play in simulation. This data is a cheap and effective training signal that allows an agent to learn which board states are likely to lead to positive game outcomes, even if that outcome is many moves away. While the quality of simulation tools in robotics has improved in recent years [10], [11], this is only true in certain settings, and we cannot always assume that the data collected in simulation is a useful representation of the real world for long-horizon decision making. Conversely, collecting real data on robots can be very expensive. Large companies may be able to learn to manipulate objects using a farm of robot arms, but that is not possible in many settings. Without access to a significant amount of training examples, policy based approaches struggle as planning horizons grow.

Another reason we might not want our agents to make decisions solely through learned policy is due to the hope that we have to develop robots which are useful in a number of different contexts. Chess playing agents need only worry about chess, so we can design an abstraction that removes any information that might be irrelevant to its task and learn a single value function to approximate the value of a world state to guide planning. However, we want our robots to be able to adapt to different tasks without the costly step of retraining our models (and potentially having to collect an entirely new dataset). Even if we are able to learn effective policies for specific skills, reasoning about when and how to deploy those skills may still be best decided via planning. Policy-based RL and BC approaches certainly have a large part to play in the future of robotics, but in the context of certain long-horizon decision making problems, their limitations imply they cannot stand alone.

The work in this thesis therefore is focused on planning for decision making. When we plan, we use a model of the world to consider how sequences of actions will affect progress toward an agent's goal. However, planning in the context of robotics problems has its own set of challenges. Planning efficiency is one such limitation. While policy based approaches can reason quickly, planning involves looking forward many steps in time. Therefore, the computational complexity scales with the size of the environment, the length of the given task, and the breadth of available actions. We will explore the challenges of planning in a robotics context in the following section.

1.3 Hierarchical Planning and Downward Refinement

As we hinted at above above, planning within hierarchical abstractions in a robotics setting presents several additional challenges which are not always present in classical AI problems. In the case of playing chess for example, we defined a hierarchical abstraction where, at the highest level, the agent considers moving one of the pieces on the board to a legal position. If this game is played with a real robot in the real world, the agent will then physically pick up the piece, following a planned trajectory to eventually place it in the proper square. Notably, for our chess playing robot, we can be certain that when it considers making a move, that move can be made[2]. A piece is never obstructed by another such that it cannot be grasped, and the target square is always open. Because of this, we can develop a plan fully in our high-level abstract representation, and only consider finding a low level plan when it is time to execute a move. In practice, the first action of the highest level plan might be making a single move: "rook to F6". Refining this action at a lower level of abstraction could result in a plan like this: "move hand to rook," "grasp rook," "move hand to F6," and "release rook." Finally, the lowest level plan for this action would correspond to the arm's trajectory in joint space which execute those commands. This is known as top-down planning, and is not always possible in a robotics setting. If we try to find coffee by traveling down a hallway, there is no guarantee that the coffee is reachable, or even present in the kitchen at all, and we should consider what our next action would be in case the first action is unsuccessful. Similarly, when our robot reaches the kitchen, we know that not every mug in the cupboard is immediately graspable. If we try to pick up something in the back of the shelf, we may first need to move other objects out of the way, or risk knocking things over.

These examples illustrate the fact that many useful abstractions in a robotics context do not (and in fact often cannot) maintain the property of *downward refinement*, which is the idea that a solution found at one level of abstraction is valid at all lower levels[12]. For example, downward refinement holds when planning to explore a particular region of space is still feasible when we try to compute the necessary robot trajectory. Forming abstractions that do not have this property implies that relevant information was removed during their creation. If we treat our coffee making task like a game of chess, and plan in a top-down manner, we may see degraded search performance in the form of backtracking (sometimes worse than planning without the abstraction) [12]. There are a number of cases in robotics (planning in partially revealed environments, Task and Motion Planning, etc.) where the

[2]This is true under the assumption that the chess playing robot is constructed such that the entire board is within reach of its manipulator.

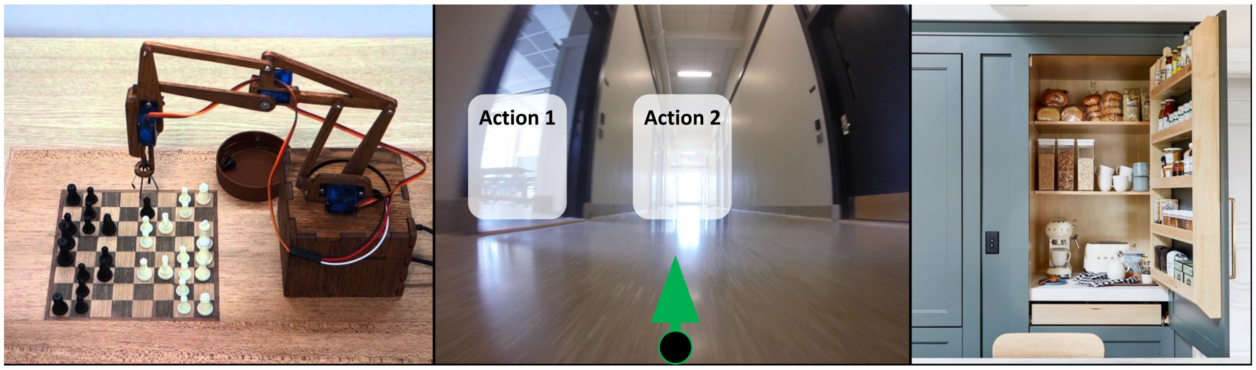


Figure 1.2: A visualization of how robotics tasks do not maintain the property of downward refinement in their abstractions. On the left, we see the chess playing robot preparing to make a move, and we can be confident that this move can be made. Conversely, in the center we consider a robot going either left or right down a hallway on its way to make coffee. Which of these actions will succeed is unclear, and the agent should account for potentially needing to backtrack at planning time. Finally on the right we see a crowded cupboard. If our robot attempts to grasp a mug in the back of the shelf without moving other objects out of the way first, it will knock over those obstructions, leading failed plans.

downward refinement property of an abstraction does not hold in general. Most available paths in a building do not lead to a kitchen, and the majority of potential grasps in a crowded cupboard do not yield collision free robot configurations. Planning naively in these types of problem settings can lead to inefficient planning, and sub-optimal solutions.

The goal of hierarchical planning is to produce a plan which is consistent across all increasingly complex layers of hierarchy before acting. In the case of attempting to pick up a coffee mug, this might entail first deciding which cup to grasp, then how to grasp it, and finally the trajectory that takes the robot from its initial configuration into one where it is holding the mug. However, as we have discussed, if the cupboard is particularly full, many potential grasps are infeasible, and trajectories that take the robot to the valid ones might bring the robot into collision with different obstacles along the way. Depending on the setting, if we consider high level actions naively, our planner will have to backtrack up the hierarchy frequently to find a feasible solution. Moreover, there are settings (like in partially revealed environments) where refining a plan to the lowest level is impossible without actually executing the action. In these cases, the robot will have to physically backtrack when a plan fails.

If the hierarchical planning problem is so difficult, how then are humans able to act quickly and reliably in the world? Returning to the coffee making example, we observed that humans often plan at a high level of abstraction before refining their decisions into low level motor actions. However, unlike many robotic systems, humans have some learned intuition about what actions are actually feasible, and which elements of the world are even relevant to a given problem. If placed in an unfamiliar building for example, most humans would reason that coffee is found in a kitchen, and so should not enter offices or classrooms to find their target. When planning to pick up an object, a human knows that if they see the mug blocked off in the back of a shelf, they will need to move obstructing objects out of the way before attempting a grasp. On the whole, humans are able to form efficient abstractions from perception, then, from experience, make predictions about when to trust which actions in those abstractions are both useful and feasible. The intuition that enables a human to plan and act quickly in such a massively complex problem is something which we would like to bestow upon our robots.

1.4 Contributions

The motivation behind this thesis is as follows. Hierarchical abstraction is necessary for enabling efficient planning in complex robotics problems. However, the abstractions we can build given the limitations of a real-world robot are inherently imperfect, and important properties like downward refinement do not always hold. As a result, the solutions returned by higher levels of a planner are often found to be invalid when refined at a lower level, leading to backtracking, wasted computational effort, and failed plans. The central idea of this thesis is this: **we can improve the efficiency of hierarchical decision making for real world robotics by explicitly reasoning about the imperfections of our abstractions during planning.** We can identify when we have discarded potentially relevant information in the process of forming our hierarchical representation, and use this information to learn, either online or from experience, properties about our abstractions. These predictions, such as which high-level actions can be refined at a lower level, or which objects in the robot's environment are relevant to its task, allow us to reason about when we can trust our planning abstractions, and therefore guide efficient search within them. In the following chapters, we present research which proves this idea across several different robotics settings, each of which we briefly introduce below.

Decision Making in Partially Revealed Environments: In Chapter 3 we address the problem of planning in the presence of uncertainty; in particular, we consider problems where a robot must solve some task in a previously unexplored environment. In such settings, abstract plans cannot be refined until the robot attempts to execute them because the world is not fully known at planning time. As a result, if the decision of which action to take is made naively, our robot may waste effort exploring regions of the environment irrelevant to its task, forcing the agent to physically retrace its steps at execution time. Building off work in Stein, Bradley, and Roy [13], we first define high-level actions derived from the environment and the given task itself, forming an abstraction which reduces the horizon of the search problem, but which we know does not satisfy the property of downward refinement. To overcome the imperfections of our abstraction, we attempt to estimate how each action contributes to our agent's progress towards completing its task. As the map is revealed, we predict the cost and the probability of success of each action from images captured by the robot and an encoding of that action using a trained neural network. In this work we consider complex tasks with temporal constraints, creating a difficult search problem. We propose a stochastic planning approach which uses our learned predictions to guide the search for the minimum-expected-cost plan. Moreover, our learned model is structured to generalize across environments and task specifications without requiring retraining. Over several different environments, varying in scale from a handful of rooms, up to the size of a building on MIT's campus, we demonstrate an improvement in total cost in both simulated and real-world experiments compared to a heuristic-driven baseline. The contributions in Chapter 3 were originally published in Bradley, Pacheck, Stein, *et al.* [14], and presented at the International Conference on Robotics and Automation in 2021.

Learning to Guide TAMP: In Chapter 3, we address planning in a setting where we can not confirm the feasibility of an abstract plan until attempting to execute it; however, there are many instances in robotics where we can refine our solutions to a low-level at planning time. In Chapter 4, we consider the problem of efficient hierarchical decision making in TAMP, where solutions take the form of high-level actions parameterized by low-level trajectories. Recent work in TAMP has enabled a new class of algorithms to better take advantage of off-the-shelf black-box samplers and solvers to find solutions to sub-problems in an abstract plan, such as motion between configurations, or inverse kinematics solutions. However, not all sub-problems are equally valuable, and many high-level plans turn out to be infeasible at a low level depending on the geometry of the scene. Existing planners typically rely on heuristics to determine which sub-problem to attempt to solve next, unable to reason about the expected cost of doing so in the broader context of the full plan until they attempt to actually solve the sub-problem directly. As such, these methods often attempt to solve sub-problems which are unlikely to succeed, leading to back tracking at planning time and wasted computational effort. We propose a novel approach for solving TAMP problems, utilizing learned models trained from experience to inform when to attempt to solve potentially expensive sub-problems using an encoding of the geometry of the scene. We take advantage of existing highly optimized planners by learning representations that can be integrated with existing abstractions to guide search in long-horizon TAMP domains. We test our approach in two simulated domains, as well as on a real Panda robot, showing improvement in planning and execution time compared to a heuristic driven baseline. The work in Chapter 4 was originally published in Bradley and Roy [15], and presented at the International Symposium of Experimental Robotics in 2023.

Hierarchical Planning in Hierarchical Scene Graphs: We demonstrate in Chapter 4 a planning approach which can reduce the time required to find a plan in many TAMP settings. However, as tasks become more complex and the scale of environments grow, how we build our planning abstraction becomes increasingly important. Recent work in the construction of 3D scene graphs has enabled mobile robots to build large-scale metric-semantic hierarchical representations of the world [16]. These detailed models contain information that is useful for planning, however an open question which remains is how best to derive a planning domain from a 3D scene graph that enables the efficient computation of executable plans. Including elements from a scene which are irrelevant to a successful plan into our planning abstraction can lead to high branching factors during search, and therefore inefficient reasoning. In Chapter 5, we present a novel approach for defining and solving TAMP problems in large-scale environments using hierarchical scene graphs. We describe a method for building sparse problem instances which enables scaling planning to large scenes, and we propose a technique for incrementally adding objects to that domain according to online feedback during planning time so as to minimize spending computation on irrelevant elements of the scene graph. Finally, we define a tri-level hierarchical planner to efficiently produce plans within our abstraction. In this way, we again use low-level information (here in the form of feedback from the scene's geometry), to guide hierarchical decision making. We evaluate our approach in two real scene graphs built from perception, including one constructed from the KITTI dataset. Furthermore, we demonstrate our system in the real world, building our representation, planning in it, and executing those plans on a real robotic mobile manipulator. These contributions were originally published in Ray, Bradley, Carlone, *et al.* [17], and presented at the International Symposium of Robotics Research 2024.

Over the next few chapters, we will discuss approaches to robotic planning which attempt to identify how to effectively guide high level search with low level information efficiently in a task agnostic way. First however, we will discuss the existing body of research in the field of robotics that is related to and built upon by the novel contributions of this thesis.

Chapter 2

Background and Related Work

We aim to enable our robots to solve complex tasks set in large, real-world environments. This involves addressing complications introduced by the scale, configuration, and uncertainty of the environment. In this thesis, we frame decision making as a hierarchical planning problem. Recent work in solving these types of problems is broad, and the space of relevant literature spans across a number of different research areas. In this chapter, we discuss hierarchical planning, focusing on the settings of 1) Task and Motion Planning (TAMP), 2) large scale planning problems, and 3) partially revealed environments. In particular, this chapter will provide useful background to the original research presented in later chapters. We also discuss existing planning approaches, and identify the shortcomings which motivate our contributions. To that end, we first define the hierarchical planning problem, highlighting where different planning approaches may be necessary depending on the problem setting.

2.1 Hierarchical Planning

In order for a robot to traverse and interact with its environment, commands must be sent to its motors, which serve to produce changes in the robot's configuration, moving it through the world. With this in mind, the goal of a robotics planning problem is to solve for a minimal cost trajectory in joint space for a robot to follow which satisfies some specified goal. A notable subfield within the larger umbrella of robotic planning research is motion planning. Stated formally, a motion planning problem [18] for a robot with d-degrees of freedom can be modeled as the search for the optimal trajectory of a point representing the robot's configuration through a d-dimensional configuration space $Q \subseteq R^d$ [19]. A *satisfying* trajectory brings the robot from its initial configuration to one of a set of goal configurations, where obstacles can be represented within Q as inadmissible configurations. The simplest way to attempt to solve a motion planning problem is by discretizing Q, and searching for a path through the resulting graph using search techniques like Dijkstra's algorithm or A^* [20], [21]. These approaches have a known worst-case complexity of $O(\texttt{branching-factor}^{\texttt{depth}})$, and therefore suffer struggle as the horizon length of a problem grows. More sophisticated approaches like sample-based motion planning [22], [23] and trajectory optimization [24], [25] can be more efficient in practice, though are still impacted as dimension and depth of search increase.

We are interested in solving problems where the robot must not only move through the world, but potentially alter the state of that world as well. This might mean that our agent is required to solve subtasks in a particular order, pick up objects, or otherwise directly interact with its environment. Unsurprisingly, this complicates our problem definition significantly, as the parameters and constraints of the motion planning problem for navigating from configuration A to configuration B may vary greatly from the parameters for doing so while holding an object (e.g., the robot's gripper may be constrained to be in contact with said object such that it is securely grasped). The point in a plan where the agent's constraints change (in this case, when the object is grasped) define a different planning *mode*. In Multi-Modal Motion-Planning (MMMP) problems, the intersection of configurations where the constraints for two modes are satisfied define hyper-planes in configuration space—think of the subset of configurations where a robot successfully grasps an object on a table—which are often of lower dimension than the configuration spaces of either mode. This implies that the probability of sampling configurations from these *mode switches* is identically zero, and so we cannot hope to do so naively with traditional motion planning techniques [19].

In practice, we can solve these kinds of problems by making discrete decisions of which planning modes the robot will switch between, intentionally sampling configurations on these transition manifolds, and treating the search for trajectories between these points as separate motion planning problems [19]. This strategy defines a planning abstraction which naturally exists in a hierarchy. At the highest level, we solve discrete decision making problems (e.g., which object to grasp, what surface to place it on, etc.), and use those solutions to guide the search for lower level parameters like configurations and trajectories. The form of the interplay between solutions to the higher- and lower-level problems is one feature which separates different approaches to hierarchical planning.

2.1.1 Top-Down Planning

The simplest strategy for a hierarchical planning problem is to solve for a plan at the highest level of abstraction first—i.e., choosing which transition manifolds to consider—and only solving for the low-level trajectory during the execution of that plan. This is appealing for a number of reasons, chief among them being it can be difficult in certain settings to construct abstractions where a planner can produce useful low-level trajectories efficiently. The earliest examples of robots planning in the real world, for example the SHAKEY robot, took this approach out of necessity [26]. At planning time, SHAKEY considered abstract actions (e.g., which room to navigate to), and only determined if its decisions were actually feasible when attempting to execute those actions.

A common formalism for encoding discrete planning problems—such as those found at the highest planning hierarchies—is the Planning Domain Definition Language (PDDL) [27][1]. In a PDDL problem, a *state* $\mathcal{I}$ is a set of facts, where each fact is an instance of a boolean function called a predicate $p(\bar{x}) \in \mathcal{P}$, which is parameterized by a tuple of symbols $\bar{x} = [x_1, \ldots, x_k]$ from a given set of symbols[2] $x \in \mathcal{O}$. Each symbol x_i is a discrete element of a state variable. Transitions between states are defined by actions $a(\bar{x}) \in \mathcal{A}$ (also parameterized by symbols) which are expressed as two sets of predicates: preconditions $\text{Pre}(a_i)$ and effects $\text{Eff}(a_i)$. An action's preconditions determine if an instance of that action can be applied from a particular state $\mathcal{I}$, and its effects define the set of facts that are added ($\text{Eff}^+(a_i)$) or removed ($\text{Eff}^-(a_i)$) from the state $\mathcal{I}$. One special case of an effect is to increase a metric of

[1]SHAKEY utilized a STRIPS style planning approach, which is similar to PDDL [26]

[2]Commonly, the elements $x \in \mathcal{O}$ are referred to as *objects*. However, some subset of these *objects* may refer to actual physical objects in a scene (e.g., a block on a table), while others represent something like a region or trajectory. To disambiguate the overloaded object term, in Chapter 5 we choose to refer to these elements as *symbols*, and note the discrepancy here.

cost, allowing us to compare the relative value of two plans. A planning *domain* is composed of lifted sets of predicates $\mathcal{P}$ and actions $\mathcal{A}$, and a problem *instance* $P = (\mathcal{P}, \mathcal{A}, \mathcal{O}, \mathcal{I}_0, \mathcal{G})$ combines a domain with an initial state $\mathcal{I}_0$ and a set of goal states $\mathcal{G}$, parameterized by symbols $\mathcal{O}$. If the current state is in the set of goal states, the problem is solved. Below we see a simple example of a PDDL action for moving from one room to another:

```
:action moveRoom
    :parameters (?x - robot ?r1 - room ?r2 - room)
    :precondition (and (inRoom ?x ?r1))
    :effect (and (inRoom ?x ?r2)
                 (not (inRoom ?x ?r1))
                 (increase (total-cost) 1))
```

Solutions to planning problems encoded in PDDL take the form of a sequence of parameterized action instances $\pi = [a_1(\bar{x}_1), a_2(\bar{x}_2), ..., a_n(\bar{x}_n)]$, where the state after taking each action satisfies the precondition of the following action [19]. If the effects of the final action in a plan π result in a state contained in the goal set, the plan solves the problem. A range of solvers [28], [29] have been developed to solve tasks specified in PDDL.

We note that, in the `moveRoom` action shown above, there is no reference to how the robot will move between these rooms. Therefore, if such a path does not exist in reality, the plan will not be executable. If a decision made by a top-down planner is invalid, the robot will be forced to physically backtrack and choose a different path. SHAKEY was unable to refine a high-level plan to low-level trajectories during planning in part because, at the time it was being developed, algorithms for low-level motion planning were not yet efficient enough (nor were on-board computers powerful enough) to be used on a real robot. Over the intervening decades however, great progress has been made in the space of motion planning. Notably, low-level motion plans can now be solved at incredibly high rates on CPUs [30], meaning it is not always necessary to delay the process of refining high-level plans to low-level trajectories. However, there do remain certain robotics problems where it is still challenging to do anything but plan in a top-down manner. One such case is in the presence of uncertainty, particularly when a robot is required to act in an environment that it has only partially explored. We consider relevant approaches for solving these types of problems later on in section 2.3.4.

2.1.2 Task And Motion Planning

In problem settings where we are able to build an accurate model of the world, naive top-down approaches often produce sub-optimal plans in robotics settings. As discussed in Chapter 1, planning abstractions in robotics problems often fail to satisfy the property of *downward refinement*. Therefore, many modern hierarchical planning approaches will attempt to refine their high-level plans into low-level trajectories before attempting to execute them. Research in this direction falls under the umbrella of Task and Motion Planning (TAMP), which augments the MMMP problem with non-geometric actions [19].

The TAMP problem jointly considers elements of high-level task planning [31], [32] and low-level motion planning [33] in an attempt to solve hybrid discrete/continuous, multi-modal planning problems [19]. As discussed above, each motion-planning sub-problem within the larger TAMP problem is differentiated by its constraints; we can therefore view TAMP as a hybrid constraint satisfaction problem [19], implying that any solution to one sub-problem must also be careful to not violate the constraints of another. Most TAMP methods take one of two approaches, either solving the full set of constraints jointly or treating each sub-problem individually. In the first approach, the problem is written as one large constrained optimization problem (typically a Mixed Integer Program), where discrete components such as which block to pick up are represented by integers, and the trajectory optimization is real-valued [19], [34], [35]. These joint optimization strategies have certain advantages, as satisfying a single optimization solves the entire problem. However, such approaches are often limited in that certain aspects of the problem may not be easily differentiable, efficiently reusing computation can be difficult, and it might not always be straightforward to incorporate off-the-shelf external samplers or solvers [19].

The primary alternative approach is to consider solving for sets of parameters that satisfy small groups of constraints, and combine the solutions into actions and plans. For example, we can sample a block placement on a platform that is free of collisions, then confirm it is positioned so that there exists a kinematically feasible configuration to execute such a placement. Approaches that break up the problem in this way can take advantage of external tools that are optimized for specific sub-problems (e.g, Fast Downward for the discrete decision making problem, efficient inverse-kinematics solvers to find collision free high-dimensional configurations, or neural networks to sample stable grasps) [19], [36], [37].

In order to solve TAMP problems, we need a method of specifying the task to our robots. Top-down planning approaches utilize PDDL to encode their planning domains, however the continuous nature of TAMP problems makes discretizing and encoding a planning problem directly in pure PDDL infeasible. For certain actions, preconditions may include facts that are either cumbersome or impossible to add to the domain. For example, it is unclear how one would enumerate all possible configuration objects for a 20-DOF robot without creating a potentially intractably large problem. To account for this, Garrett, Lozano-Pérez, and Kaelbling [36] define PDDLStream. PDDLStream problems augment PDDL encodings with object generators called *streams* $s \in S$, which allow the planner to represent sub-problems relevant to the problem. Streams consist of: sets of (1) input and (2) output objects, (3) domain predicates which must be true in the input, (4) action predicates to be certified if the queried stream is successful, and (5) an external function that is called when the stream is queried. When an action has a precondition which can only be certified by a stream, that stream can be queried in an attempt to solve the associated sub-problem, and determine if said precondition can be certified to be true. In this way, we can define a discrete planning problem in PDDL, and add the solutions to continuous sub-problems to that domain as needed. We consider encoding TAMP in PDDLStream in greater detail in Chapter 4.1, and refer the reader to Garrett, Lozano-Pérez, and Kaelbling [36] for a more detailed description.

Different TAMP techniques can be further classified into one of three categories [19]. **1)** Sequence first strategies solve for an abstract plan and then attempt to solve for the parameters which would produce a concrete plan. These approaches are most like the top-down strategy discussed earlier. However, if we have a mechanism for *backtracking* during planning, we can search for new high-level plans when refinement fails [38]–[40]. **2)** Satisfaction first approaches flip this idea on its head by solving select sub-problems first, then using those concrete parameters in the search for high-level action plans [41]–[43]. For example, we can solve for various grasps on a number of objects, any of which might be used in a plan which moves the object. If a complete plan cannot be found with the given parameters, more sub-problems can be incrementally solved for new ones. These approaches can struggle in cluttered environments, where many objects in a scene are irrelevant to the final plan. **3)** Finally, we can *interleave* the search for high-level plans and low-level parameters, and thus exploit the advantages of each strategy [36], [37], [44].

There are several TAMP solvers that have been developed to use PDDLStream to define TAMP problems, some of which we have already touched on. One general approach is to optimistically assume that any time a stream is needed to certify an action predicate, it can be queried successfully, then generate abstract plans π, which contain actions that have unknown parameters as a result of this assumption. The approach outlined in Garrett, Lozano-Pérez, and Kaelbling [36] is an *interleaved* method, where we can define certain "pre-discretized" elements, as well as those sub-problems for which it is more efficient to delay solving. Ren, Chalvatzaki, and Peters [37] similarly encode TAMP problems in PDDLStream, however, search through potential plan skeletons using stochastic search. Regardless of the strategy however, TAMP remains a complex planning problem. As proven in Vega-Brown and Roy [45], TAMP is P-SPACE complete, and so great care must be taken to plan intelligently. Recent research has attempted to overcome the inherent difficulty of TAMP by guiding decision making with learning.

2.1.3 Learning to guide TAMP

There has been significant recent progress in improving planning for TAMP problems using learned models. Most relevant to the work in this thesis are those contributions which attempt to accelerate search from experience [46]–[53]. Some learn explicitly which components of a given domain are relevant for a particular TAMP problem, though do not further guide search within their reduced domain [49], [50]. Kim, Kaelbling, and Lozano-Pérez [51] learn an action sampling distribution for geometric motion planning problems, but do not take advantage of off-the-shelf samplers or solvers. Kim and Shimanuki [48] learn a Q-function as a heuristic to use in search for a geometric TAMP problems, however do not learn to bind the continuous parameters of its actions. Closely related, Khodeir, Agro, and Shkurti [53] specifically score the relevance of streams within the PDDLStream framework, and improve search for stream-plans. However, they do not consider the search for an action's parameters. Finally, the work in Xu, Ren, Chalvatzaki, *et al.* [46] learns a feasibility predictor from images to accelerate stochastic tree search [37]. However, this work does not predict costs, and thresholds the predicted feasibility to bound branches in search. This can lead to planning failure if a feasible branch is below the defined threshold.

We address the problem of guiding TAMP in Chapter 4. In the following section, we consider how we can build the representations needed for hierarchical planning on a real-world robot.

2.2 Building Hierarchical Planning Abstractions

In order to make decisions on how to act in the world according to the approaches considered in Section 2.1, a robot must have access to some sort of representation of its environment, which it can then use to derive a planning abstraction. In many problems considered by the AI planning community—chess is one of these—the planning abstraction can be defined by hand[1]. However, a robot operating in the real world may be required to deduce its planning representation from information it receives from its sensors. Robotic mapping can be used to create useful representations of an environment from sensor measurements [54]. The term *useful* in the above sentence can mean different things depending on the robot's capabilities and the requirements of any potential assigned tasks. In the context of planning, a map is useful insofar as it allows a robot to efficiently solve for an executable plan to move through and interact with the space it represents.

The type of environmental representation appropriate for a particular robot therefore depends in part on the way it perceives the world, i.e., the sensors available to it, as well as the task it is trying to accomplish. Consider a robot equipped with sensors that are able to determine the distance to obstacles within direct line of sight. Hardware like LIDAR or SONAR are examples of such specialized sensors that provide this information and are common in the field of robotics [55]. Similarly, range information can be extracted from depth cameras (RGB-D), or even a calibrated stereo camera system. Tools such as these are well-suited to build dense geometric representations of the world which store information relevant for avoiding obstacles during navigation. However, we are interested in giving our robots tasks beyond simple navigation, and so a representation of free space may be insufficient for producing useful planning models. To build more detailed maps, we require perception systems which can identify relevant non-geometric, semantic information in a given scene. Recent progress in the area of deep learning has enabled real-time object detection and semantic segmentation of camera images [56], [57], which in turn has made possible the building of hybrid metric-semantic maps online [16].

2.2.1 Metric Representations

One of the most common map representations used in robotics (and one of the first hypothesized) is the occupancy grid. Developed by Elfes and Moravec in the 1980's, occupancy grid mapping is a technique to convert depth measurements into a dense representation of obstacles in the world [58]–[60]. Occupancy grids probabilistically represent the world as a discretized grid, where each cell is either free space, occupied by an obstacle, or unobserved. An advantage of this representation is that it is well understood both how to construct and plan through such a grid [20], [21]. Furthermore, because a grid can explicitly represent unknown space, it is possible to generate plans through an occupancy grid which can be useful for navigation through partially explored environments. In fact, the work in Chapter 3 takes advantage of this property. In higher dimensions, volumetric models [61], point-clouds [62], and 3D meshes [63] are all ways to perceive and represent the observed free space of an environment. However, simple occupancy information is not sufficient if we want our robots to be able to solve complex tasks beyond navigation.

In order to enable our robots to solve more complex tasks, we must include information beyond simple geometry when building our maps. For example, we can further augment an occupancy grid with *semantic* labels (e.g., whether a light is on or off or the color of a block) to form a Labeled Transition System (LTS). We delve deeper into the semantics of an LTS in Chapter 3.1.1, but LTS can be thought of as an occupancy grid, where each element in that grid has some semantic marker associated with it. Like an occupancy grid, this label could indicate that the cell is occupied by an obstacle, however a cell might also have other labels. For example, if our cameras detect coffee beans within a particular grid cell, we apply a *coffee beans* label to that cell in the LTS. Beyond two dimensional representations, we can add semantics to volumetric [64] and mesh [65] representations. These types of hybrid semantic-geometric abstractions allow a robot to represent tasks such as moving to particular semantic regions in a map (perhaps in some specific order). They are particularly useful for the kind of top-down planning approaches discussed in section 2.1.1, and later in Chapter 3. However, if we want our robot to be able to reason about actions in higher dimensions (for example, actually moving an arm to pick up the coffee beans) we must construct hierarchical world models to enable hierarchical planning methods.

2.2.2 Hierarchical Mapping

In order to utilize hierarchical planning approaches discussed in Section 2.1, our planning domains must contain both high-level abstract elements (like objects), as well as low-level geometric information. The primary aim of Simultaneous Localization and Mapping (SLAM) research is to accurately estimate a robot's trajectory through space from sensor input, and so build a factor graph of landmarks and poses which are optimized using bundle adjustment [66]. The goal of accurate map building is somewhat, but not perfectly, aligned with our interests, which are to build a detailed hierarchical representation that is useful for planning. For example, one line of research in the SLAM community includes object detections as well as metric information in the map representation [67], [68]. These objects can be useful for the purposes of localizing an agent within its map, but also identifying elements in a scene a robot may be able to manipulate or otherwise interact with.

There have been focused efforts in the direction of hierarchical map building for the purposes of planning. Early work in hierarchical mapping set out to combine low-level dense metric maps with higher-level topological representations [54], [69]. Using these topo-metric abstractions, hierarchical planners can solve navigation problems first quickly in the topological layer, then refine solutions to trajectories in the dense, metric level. In higher dimensions, there has been substantial recent work enabling construction of information-rich 3D scene graphs, initially introduced by Armeni, He, Gwak, *et al.* [70]. Subsequent work focused on building 3D scene graphs from real-world sensor data [71], real-time performance [72], [73], and improving the higher-level abstractions [74], [75]. The strong performance of foundation models on open-vocabulary tasks (e.g., Large Language Models such as ChatGPT [76]) has led to a series of works on combining open-vocabulary language embeddings with 3D scene graphs [77]–[79]. These open-vocabulary works all feature object navigation or retrieval tasks executed on real robots, although the task structure is simple and the focus is on mapping natural language to an object grounded in the scene graph.

These maps often have a semantically labeled metric mesh at the lowest level, with higher levels of abstraction layered on top. One notable representation with this structure are Hydra scene graphs, presented in detail by Hughes, Chang, and Carlone [16]. The specific layers of a Hydra scene graph are as follows:

1. **Metric/Semantic Mesh:** The lowest level of a Hydra scene graph is a triangular mesh, where nodes are labeled according to semantic class predicted by on-board semantic segmentation predictions.

2. **Objects and Agents:** The second layer is composed of objects and agents (such as humans or robots), generated by identifying and clustering regions of homogeneously labeled mesh elements. If a cluster is large enough (and fits the definition of a known object), that object is included in this layer.

3. **Places and Structures:** Structures define elements in the scene like walls, which divide free space. Places conversely represent clustered regions of free space, indicating where a robot may be able to safely travel. Edges connecting place elements imply traversability (though this is not always the case in practice).

4. **Rooms:** Levels 4 and 5 of the scene graph contain information relevant to higher levels of connectivity. For indoor environments, at level 4 we cluster places into rooms.

5. **Buildings:** We can further cluster connected rooms in buildings. These regions can be used to guide hierarchical planning. Notably, there are versions of Hydra where the higher levels of abstraction are not defined by hand, but are more adaptable, allowing the framework to be used in a variety of settings (including outdoors) [75].

In the following section, we introduce research related to planning efficiently within 3D scene graphs.

2.2.3 Planning in Large Scale Environments Using Scene Graphs

Scene graphs, such as Hydra, are useful representations which can allow agents to encode large environments. In order to use these models for planning, there has been recent work focused on deriving structured planning domains from 3D scene graphs. Agia, Jatavallabhula, Khodeir, *et al.* [50] derive a PDDL representation for task planning from scene graphs. The amount of data contained within a scene graph representing a large environment can be quite significant. Therefore, the richness of data stored in a scene graph can be overwhelming to a planner, and so the authors utilize the graph's hierarchy to sparsify their representation and make planning tractable. Specifically, the authors remove all symbols that are not related to an element specified in the agent's goal (or related higher-level symbols) in order to prune the planning domain. For example, if the robot is tasked to go to a particular place, all other places are removed, as are rooms or buildings that the given place is not in, etc. Due to this aggressive pruning, the approach used by Agia, Jatavallabhula, Khodeir, *et al.* [50] is only guaranteed to produce valid solutions for very specific planning domains, and it is unclear how to extend this approach to a more general set of planning tasks. Dai, Asgharivaskasi, Duong, *et al.* [80] present a method for grounding natural language commands in LTL formulae, leveraging the hierarchy of the scene graphs to accelerate planning. Unfortunately for both of these approaches, scene graphs built in the real world may not satisfy the property of downward refinement [12], breaking many of the assumptions in symbolic task planning. The existence of low-level geometric constraints means that aggressive pruning and downward planning approaches lead to plans which cannot be executed in the real world. In Chapter 5, we address this by deriving TAMP representations from scene graphs, and consider how we can reduce the size of these domains while accounting for the imperfections in the scene graph abstraction.

There has recently been additional work in accelerating TAMP. Some efforts focus on identifying superfluous elements in scenes which would otherwise distract search. Silver, Chitnis, Curtis, *et al.* [49] learn to predict which symbols are relevant to a particular TAMP problem. Khodeir, Sonwane, Hari, *et al.* [81] and Vu, Migimatsu, and Bohg [82] both address PDDLStream's poor scaling as the number of objects grows, with Khodeir, Sonwane, Hari, *et* *al.* [81] proposing a algorithm that guides the search through task skeletons based on failures by the motion planner and Vu, Migimatsu, and Bohg [82] introducing a more intelligent method for instantiating streams. Meanwhile, Khodeir, Agro, and Shkurti [53] and Bradley and Roy [15] learn to guide search through predictions of relevance or feasibility. These approaches are complementary to the work presented in Chapter 5, though learning in large scene graph suffers from issues of generalization. Outside of learning based approaches, Srivastava, Fang, Riano, *et al.* [39] attempt to guide TAMP from failed motion plans, much like we do in Sec 5.2.4, however do so by adding additional goal conditions, an approach that can struggle certain settings, but can be complementary to our own. In Chapter 5, we propose an approach to pruning large-scale scene graphs which enables efficient planning in these domains.

2.2.4 Deriving Abstractions over Actions

So far in this section, we have discussed research related to forming hierarchical abstractions of a robot's environment to enable efficient planning. However, we have not yet considered how the actions the robot can take in those environments are derived. There is a long history of research in defining abstractions over actions for robot planning problems [83]. Often, in representations like PDDL, these abstractions are defined by hand, and correspond to the capabilities of robot [19]. However, recent work has shown that deriving an action set from the environment according to some predefined rules can lead to more efficient planning[13]. We often take this approach in this thesis. More recently, there has been work in the direction of autonomously generating symbolic representations from low-level robotic skills [1], [84]. In this way, an agent can use experience to shape its world model, and define how it can act within that model. Though we do not use this approach in this thesis, these strategies are complementary to our own, and can be incorporated in future work.

2.3 Planning in Partially Revealed Environments

The work discussed in Section 2.2 aims to enable an agent to build a detailed map as it travels. However, until the entire world has been observed, there will be areas for which the robot does not yet have an accurate model. In this section, we provide background and related work to the research presented in Chapter 3, wherein we address the question of hierarchical planning in partially revealed environments.

Planning in the presence of uncertainty is a ubiquitous problem in robotics. Every part of the robotic system is, in some way, affected by the reality that we cannot know the true state of the world, nor can we know how our actions will affect that state exactly. No sensor can give an agent instantaneous and complete knowledge of its environment at every point of time; robots can only observe what is in direct line of sight and within some limited range. Laser range sensors, RGB cameras, and other instruments may only be reliable up to a certain distance, and none of these "see" through physical obstructions like walls. Even if a robot's sensors are perfectly noise free, and detect everything within their range exactly, there will inevitably be regions of the environment where the robot has incomplete information. Despite this, we still want our robots to make informed decisions even when the world is not fully known.

2.3.1 Modeling Planning Under Uncertainty as a Partially Observable Markov Decision Process (POMDP)

The problem of autonomously acting within unknown environments can be formulated as a Partially Observable Markov Decision Process (POMDP)—a generalization of an Markov Decision Process (MDP) [85]. In an MDP, an agent knows exactly the state of the world, what actions it can take for each state, and the possible outcomes of each action in each state. The defining characteristic of a POMDP however, is that the agent does not have full knowledge of the state of the world at all times, and so must maintain a belief of the true state based on its observations [85]. The belief at a particular state $b_t(s)$ can be defined as the probability that s represents the true state at a given time. The belief is a sufficient statistic for all past observations and the initial prior belief state of an agent, meaning the distribution encodes all information needed to act under uncertainty [85], [86].

A POMDP can be written as an n-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, \Omega, \gamma, b_0)$, each element of which we define below:

1. **State** $\mathcal{S}$: The state may represent both the internal state of the robot $\mathcal{Q}$, and the state of the world, usually represented by a map $\mathcal{M}$, such that $\mathcal{S} = \{\mathcal{M} \times \mathcal{Q}\}$. Depending on the robot, a robot's state can have many components such as various joint angles and velocities, or be as simple as a pose in 2D space. The state of the map, which can be continuous or discretized (like an occupancy grid), does not change with actions. However, our belief about the map does change with each observation, hence decisions are made with respect to the belief.

2. $\mathcal{A}$ Actions: The actions available to the robot at any given instance. An action takes the robot from one state to another with some probability. In many robotics problems, the action set is some low-level control, like changing the steering angle or velocity. We might also consider higher-level abstract actions. This will be discussed in greater detail in Chapter 3. In solving a POMDP, the robot will attempt to find the optimal action for a given state with respect to some objective function.

3. $\mathcal{T}$ Transition Probability: The probability that taking a certain action will result in the robot being in a given state. For example, if a wheeled robot is directed to make a turn, there may be some probability that the wheels will slip, and the robot will end up in a different state than what was intended.

4. $\mathcal{R}$ Reward: The reward (sometimes formulated as a cost) is an element in the agent's objective function that the autonomous agent is maximizing (or minimizing in the case of cost) to solve the POMDP. Rewards can be given for taking certain actions, or for reaching intermediate states. We can also define the reward such that the optimal plan a robot can find is the one that reaches the goal in the shortest possible time.

5. Ω Observation: An observation is all information that an agent receives about the world at a given time. The observation contains information about the state of the world, and is generated probabilistically according to some known distribution. For the case of a discretized state-space such as an LTS, an observation contains information about which cells in the grid are occupied by obstacles.

6. $\mathcal{O}$ Observation Probability: The conditional observation probability represents the prob-

ability that a given observation correctly represents the world. The uncertainty of which observation is generated can come from noise in the sensors, though in this work we are primarily concerned with incomplete observations received by a mobile robot. For example, if the robot receives an observation from a laser-range sensor, it only has information about the environment that is in direct line of sight from the sensor. Obstacles occlude whatever is behind them, so measurements of the occluded region would have a conditional observation probability of zero.

7. γ Discount Factor: A scalar value $0 \leq \gamma \leq 1$, which accounts for the present value of future rewards. The discount factor is relevant when we attempt to compute the relative rewards (or costs) of different sequences of actions. A high discount factor implies that future and present reward are valued similarly. In our work, we do not discount future reward at all, and so set $\gamma = 1$.

8. b_0 Initial Belief: Our prior for the initial state of the world. In a setting with perfect local sensing, this would be represented by certainty over observed space. Without any additional information, we would conversely have a uninformed prior over all unobserved space.

To "solve" a POMDP, the agent searches for the best action it can take in a given state, with the goal of optimizing some objective function. The expected cost of an action under the optimal policy can be computed recursively using the Bellman Equation [87]:

$$Q^*(b_t, a_t \in \mathcal{A}(b_t)) = \sum_{b_{t+1}} P(b_{t+1}|b_t, a_t) \left[R(b_{t+1}, b_t, a_t) + \gamma \min_{a_{t+1} \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a_{t+1}) \right] \tag{2.1}$$

where $P(b_{t+1}|b_t, a_t)$ is the probability of reaching belief b_{t+1} taking action a_t from belief b_t, and $R(b_{t+1}, b_t, a_t)$ is the expected cost (or negative reward of doing so).

Eq. (2.1) is often referred to as *doubly exponential*, and is extremely challenging to compute. Over a given planning horizon T, the complexity grows exponentially first over the available actions at each step. The same is true for fully observable MDPs, however, in a POMDP, we must also account for the partial observability of the problem. At each step, we consider an action not only for all previous actions, but also for all possible histories of observations (action observation pairs). As a result, any problem which we frame as a POMDP can become generally intractable as action and observation spaces grow.

2.3.2 Solving POMDPs Exactly and Approximately

Work on solving navigation and exploration tasks in uncertain environments using POMDPs dates to the mid-1990's [85], [88], [89]. Due to the computational challenges of solving POMDPs however, most techniques established through 1990 were too inefficient to be used on anything beyond very simple cases (2-5 states at most) [90]. In 1994, Cassandra, Kaelbling, and Littman pushed the field forward when he published his work on the "Witness Algorithm," which is capable of solving POMDPs in environments with up to 16 states exactly [91]. This size of state space is obviously quite limiting, and so approximate methods for solving POMDPs are often used in practice. Approaches like point based value iteration [92] and Q-learning [88] are a few such methods. Despite these advancements however, as the size of environments expands beyond simple toy examples, these methods are no longer tractable for a robot operating in real time.

Recent efforts have been made to approximate POMDPs using stochastic search to solve problems of a larger scale in an any-time manner. One class of approaches involves solving smaller problems which yield a policy similar to the solution to the full POMDP. DESPOT (and its derivatives) search a set of randomly sampled *scenarios*, which helps alleviate the problems associated with solving POMDPs in high-dimensional state-spaces [93]. The algorithm finds an approximately optimal policy by simulating the execution of all policies in the belief tree of sparsified samples. Another class of online solver similarly uses Monte-Carlo Tree-Search (MCTS) style stochastic search to sample actions, state transitions, and observations to find the highest value action sequence [94]. POMCP is designed for discrete state and action spaces, while POMCPOW is specifically designed to handle continuous state, action, and observation spaces [95]. While each of these methods returns an estimate of the best action after a search budget has been exceeded, the quality of that solution decreases as the problem becomes more complex. Therefore, long-horizon planning using these approaches is still difficult, particularly as state, action, and observation spaces grow.

2.3.3 Navigation In Partially Revealed Environments

While a major goal of this thesis is to enable solving long-horizon planning problems, work in navigating short-horizon navigation problems can be insightful. Recently, progress has been made in searching for collision-free paths through partially explored environments, wherein an autonomous agent must reason about unknown space at a small scale. Karaman and Frazzoli [96] navigate unknown random forest environments by making strong assumptions about the environment distribution. Since they know the locations of trees in their forests are generated by a homogeneous Poisson process, and know the dynamics of their agent (modeled after a bird at high-speeds), the authors are able to bound collision probabilities for given speeds. Some recent work has instead focused on using a dynamic action set [97] or boundaries between free and unknown space [98], [99] for navigation/exploration of unknown environments.

If the distribution over states is not known, learning can be used to estimate the expected cost [100] of actions in unknown space. Most literature that uses learning to explore unknown environments focuses on short time horizon planning. Richter *et al.* [101] learn to solve an approximate POMDP for navigation of unknown environments, yet restrict their planning horizon to only a few time-steps. In Richter and Roy's later work [102], the authors leverage supervised learning to estimate the collision probabilities of trajectories that enter unknown regions of the map in order to navigate structured environments more quickly. Though predictions are limited to short-term trajectories, these publications were inspirational in a number of ways. First, the authors utilize a discrete action set of 50 different actions as an action-centric abstraction for planning. Second, approximations are made to the Bellman equation to allow it to be factored into components (such as collision probability) that can be estimated via learning. The key takeaway is that we can plan abstractly in a robotics context more efficiently if we can predict in what context our abstract actions will be refinable.

Stein, Bradley, and Roy [13] build upon the approach in Richter and Roy [102], making progress in planning to minimize cost in navigating partially explored environments, now over longer planning horizons. To make tractable predictions about the unknown portion of the environment, the authors factor the Bellman equation to evaluate expected cost of an action in terms of the high-level decisions available to an agent, and the high-level outcomes that result. Specifically, they define subgoals in the plans they generate as *frontiers*—the boundaries between free and unknown space in a dense map representation—and consider attempting to reach the final goal beyond each of these frontiers as a separate high-level action. By reformulating the equations for planning hierarchically in terms of these high-level actions—for which they learn the costs and probabilities of success—and outcomes, the agent can navigate intelligently in partially revealed environments without the need to enumerate all possible configurations of unknown space. The authors take a top-down planning approach by only refining to a low level trajectory during execution, but are still able to perform well by utilizing their learned models to predict when an action can be refined to a executable trajectory, and the cost of doing so. Such approaches demonstrate improved navigation in partially revealed environments, but make assumptions which make solving more complex tasks impossible. We attempt to address these limitations in Chapter 3.

2.3.4 Solving Complex Tasks Under Uncertainty Using Temporal Logic

In Stein, Bradley, and Roy [13], the authors focus on the task of goal-directed navigation only. As a result, the only outcomes the system considers when choosing a particular high-level action are whether the frontier of interest will or will not lead to the unseen goal. For more complex tasks, single goal locations may no longer make sense, and planning intelligently to achieve specifications with such non-markovian goals requires considering both a different planning abstraction and approach. One approach for specifying more complex tasks to a robot is with Linear Temporal Logic (LTL) [103], which allows a user to define goals where the robot may be required to satisfy various subtasks.

Temporal logic synthesis has been used to generate provably correct controllers for solving LTL based problems, although predominantly in fully known environments [104]–[108]. Recent work has looked at satisfying LTL specifications under uncertainty [109], [110], yet these works are restricted to small state spaces due to the general nature of the POMDPs they handle. Other work has explored more restricted sources of uncertainty, such as scenarios where tasks specified in LTL need to be completed in partially explored environments [111]–[115] or by robots with uncertainty in sensing, actuation, or the location of other agents [116]–[119]. Ayala, Andersson, and Belta [111] introduce a method for a robot to explore an environment while not violating an Syntactically Co-Safe Linear Temporal Logic (scLTL) specification until it has found a path to satisfy the specification. This method uses a heuristic based on the specification and known space to decide where to travel next. However, it does not incorporate information about what could be in unknown space to minimize the cost of fulfilling the specification, resulting in suboptimal plans in general. Sarid, Xu, and Kress-Gazit [112] explore an environment while satisfying a specification and, as more of the environment is revealed, replan to incorporate newly discovered areas into the specification, though the cost of satisfying the specification is not considered. In Lahijanian, Maly, Fried, *et al.* [113], given a specification and a map of an environment, the authors plan a path to satisfy the specification and re-plan when they encounter unexpected obstacles. When these robots plan in partially explored environments, they take the best possible action given the known map, but either ignore or make naive assumptions about unknown space. This leads to inefficient planning as the robot may frequently be forced to backtrack depending on the orientation of unobserved space. In Chapter 3, we attempt to address this by learning when we can trust our abstract actions to be feasible.

To minimize the cost of satisfying LTL specifications, other recent works have used learning-based approaches [120]–[123], yet these methods are limited to relatively small, fully observable environments. Littman, Topcu, Fu, *et al.* [120] demonstrate a formalism for building a MDP from an LTL specification that enables Q-learning to provably converge. Toro Icarte, Klassen, Valenzano, *et al.* [123] present a method for decomposing an LTL specification into subtasks, which can then be learned using separate Deep Q-networks and combined in new specifications. Li, Vasile, and Belta [124] demonstrate learning in continuous manipulation environments, but are similarly restricted to fully observable domains. Sadigh, Kim, Coogan, *et al.* [125] and Fu and Topcu [121] apply learning to unknown environments and learn the transition matrices for MDPs built from LTL specifications. However, these learned models struggle to generalize to new specifications and are demonstrated on relatively small grid worlds. Paxton, Raman, Hager, *et al.* [126] introduce uncertainty during planning, but limit their planning horizon to 10 seconds, which is insufficient for the specifications explored here. Similarly, Carr, Jansen, and Topcu [127] synthesize a controller for verifiable planning in POMDPs using a recurrent neural network, yet are limited to planning in small grid worlds. In the next chapter, we address the limitations of these strategies, and introduce a novel approach for learning to guide hierarchical planners to solve long-horizon, complex tasks in partially revealed environments.

Chapter 3

Hierarchical Decision Making in the Presence of Uncertainty

In this chapter, we consider the problem of hierarchical planning in the case where our world model is incomplete due to the presence of environmental uncertainty. The particular form of uncertainty we address here involves scenarios where our robot is capable of building an accurate map—meaning the perception and dynamics models are very good—but that map is incomplete. This uncertainty is not due to failures of perception, or inaccuracy in learned models, but instead is a natural result of the limitations of what is physically possible. No matter how accurate our perception systems are, for example, no RGB camera can see through walls. Depending on the task we assign our robot, it may have to reason about entering and exploring unknown space, and so must account for this type of uncertainty when making decisions. In this chapter, we consider how to construct planning abstractions in these cases, accounting for the fact that the actions within our abstractions have uncertain outcomes.

The specific problem setting we consider in this chapter is as follows: our goal is to enable an autonomous agent to find a minimum cost solution to multi-stage planning tasks when the agent's knowledge of the environment is incomplete — i.e., when there are parts of the world the robot has yet to observe. As an example, imagine a robot tasked with extinguishing a small fire in a building. To do so, the agent could either find an alarm to trigger the building's sprinkler system, or locate a fire extinguisher, navigate to the fire, and put it out. We can represent sequential planning problems like this one in a number of ways, one of which is by specifying agent's goal in temporal logic, which has a long history of being utilized for planning in fully known environments (e.g., [104]–[108]). However, when the environment is initially unknown to the agent, efficiently planning to minimize expected cost to solve these types of tasks can be difficult.

Why is this problem challenging for our robots? Planning in partially revealed environments is difficult in part due to the complications associated with reasoning about unobserved regions of space. Consider our firefighting robot in a building it has never seen before, equipped with a sensor enabling it to build a map of what it sees. To clarify our focus, we will assume this sensor provides perfect perception of what is in direct line of sight from the robot's current position. Even with this idealized local sensing, the locations of any fires, extinguishers, and alarms may not be known. Therefore, the agent must consider all possible configurations of unknown space—including the position of obstacles—to find a plan that satisfies the task specification while ideally minimizing time spent executing that plan. As discussed in Chapter 2.3.1, this form of planning under uncertainty is fundamentally a Partially Observable Markov Decision Process (POMDP). We are able to simplify things in this case by modeling the world as a Locally Observable Markov Decision Process (LOMDP) [128], a special class of POMDP, which by definition includes our assumption of perfect range-limited perception. However, planning within even this somewhat simplified model requires access to a distribution over possible environment configurations, which will scale poorly as the complexity of the environment increases [85], [88], [129].

Since finding optimal policies for large POMDPs is extremely computationally intensive in general [129], planning to satisfy temporal logic specifications in full POMDPs is often limited to relatively small environments or time horizons [109], [110]. Thus, many approaches for solving tasks specified with temporal logic in larger, more realistic scenes often make simplifying assumptions about known and unknown space, or focus on maximizing the probability a specification is never violated [111]–[115]. Specifically, recognizing that we cannot reason about all possible configurations of unknown space, we might be tempted to assume that anything that has not yet been observed is free space, and immediately test our full plan for (an optimistic measure of) feasibility. Taking this approach, Ayala, Andersson, and Belta [111] attempt to solve specifications entirely in the known map, and if that is not possible, are guided by heuristics to a point on the boundary of known space to explore the environment. The unstated assumption in this technique is that any action which enters unknown space will succeed, and so the planner does not have to actively plan over the unobserved potions of the environment. Unsurprisingly, such strategies can result in suboptimal plans, as they do not recognize that the implicit planning abstraction does not satisfy the downward refinement property, nor do they consider the likelihood that entering a particular region of unknown space might fail to lead to the goal.

A number of methods use learned policies to minimize the cost of completing tasks specified using temporal logic [120], [121], [123], [125], [126]. However, due to the complexity of these tasks, many are again limited to fully observable small grid worlds [120], [121], [123], [125] or short time-horizons [126]. Moreover, for many of these approaches, changing the specification or environment requires retraining the entire system. This is a natural result of attempting to learn one function to guide the agent's search for a particular problem. If we then want to alter the task in some way, either by modifying the environment or changing the goal, whatever these models have learned may no longer be beneficial. For example, imagine that you have learned a policy which first finds a fire extinguisher, then leads the agent to the fire. If you suddenly change your goal to have your robot find a fire alarm, the learned policy will be inherently useless. Recent work by Stein, Bradley, and Roy [13] (which we will discuss in more detail later in this chapter) uses supervised learning to predict the outcome of taking actions through unknown space using the structure and appearance of the observed part of the environment. However, this work was restricted to goal-directed navigation and does not consider richer sets of tasks specified with temporal logic.

To address the challenges of planning over a distribution of possible futures that are impossible to refine at planning time, we introduce Partially Observable Temporal Logic Planner (PO-TLP). PO-TLP enables real-time planning for tasks specified in Syntactically Co-Safe Linear Temporal Logic (scLTL)[1] [130] in unexplored, arbitrarily large environments. To do this, we define an abstraction over a given task defined by an scLTL specification and a partially observed metric-semantic map. Since completing a task in this setting may not

[1]scLTL specifies tasks that can be accomplished in finite time.

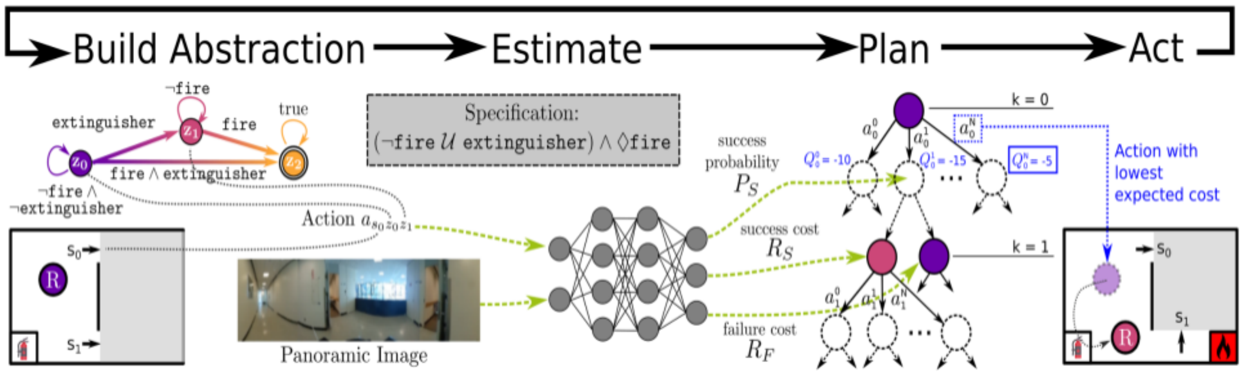


Figure 3.1: **POTLP Overview.** Given a specification (e.g., (¬`fire` $\mathcal{U}$ `extinguisher`) ∧ ◊`fire`), a Deterministic Finite Automaton (DFA) represents the high-level steps needed to accomplish the task. We can further define *subgoals* in the agent's environment with boundaries between observed (white) and unexplored (gray) space, and regions labeled with propositions relevant to the task, e.g., extinguisher and fire. **Build Abstraction:** We define *high-level actions* composed of *subgoals* and transition in task space, defining what the robot will attempt to do in known and unknown space. **Estimate:** For each possible action, we estimate its probability of success (P_S) and costs of success (R_S) and failure (R_F) using a neural network. **Plan:** We compute the expected cost of different sequences of actions utilizing these estimates within PO-UCT search. **Act:** The agent selects an action with the lowest expected cost, and moves along the path defined by that action, meanwhile receiving new observations, updating its map, and re-planning.

be possible in known space alone, we then define a set of dynamic *high-level actions* based on transitions between states in our task abstraction and available *subgoals* in the map—points on the boundaries between free and unknown space. We then approximate the full POMDP model such that actions either successfully make their desired transitions or fail, splitting future beliefs into two classes. By simplifying the planning problem in this way, we are able to more easily learn key statistics about the result of taking individual actions; specifically, we train a neural network from images to predict the costs and outcome of taking each action. These predictions allow us to avoid worrying about the specific state of the world (e.g., where exactly a fire extinguisher is located), and instead reason over only what is relevant to solving the given task at planning time, such as which direction down the hallway should be taken to find that extinguisher, irrespective of its exact pose. We use these models to guide a variant of Monte-Carlo Tree Search (PO-UCT [94]), which enables our agent to find the best high-level action for a given task and observation. Our model learns directly from visual input and can therefore be used across different novel environments and generalize over multiple tasks without retraining. After solving for the best action, our agent moves through observed space and continually re-plans as more of the environment is revealed, ensuring both that the specification will never be violated and that we return the optimal trajectory if the environment is fully known.

In the remainder of this chapter, we will present our approach for planning hierarchically in the presence of uncertainty as follows. First, we will discuss the intuition behind learning over subgoals to guide planning in partially revealed environments. To do this, we will consider how we might compute the cost of a goal-directed navigation task in this setting, and derive an equation to simplify that computation to enable tractable planning. We will then define the abstraction we use to encode the more complex planning problems we are interested in, next considering how we plan within this abstraction, specifically accounting for the fact that these high-level actions can fail depending on the unknown parts of the environment. Finally, we present how we train a neural network to predict the cost and outcomes of taking actions in this abstraction from visual input. We use these models to inform our high-level planner about low level information, enabling efficient top-down hierarchical planning. Once we have defined our approach, we apply PO-TLP to multiple tasks in simulation and the real world, showing improvement over a heuristic-driven baseline.

3.1 Planning over Subgoals for Complex Tasks

The aim of this chapter is to enable robots to efficiently solve complex tasks in partially revealed environments. We begin by defining a dense representation of the world, and a framework for decision making within that representation. We quickly see however that this presents an intractable planning problem for our agent, and so we propose an abstraction over actions to simplify the computation of cost for a plan within that abstraction. We subsequently derive an equation for computing expected cost within our abstraction, and propose an approach for planning to find the optimal action. First however, we must fully define the underlying, un-abstracted planning problem.

3.1.1 Defining our Problem

As discussed above, we can represent the challenge of solving tasks in the presence of uncertainty as a Partially Observable Markov Decision Process (POMDP). Here, we will define the specific instantiation of a POMDP (written as a Belief MDP) that we consider for this problem. In Chapter 2.3, we defined a template for POMDPs, which can be re-written as an n-tuple: $\mathcal{P} = (B, \mathcal{A}, P, R)$, each element of which we instantiate below. To specify these terms for a planning problem, we must first describe how we represent the world, as well as how we communicate a task to our robot.

Labeled Transition System (Encoding the Environment): We use a Labeled Transition System (LTS) to encode a discretized, metric-semantic representation of the robot's environment. An LTS $\mathcal{T}$ is a tuple $(X, x_0, \delta_{\mathcal{T}}, w, \Sigma, l)$, where: X is a discrete set of states of the robot and the world, $x_0 \in X$ is the initial state, $\delta_{\mathcal{T}} \subseteq X \times X$ is the set of possible transitions between states, the weight $w : (x_i, x_j) \to \mathbb{R}^+$ is the cost incurred by making the transition (x_i, x_j), Σ is a set of *propositions* with $\sigma \in \Sigma$, and the labeling function $l : X \to 2^{\Sigma}$ is used to assign which propositions are true in state $x \in X$. An LTS can be thought of as an occupancy grid, where each element in that grid has some semantic label. Like an occupancy grid, this label could indicate that the cell is occupied by an obstacle, though they might also have other labels. An example LTS is shown in Fig. 3.4A where $\Sigma = \{\texttt{fire}, \texttt{extinguisher}, \texttt{obstacle}\}$, $l(x_3) = \{\texttt{extinguisher}\}$ and $l(x_4) = \emptyset$ (indicating the cell is free-space). States $x \in X$ are comprised in part by a physical locations, and labels (like `fire` or `extinguisher`) indicate what is present at that location. States can have multiple labels when applicable, and can be labeled `unknown` when the space has not yet been observed. A finite trajectory through an LTS τ is a sequence of states $\tau = x_0, x_1, \ldots, x_n$ where $(x_i, x_{i+1}) \in \delta_{\mathcal{T}}$. From that sequence, we can generate a finite word $\omega = \omega_0, \omega_1, \ldots, \omega_n$ where each letter ω_i represents the labels found in the associated state $\omega_i = l(x_i)$. This finite word allows us to determine how the agent will interact with the world for a given trajectory. The question of how we might build metric-semantic representations like an LTS on real robots will be addressed in greater detail in Chapter 5.

Temporal Logic (Describing the Task): To specify tasks within an LTS, we use Syntactically Co-Safe Linear Temporal Logic (scLTL), a fragment of Linear Temporal Logic (LTL) [130]. Formulas in scLTL are written over a set of atomic propositions Σ with Boolean operators (negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$)) and temporal (next ($\bigcirc$), until ($\mathcal{U}$), eventually ($\Diamond$)). The syntax for scLTL is:

$$\varphi := \sigma \mid \neg\sigma \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \Diamond\varphi,$$

where $\sigma \in \Sigma$, and φ is an scLTL formula. The semantics are defined over infinite words $\omega^{\text{inf}} = \omega_0, \omega_1, \omega_2, \ldots$ with letters $\omega_i \in 2^{\Sigma}$. Intuitively, $\bigcirc\varphi$ is satisfied by a given ω^{inf} at step i if φ is satisfied at the next step $(i+1)$, $\varphi_1 \mathcal{U} \varphi_2$ is satisfied if φ_1 is satisfied at every step until φ_2 is satisfied, and $\Diamond\varphi$ is satisfied at step i if φ is satisfied at some step $j \geq i$. For the complete semantics of scLTL, refer to Kupferman and Vardi [130]. These semantics allow us to assign our robot tasks which enforce temporal constraints (i.e., we can specify which subtasks must be accomplished in what order), thus enabling the specification of complex goals. For example, if we want the firefighting robot to avoid fire until it reaches the extinguisher, and only then go to the fire, our task can be written as: $(\neg\texttt{fire}\,\mathcal{U}\,\texttt{extinguisher}) \wedge \Diamond\texttt{fire}$.

Deterministic Finite Automaton (Representing the Task): We would now like to consider how our robot might satisfy a given specification, and so seek to define a graphical representation of the task. Research in temporal logic verification and synthesis provides insight into this process [104], [105]. A Deterministic Finite Automaton (DFA), constructed from an scLTL specification, is given by the tuple $\mathcal{D}_{\varphi} = (Z, z_0, \Sigma, \delta_{\mathcal{D}}, F)$ [131]. A DFA is composed of a set of states, Z, with an initial state $z_0 \in Z$. Each $z_i \in Z$ can be thought of as a point in task space. The transition function $\delta_{\mathcal{D}} : Z \times 2^{\Sigma} \to Z$ takes in the current state, $z \in Z$, and a letter $\omega_i \in 2^{\Sigma}$, and returns the next state $z \in Z$ in the DFA. The DFA has a set of accepting states (representing the concept of a "goal" in task space), $F \subseteq Z$, such that if the execution of the DFA on a finite word $\omega = \omega_0\omega_1 \ldots \omega_n$ ends in a state $z \in F$, the word belongs to the language of the DFA. Stated differently, if we execute the DFA on a word ω, and that execution ends in an accepting state, then that word satisfies the task specification

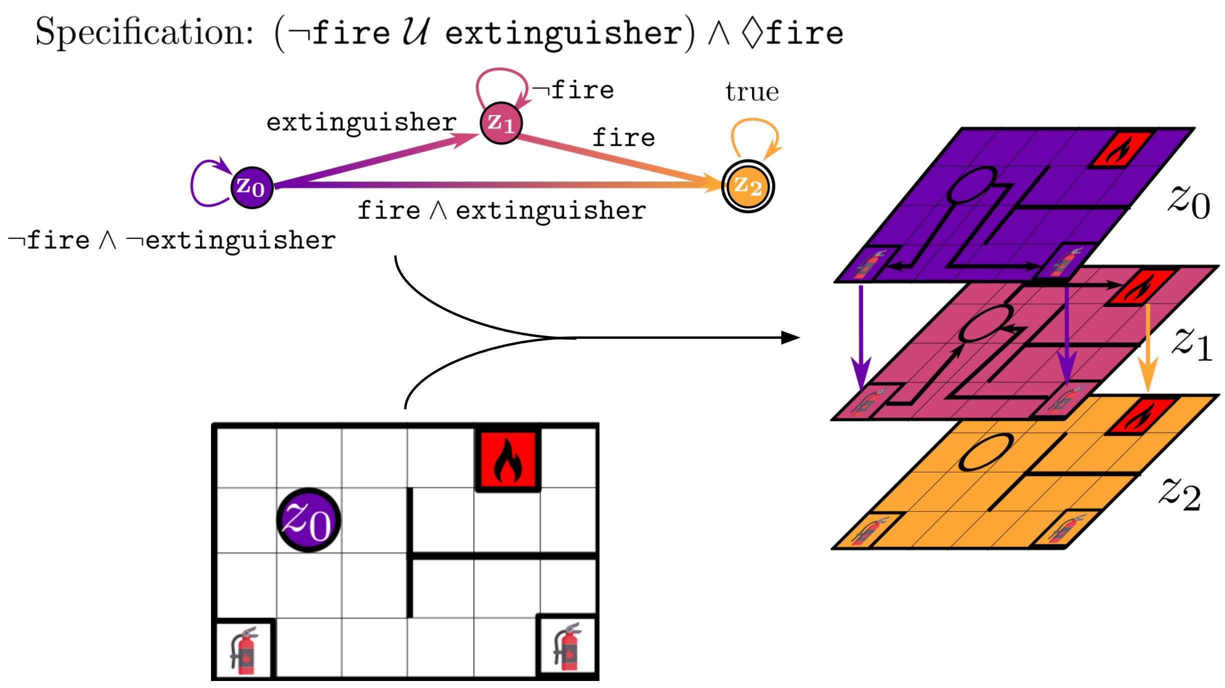

Specification: $(\neg\texttt{fire}\ \mathcal{U}\ \texttt{extinguisher}) \wedge \Diamond\texttt{fire}$
$\neg\texttt{fire}$
true
extinguisher
z_1
fire
z_0
z_2
$\texttt{fire} \wedge \texttt{extinguisher}$
$\neg\texttt{fire} \wedge \neg\texttt{extinguisher}$
z_0
z_0
z_1
z_2

Figure 3.2: A visualization of how a task specification can be converted to a Deterministic Finite Automaton (top). We then combine it with a Labeled Transition System (bottom) to form a Product Automaton (right). This representation fully defines the graph an agent must find a path through in order to accomplish the specified task in the given environment.

for the given initial state. While in general LTL formulas are evaluated over infinite words, the truth value of scLTL can be determined over finite traces. Fig. 3.2 shows the DFA for $(\neg\texttt{fire}\ \mathcal{U}\ \texttt{extinguisher}) \wedge \Diamond\texttt{fire}$.

Product Automaton (Combining Task and Environment): We would like to be able to plan to solve a specified task in a given environment, and so require a single representation in which to find such a plan. A Product Automaton (PA) is a tuple $(P, p_0, \delta_P, w_P, F_P)$ which captures the combined behavior of the DFA and LTS. The states $P = X \times Z$ keep track of both the LTS and DFA states, where $p_0 = (x_0, z_0)$ is the initial state. A transition between states is possible iff the transition is valid in both the LTS and DFA, and is defined by $\delta_P = \{(p_i, p_j) \mid (x_i, x_j) \in \delta_{\mathcal{T}}, \delta_{\mathcal{D}}(z_i, l(x_j)) = z_j\}$. When the robot moves to a new state $x \in X$ in the LTS, it transitions to a state in the DFA based on the label $l(x)$. The weights on transitions in the PA are the weights in the LTS ($w_P(p_i, p_j) = w(x_i, x_j)$). Accepting states in the PA, F_P, are those states with accepting DFA states ($F_P = X \times F$). We are able to plan directly in the known portion of this representation to satisfy our robot's task for a given environment. Figure 3.2 shows how we combine a DFA and LTS to form a PA.

Planning in Partially Observable Product Automata: Remember that our agent will be tasked with solving complex tasks in environments that have not yet been fully explored. To represent the problem of planning through a partially revealed PA, we model the world as a Locally Observable Markov Decision Process (LOMDP) [128], a POMDP [132] where space within line of sight from our agent is fully observable. Here we define the full tuple of the associated belief MDP.

1. **Beliefs** $B = \{B_{\mathcal{T}} \times B_X \times B_Z\}$: We can think of each belief state as distributions over three entities $b = \{b_{\mathcal{T}}, b_x, b_z\} \in B$: the environment (abstracted as an LTS) $b_{\mathcal{T}}$, the agent's position in that environment b_x, and the agent's progress through the DFA b_z defined by the task. Belief $b_{\mathcal{T}} \in B_{\mathcal{T}}$ defines a distribution over the set of labeled transition systems. Because we assume known space is perfectly observed, each cell is given a label such as `free`, `occupied`, `FIRE`, etc. If the cell has not yet been observed, it is labeled `unknown`, allowing us to treat $b_{\mathcal{T}}$ as a labeled occupancy grid that is revealed as the robot explores. In principle, we could similarly define uncertainty in the robot's position in the map, uncertainty in the given task, or uncertainty in the progress through that task. However, with our assumption of perfect sensing in the LOMDP formulation, we collapse b_x and b_z to point estimates, meaning we know the state of the robot and what it has done in terms of its given task exactly.

2. **Actions** $\mathcal{A}$: The set of available actions from a given state in the PA, which at the lowest level of abstraction are the feasible transitions given by δ_P. Thus, from any state the agent has 8 available low-level actions, which move it through its PA.

3. **Transition Probabilities** P: $P(b_{t+1}|b_t, a_t)$ defines the probability that taking an action a_t from belief b_t transitions the agent to belief b_{t+1}. For actions in known space, we know the label of the region the agent is moving, and so know with certainty both its position in both physical and task space. However, when entering unknown space, while the location

of the next state is known, the label of that state is not, and depends on the distribution over environments in the belief.

4. **Rewards** R: $R(b_{t+1}, b_t, a_t)$ defines the instantaneous reward of taking action a_t to get from belief b_t to b_{t+1}. Here we formulate reward as a cost, meaning we assign negative reward according to the weight in the LTS. As a result, the optimal plan will be the one which transitions the agent to an accepting state in the PA in as few actions as possible. We assign an arbitrarily high cost to actions which move the agent into a region that is occupied by an obstacle or which violate the specification, ensuring we never find plans which lead to a failed plan.

Using the POMDP model, we can represent the expected cost of taking an action using a belief-space variant of the Bellman equation [132], as derived in Chapter 2.3:

$$Q(b_t, a_t) = \sum_{b_{t+1}} P(b_{t+1}|b_t, a_t)\Big[R(b_{t+1}, b_t, a_t) + \min_{a_{t+1}\in\mathcal{A}(b_{t+1})} Q(b_{t+1}, a_{t+1})\Big], \tag{3.1}$$

where $Q(b_t, a_t)$ is the cost of taking action a_t given belief b_t. This equation can, in theory, be solved to find the optimal action for any belief. However, given the size of the environments we are interested in, directly updating the belief in evaluation of Eq. (3.1) is intractable for two reasons. First, due to the *curse of dimensionality*, the size of the belief grows exponentially with the number of states. In the POMDP described above, this corresponds to the number of cells in the unobserved portion of the map, and is therefore quite large. Second, by the *curse of history*, the number of iterations needed to solve Eq. (3.1) grows exponentially with the planning horizon. Given that each action is is a single transition between adjacent cells, in large environments with a high resolution, we may require hundreds of actions to reach our goal. Of course, solving this equation also requires access to a distribution over all possible environments the agent might encounter, which is impossible to obtain for any real world setting (as mentioned in Fig. 3.3). For these reasons, planning to satisfy arbitrary specifications for the POMDP defined above using Eq. (3.1) is intractable for large environments, and we therefore require a simplifying abstraction.

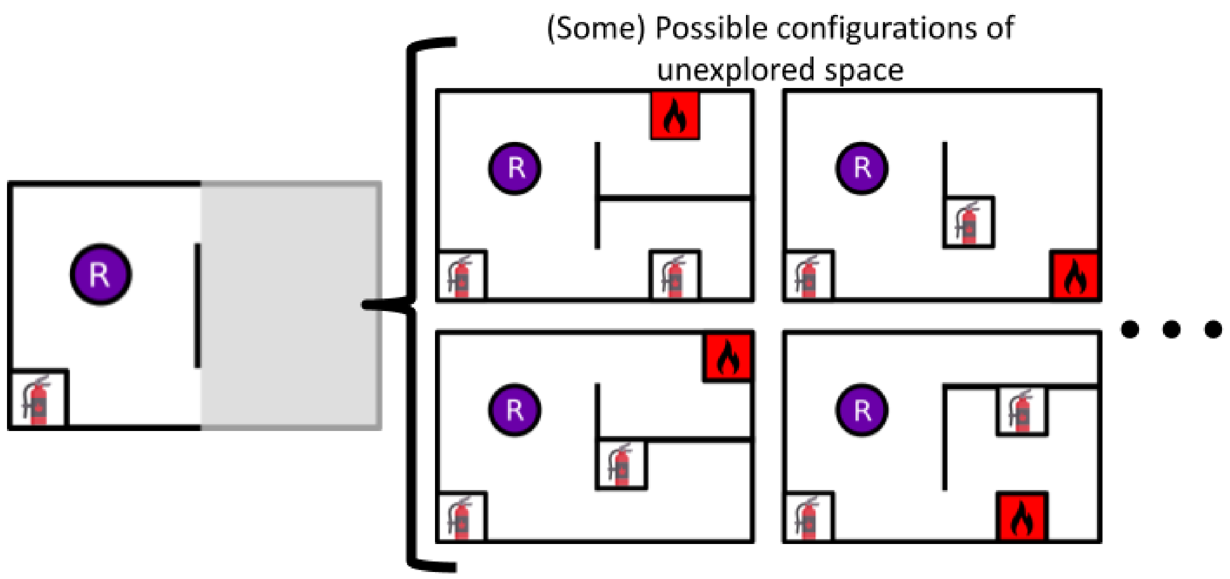


Figure 3.3: Attempting to predict the exact orientation of unknown space is an impossible problem. We cannot assume access to a distribution over possible environments, and even if we could, there are far to many such configurations to tractably reason about.

3.1.2 Defining an Abstraction over Actions and Outcomes

To overcome the challenges associated with solving Eq. (3.1), in this section we present an abstraction first over actions, then the outcomes of taking those actions. As was discussed in Chapter 1, humans do not constrain themselves to plan at the level of moving between adjacent cells in labeled occupancy grid. Instead, we can recognize that trajectories which enter unknown space can be grouped by the topology of the environment (e.g., going left or right down the hallway). We take inspiration from that insight, and notice that the structure of an LTS allows for the identification of frontiers in our map, each representing a contiguous boundary between free and unknown space. Similarly, we recognize that humans do not move through the world aimlessly, but are driven to accomplish some goal. Progressing toward the robot's goal requires making transitions in the DFA. For example, given the specification $(\neg\texttt{fire}\ \mathcal{U}\ \texttt{extinguisher}) \wedge \Diamond\texttt{fire}$ (as in Figures 3.1, 3.2, and 3.4), the robot must first retrieve the extinguisher and then reach the fire. Depending on what has been seen by the robot, some or all of this task might be achievable in the known map, however, if the task cannot be completed entirely within known space, the robot must reason about acting in areas that it has not yet explored. As such, our action set is defined by *subgoals* $\mathbb{S}$ in physical space and transitions in task space. Specifically, when executing an action, the robot first plans through the PA in known space to a point on a frontier, and then enters the unexplored space beyond that subgoal to attempt to transition to a new state in the DFA.

For belief state b_t, action $a_{sz'z''}$ defines the act of traveling from the current state in the LTS x to reach subgoal $s \in \mathbb{S}_t$ at a DFA state z', and then attempting to transition to DFA state z'' in unknown space. Our newly defined set of possible actions from belief state b_t is therefore:

$$\mathcal{A}(b_t) = \{a_{sz'z''} \mid s \in \mathbb{S}_t, z' \in Z_{\text{reach}}(b_t, s), z'' \in Z_{\text{next}}(z')\}, \tag{3.2}$$

where $Z_{\text{reach}}(b_t, s)$ is the set of DFA states that can be reached while traveling in known space in the current belief b_t to subgoal $s \in \mathbb{S}_t$, and $Z_{\text{next}}(z') = \{z'' \in Z \mid \exists \omega_i \in 2^{\Sigma} \text{ s.t. } z'' = \delta_{\mathcal{D}}(z', \omega_i)\}$ is the set of DFA states that can be visited in one transition from z'. Each action therefore represents what the robot does in known space, where it enters unknown space, and what it attempts to do upon entering unknown space. Fig. 3.4B illustrates an example of available high-level actions for a given task and environment.

Equation 3.2 defines a set of abstract actions which has the potential to greatly reduce our planning horizon (solving the specification in Fig.3.4 requires only two sequential successful actions). However, we have not yet considered how taking one of these actions affects the agent's belief according to Eq. 3.1. Let $Z(b_{t+1})$ refer to b_z (the component of the belief related to progress through task space) at the next time step, and $b_{t+1} \in B_{z''}$ define the subset of beliefs s.t. $Z(b_{t+1}) = z''$. Without loss of generality, we can split future belief states b_{t+1} into two classes—futures where a given action is successful, $Z(b_{t+1}) = z''$, and futures where it is not: $Z(b_{t+1}) = z'$. Note that this abstraction over the belief space does not exist in general, and is enabled by our assumption of perfect local perception. Under an optimistic assumption that all actions are successful, every future state falls into the first bucket. However, this is of course not the case in reality, and by considering the case when actions fail, we account for the fact that our new action set is not downward-refineable. We can represent the Bellman equation with our factored belief accordingly:

$$
\begin{aligned}
Q(b_t, a_t \in \mathcal{A}(b_t)) = P_S(b_{t+1} \in B_{z''}|b_t, a_t) \sum_{b_{t+1} \in B_{z''}} P_{Z''}(b_{t+1}|b_t, a_t) \left[R(b_{t+1}, b_t, a_t) + \min_{a \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a) \right] + \\
(1 - P_S(b_{t+1} \in B_{z''}|b_t, a_t)) \sum_{b_{t+1} \notin B_{z''}} P_{Z'}(b_{t+1}|b_t, a_t) \left[R(b_{t+1}, b_t, a_t) + \min_{a \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a) \right],
\end{aligned}
\tag{3.3}
$$

where $P_S(b_{t+1} \in B_{z''}|b_t, a_t) \equiv \sum_{b_{t+1} \in B_{z''}} P(b_{t+1}|b_t, a_t)$ is the proportion of states in which the agent successfully reaches the goal after selecting action a_t from belief state b_t. We further define the following term:

$$
P_{Z''}(b_{t+1}|b_t, a_t) \equiv \frac{P(b_{t+1}|b_t, a_t)}{P_S(b_{t+1} \in B_{z''}|b_t, a_t)},
\tag{3.4}
$$

which is normalized according to $\sum_{b_{t+1} \in B_{z''}} P_{z''}(b_{t+1}|b_t, a_t) = 1$. Thus, the first term in Eq. (3.3) can be thought of as the expected cost over actions that succeed in reaching the intended state in the DFA times the proportion of actions that succeed (both of which, we later estimate using learning). $P_{z'}$ is defined similarly over states that do not reach the goal.

To ground our thinking briefly, let us attempt to compute the cost of taking a single action $a_{sz'z''} \in \mathcal{A}$. When executing action $a_{sz'z''}$, the robot reaches subgoal s in DFA state z', accumulating a cost $D(b_t, a_{sz'z''})$ which can be determined using Dijkstra's algorithm in the known map. In Eq. 3.3, this cost exists for both the case where our action succeeds or fails, so we can pull it outside each summation. Once the robot enters the unknown space beyond the subgoal, the action has some probability $P_S(b_{t+1} \in B_{z''}|b_t, a_x z'z'')$ of successfully transitioning from z' to z''. Furthermore (remembering to subtract the cost in known space), each action also has an expected cost of success $R_S(b_t, a_{sz'z''})$, and expected cost of failure $R_F(b_t, a_{sz'z''})$ in unknown space such that:

$$
\begin{aligned}
R_S(b_t, a_{sz'z''}) &\equiv \sum_{b_{t+1} \in B_{z''}} P_{Z''}(b_{t+1}|b_t, a_t) R(b_{t+1}, b_t, a_t) - \frac{D(b_t, a_{sz'z''})}{P_S(b_{t+1} \in B_{z''}|b_t, a_t)}, \\
R_F(b_t, a_{sz'z''}) &\equiv \sum_{b_{t+1} \notin B_{z''}} P_{Z'}(b_{t+1}|b_t, a_t) R(b_{t+1}, b_t, a_t) - \frac{D(b_t, a_{sz'z''})}{1 - P_S(b_{t+1} \in B_{z''}|b_t, a_t)}.
\end{aligned}
\tag{3.5}
$$

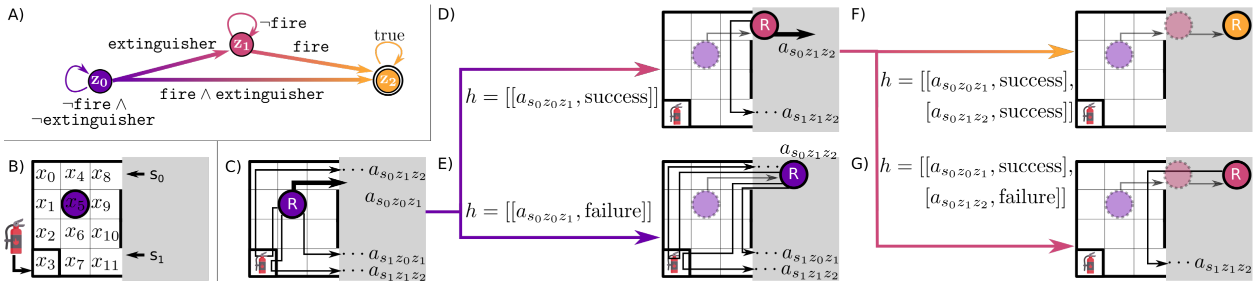

A)
extinguisher
¬fire
fire
true
¬fire ∧ ¬extinguisher
fire ∧ extinguisher
B)
s₀
s₁
C)
D)
E)
F)
G)
R
$h = [[a_{s_0 z_0 z_1}, \text{success}]]$
$h = [[a_{s_0 z_0 z_1}, \text{failure}]]$
$h = [[a_{s_0 z_0 z_1}, \text{success}], [a_{s_0 z_1 z_2}, \text{success}]]$
$h = [[a_{s_0 z_0 z_1}, \text{success}], [a_{s_0 z_1 z_2}, \text{failure}]]$
$a_{s_0 z_1 z_2}$
$a_{s_0 z_0 z_1}$
$a_{s_1 z_0 z_1}$
$a_{s_1 z_1 z_2}$

Figure 3.4: **High-level actions.** Our agent is attempting to satisfy the specification $(\neg\texttt{fire}\ \mathcal{U}\ \texttt{extinguisher}) \wedge \Diamond\texttt{fire}$ with the DFA in (A). The robot ("R") is in a partially explored environment with two subgoals s_0 and s_1 between observed (white) and unexplored (gray) space that is abstracted into an LTS (B). As the robot considers high-level actions at different stages of its plan (C), its color indicates belief of the DFA state (according to (A))). In this rollout, the robot considers action $a_{s_0 z_0 z_1}$, and outcomes where it succeeds (D), and fails (E). If successful, the robot has two actions available and considers $a_{s_0 z_1 z_2}$, which in turn could succeed (F) or fail (G).

Rather than computing these values explicitly, they are estimated via learning given an encoding of the action and visual input as discussed in Section 3.2. We can now express the expected instantaneous cost of an action as:

$$\sum_{b_{t+1}} P(b_{t+1}|b_t, a_{sz'z''})R(b_{t+1}, b_t, a_{sz'z''}) =$$
$$D(b_t, a_{sz'z''}) + P_S(b_{t+1} \in B_{z''}|b_t, a_t)R_S(b_t, a_{sz'z''}) + (1 - P_S(b_{t+1} \in B_{z''}|b_t, a_t))R_F(b_t, a_{sz'z''}). \tag{3.6}$$

Evaluating the recursive term: $\min_{a \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a)$ in Eq. (3.3) however, remains particularly difficult to compute.

3.1.3 Estimating Future Cost using High-Level Actions

Computing the sequence of actions with minimum expected cost according to Equation 3.3 requires grappling with evaluating the expected cost after the first action is taken. This term $(\min_{a \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a))$ depends upon the agent's belief at the next time-step b_{t+1}, the computation of which requires maintaining a distribution over the environment, $b_{\mathcal{T}}$. Of course, making an informed guess about what unknown space might look like after taking an action, then taking an expectation over all possible configurations based on that prior is infeasible. Instead, we reduce this term by making a simplifying assumption: that we can estimate the expected cost of future states from the current known map $\mathcal{T}_{\text{known}}$. As a result, our belief at time t can now be represented as $b_{t+1} = \{\mathcal{T}_{\text{known}}, x_{t+1}, z_{t+1}\}$, where the second and third terms represent the position of the robot in physical and task space respectively. We therefore do not explicitly model the evolution of the map state as we compute the expected cost of an action, meaning the set of subgoals $\mathcal{S}$ available for future actions does not change after an action is taken.

In order to compute the cost of an action in a sequence, we must know the results of the actions that came before it. Since we cannot tractably update and maintain a distribution over maps during a complete simulated trial (referred to as a *rollout*), we instead keep track of the *rollout history* $h = [[a_0, o_0], \ldots, [a_n, o_n]]$. A *rollout history* is defined as a sequence of high-level actions a_i considered during planning and their simulated respective outcomes $o_i = \{\texttt{success}, \texttt{failure}\}$. This form allows us to determine the belief $b_t = \{\mathcal{T}_{\text{known}}, x_t, z_t\}$ as defined above for any t, and therefore the available set of actions $\mathcal{A}(b_t)$. During a rollout, the set of available future actions is further informed by actions and outcomes already considered in h. For example, if we simulate an action in a rollout, and it fails, we should not consider that action a second time. Conversely, we know a successful action would succeed if simulated again in that rollout. In Fig. 3.4-G, the robot imagines its first action $a_{s_0 z_0 z_1}$ succeeds, while its next action $a_{s_0 z_1 z_2}$ fails, making the rollout history $h = [[a_{s_0 z_0 z_1}, \texttt{success}], [a_{s_0 z_1 z_2}, \texttt{failure}]]$. When considering the next step of this rollout, the robot knows it can always find an extinguisher beyond s_0, and there is no fire beyond s_0. To track this information during planning, we define a *rollout history-augmented belief* $b_h = \{\mathcal{T}_{\text{known}}, x, z, h\}$, which augments the belief with the actions and outcomes of the rollout up to that point. To reiterate, we maintain the history-augmented belief b_h only during planning, to avoid the complexity of maintaining a distribution over possible future maps from possible future observations during rollout.

Using b_h, we also define a *rollout history-augmented action set* $\mathcal{A}(b_h)$, in which actions known to be impossible based on h are pruned, and with it, a *rollout history-augmented success probability* $P_S(b_h, a_{sz'z''})$ which is identically one for actions known to succeed. Furthermore, because high-level actions involve entering unknown space, instead of explicitly considering the distribution over possible robot states, we define a *rollout history-augmented distance* function $D(b_h, a_{sz'z''})$, which takes into account the physical location of the robot as a result of taking the last action in b_h. If an action leads to a new subgoal ($s_{t+1} \neq s_t$), the agent accumulates the success cost R_S of the previous action if that action was simulated to succeed and the failure cost R_F if it was not.

By planning with b_h, the future expected reward can be written so that it no longer directly depends on the full future belief state b_{t+1}, allowing us to approximate it as follows:

$$\begin{aligned}\sum_{b_{t+1}} P(b_{t+1}|b_t, a_{sz'z''}) \min_{a' \in \mathcal{A}(b_{t+1})} Q(b_{t+1}, a') \approx \\ P_S(b_h, a_{sz'z''}) \min_{a_S \in \mathcal{A}(b_{h_S})} Q(b_{h_S}, a_S) + [1 - P_S(b_h, a_{sz'z''})] \min_{a_F \in \mathcal{A}(b_{h_F})} Q(b_{h_F}, a_F),\end{aligned} \tag{3.7}$$

where $h_S = h.\texttt{append}([a_{sz'z''}, \texttt{success}])$ is the rollout history conditioned on a successful outcome (and h_F is defined similarly for failed outcomes). We now combine Eq. (3.6) and Eq. (3.7) with Eq. (3.3), and represent the full Bellman equation as follows:

$$\begin{aligned}Q(b_h, a_{sz'z''}) = D(b_h, a_{sz'z''}) + \underline{P_S(b_h, a_{sz'z''})} \times \left[\underline{R_S(b_h, a_{sz'z''})} + \min_{a_S \in \mathcal{A}(b_{h_S})} Q(b_{h_S}, a_S)\right] \\ \underline{\textit{Underline denotes terms estimated via learning}} \quad + \left[1 - \underline{P_S(b_h, a_{sz'z''})}\right] \times \left[\underline{R_F(b_h, a_{sz'z''})} + \min_{a_F \in \mathcal{A}(b_{h_F})} Q(b_{h_F}, a_F)\right],\end{aligned} \tag{3.8}$$

where each term is fully redefined below:

1. $b_h = \{\mathcal{T}_{\text{known}}, x, z, h\}$: The current history augmented belief of the agent, defined by the known map $\mathcal{T}_{\text{known}}$, the agent's position in physical space x, it's position in task space z, and the history of actions and outcomes simulated to this point in the plan h.

2. $D(b_h, a_{sz'z''})$: the cost of traveling from x, the current location of the agent, to a subgoal s (as specified by an action). For the first action in a rollout, determining this cost is as simple as planning the shortest cost path through the PA from the robot to the chosen subgoal. Note this shortest path may require making transitions in the DFA depending on the action. Since this motion plan occurs entirely within known space, we use Dijkstra's

algorithm as a heuristic to calculate this cost. In the recursive part of this cost function, if the subsequent subgoal is the same as the previous one (e.g., we try to find the fire beyond the same frontier that we found the extinguisher), we set this value to 0. If however the subgoal differs from previous one in h, $D(b_h, a_{sz'z''})$ also incorporates the cost of returning to known space after taking the previous action. We approximate this additional cost by augmenting D with either R_S or R_F (depending on outcome), making an implicit assumption that the cost of returning to known space is approximately the same as the cost of success or failure of the previous action.

3. $P_S(b_h, a_{sz'z''})$: the probability that the chosen action will succeed given the current belief.

4. $R_S(b_h, a_{sz'z''})$: the expected cost of success. Suppose the robot successfully transitions to z'' via the subgoal specified by action $a_{sz'z''}$. Then, with probability P_S, we accrue an expected cost of successfully executing this transition in the unknown space beyond s.

5. $R_F(b_h, a_{sz'z''})$: the failure cost. In the event that the agent does not make it's desired transition, we say that it instead *explores* the space beyond the subgoal of interest. With probability $1 - P_S$, the agent then accumulates the expected cost of trying to make this transition beyond subgoal s, therefore remaining in z'.

6. $Q(b_{h_O}, a_O)$: the future expected cost for outcome O. Whether the robot either succeeds ($O = S$) or fails ($O = F$) in executing it's chosen action, this represents the cost of considering a different action.

This factorization of the Bellman equation presents a much shorter horizon problem then we previously faced. Moreover, our agent is no longer expected to reason about the distribution of all possible PAs, thereby reducing the computational complexity of solving for the best action. As highlighted above, many of these terms are estimated via learning, which will be discussed in Section 3.2. In the following section however, we more carefully consider how we solve for the minimum expected cost sequence of actions according to our factored Bellman equation. This will allow us to guide our decision making within the high-level abstraction we have defined with the low-level information we pass to our learned models, enabling efficient hierarchical planning.

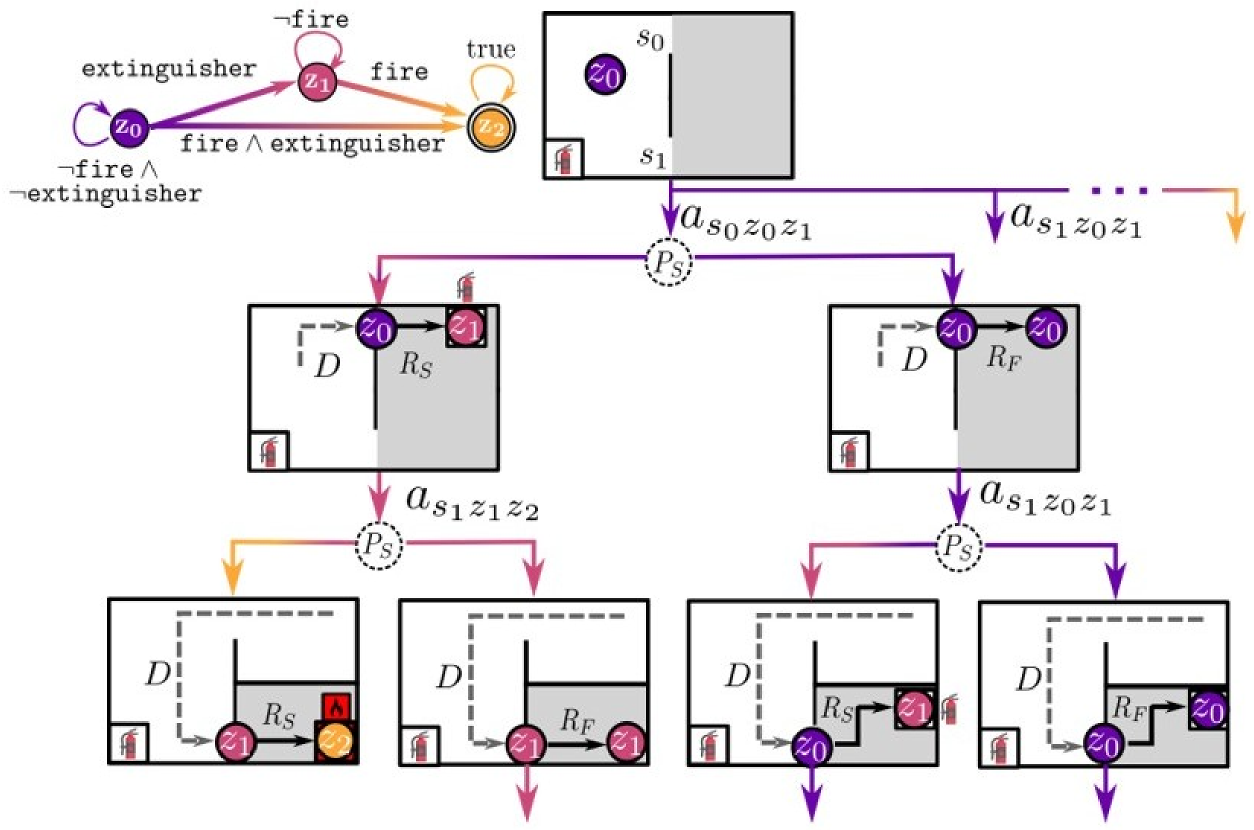


Figure 3.5: A subset of a full search tree with our abstract actions and binary outcomes. We highlight the different elements of cost, which vary based on the outcome (success or failure) of a given action. Equation (3.8) compactly represents this cost. For the simple environment shown here, we can completely enumerate all possible actions and search over their different orderings. However, as we scale up in task and environmental complexity, computing this equation exactly becomes difficult.

3.1.4 Planning with High-Level Actions using PO-UCT

Equation (3.8) demonstrates how expected cost can be computed using our high-level actions, given estimated values of P_S, R_S, and R_F. For example, if our task were simple goal directed navigation (e.g., find the building exit), the only actions available to agent would be to attempt to reach this goal beyond each subgoal. For a reasonable set of frontiers, we can consider every possible ordering of actions, compute the cost of each according to Eq. (3.8), and solve for the optimal ordering. The work in Stein, Bradley, and Roy [13] demonstrates this, though even in the experiments highlighted in that work, the approach shows some limitations. As environments scale, the number of subgoals increase, and the authors relied upon heuristics to prune the agent's action set. In our setting, depending on the task, each subgoal might be associated with many different actions. Moreover, several actions may need to succeed for the specification to be satisfied, severely increasing the computational burden—exponential in both the number of subgoals and the size of the DFA. We see this expanding tree in figure 3.5.

Instead of attempting this exhaustive search, we adapt Partially Observable UCT (PO-UCT) [94], a generalization of Monte-Carlo Tree-Search (MCTS) which tracks histories of actions and outcomes, to select the best action for a given belief using sampling. The nodes of our search tree correspond to belief states b_h, and the actions available at each node are defined according to the rollout history as discussed in Section 3.1.3. The complete search procedure is as follows:

1. Tree traversal: Beginning from the root node, we traverse the search tree according the the UCT equation:

$$a^* \leftarrow \underset{a \in \mathcal{A}(b_h)}{\arg\max}\, V(ha) + c\sqrt{\frac{2ln(N(h))}{N(ha)+1}}, \tag{3.9}$$

   where $V(ha)$ represents the average cost seen at after taking action a from history h. $N(h)$ is the visitation count of the current node having been reached via h, and $N(ha)$ represents the number of times action a has been taken from the current node. c is an exploration constant, intended to balance exploiting what the search has seen thus far, and exploring new parts of the tree. Upon selecting an action a, we sample an outcome o according to the Bernoulli distribution parameterized by P_S. We then continue this traversal process until we sample an outcome we have not yet observed.

2. Expansion: Upon sampling a new outcome, we augment the history with the sampled action and outcome hao, and add a new node to our tree representing this history.

3. Rollout: Next, we identify every action which might be taken from this new node, and do the following. 1) randomly sample an action from $\mathcal{A}(b_h)$ and outcome according to P_S, accruing cost by R_S or R_F depending on that outcome. Note, we also sample the actions weighted by their likelihood of success as a heuristic. We continue sampling actions and outcomes until the specification is satisfied (or we run out of actions).

4. Back Propogation: We then update the statistics of all nodes in the full history. Specifically, we tabulate the number of times each node has been visited, and the average cost of all such histories.

Search continues in this manner until a pre-specified timeout is reached, and we return the optimal action sequence to our agent to execute. PO-UCT search prioritizes the most promising branches of the search space, avoiding the cost of enumerating all potential histories. By virtue of being an anytime algorithm, PO-UCT also enables budgeting computation time, allowing for faster online execution as needed on a real robot. Once our agent has searched for the action with lowest expected cost, $a^*_{sz'z''}$, it generates a motion plan through space to the subgoal associated with the chosen action. While moving along this path, the agent receives new observations of the world, updates its map, and replans (see Fig. 3.1 and Algorithm 1). Because our agent is constantly updating it s map and replanning, we are guaranteed to never take an action which violates our specification (or collides with an obstacle). Moreover, as the map is revealed, assuming the task can be solved, eventually our agent will observe enough of the environment that a goal state can be reached entirely within known space.

Algorithm 1 PO-TLP

```
Function PO-TLP($\theta$):                                  // $\theta$: Network Parameters
    $b \leftarrow \{b_{\mathcal{T}_0}, b_{x_0}, b_{z_0}\},\ Img \leftarrow Img_0$
    while True do
        if $b_z \in F$ then                            // If in accepting state: SUCCESS
            return SUCCESS

        $a^*_{sz'z''} \leftarrow$ HighLevelPlan($b, Img, \theta, \mathcal{A}(b)$)
        $b,\ Img \leftarrow$ ActAndObserve($a^*_{sz'z''}$)

Function HighLevelPlan($b,\ Img,\ \theta,\ \mathcal{A}(b)$):
    for $a \in \mathcal{A}(b)$ do
        $\underline{a.P_S}, \underline{a.R_S}, \underline{a.R_F} \leftarrow$ EstProps($b, a, Img \mid \theta$)
    $a^*_{sz'z''} \leftarrow$ PO-UCT($b, \mathcal{A}(b)$)

    return $a^*_{sz'z''}$
```

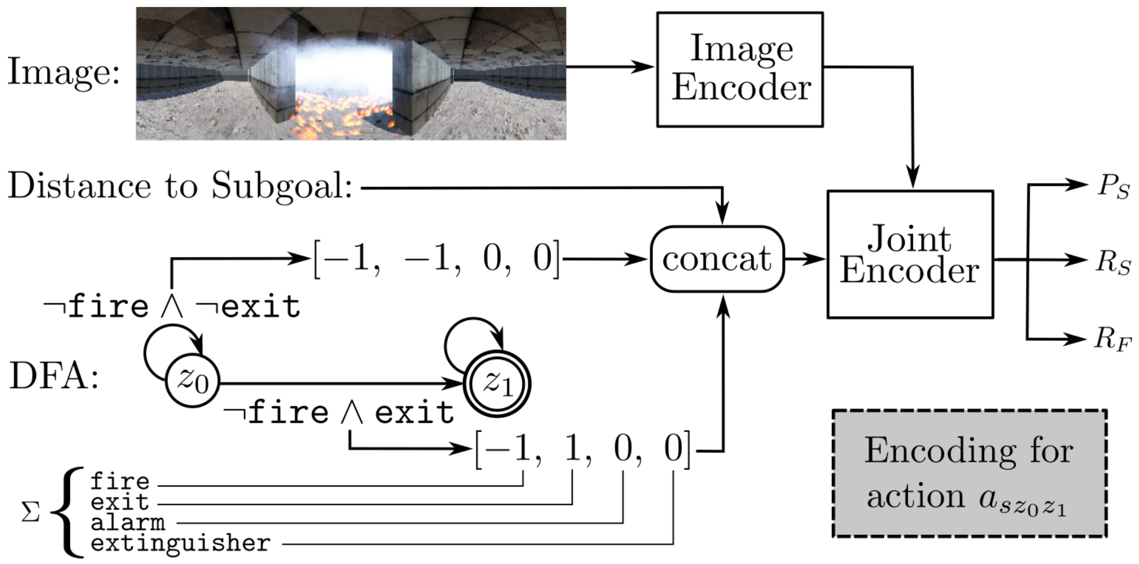


Figure 3.6: **Neural network inputs and outputs.** Estimating P_S, R_F, and R_S for action $a_{sz_0z_1}$: attempting to reach an exit while avoiding fire.

3.2 Learning Transition Probabilities and Costs

In order to plan according to the approach outlined above, we rely on values for P_S, R_S, and R_F for arbitrary actions $a_{sz'z''}$. Computing these values explicitly from the belief (as defined in Section 3.1.2) is intractable, so we train a neural network to estimate them from visual input and an encoding of a selected action.

3.2.1 Encoding a Transition

A successful action $a_{sz'z''}$ results in the desired transition in the DFA from z' to z'' occurring in unknown space. However, encoding actions directly using DFA states would result in a model that requires retraining from scratch every time we want to change the agent's task specification. Instead, we define an encoding that represents formulas over the set of propositions Σ. These formulas, which we represent in negative normal form [133], are defined over the truth values of Σ, and thus generalize over any specification written with Σ in similar environments.

To progress from z' to z'' in unknown space, the robot must travel such that it remains in z' until it realizes changes in proposition values that allow it to transition to z''. We therefore define two n-element feature vectors $[\phi(z', z'), \phi(z', z'')]$ where $\phi \in \{-1, 0, 1\}^n$, which serve as input to our neural network. The first vector defines the predicates which must hold to remain in z', while the second gives those which must hold to transition to z'' For the agent to stay in z', if the ith element in $\phi(z', z')$ is 1, the corresponding proposition must be true at all times; if it is -1, the proposition must be false; and if it is 0, the proposition has no effect on the desired transition. The values in $\phi(z', z'')$ are defined similarly for the agent to transition from z' to z''. Fig. 3.6 illustrates this feature vector for a task specification example. These vectors fully encode the information needed to define a transition, without requiring information specific to a task. Because this representation is not dependent on the whole task, we are able to utilize the same network with different task specifications, as long as the agent has been trained on data with the relevant predicates.

3.2.2 Network Architecture and Training

Our network takes as input a 128×512 RGB panoramic image centered on a subgoal, the scalar distance to that subgoal, and the two n-element feature vectors ϕ describing the transition of interest, as defined in Section 3.2.1. The input image is first passed through an image encoder of 4 convolutional layers, each followed by a batch normalization operation, a ReLU activation function, and a max-pooling operation. Next, we concatenate the feature vectors and a vector containing the distance to the subgoal to each element (augmenting the number of channels accordingly), and continue encoding jointly for 8 more convolutional layers. Finally, the encoded features are passed through 5 fully connected layers, and the network outputs the properties required for Eq. (3.8)—P_S, R_S, and R_F.

We trained our network using the Adam optimizer in Pytorch with default parameters over 100k steps with a batch size of 256 and using roughly 300k training samples for each environment. The learning rate begins at 0.004 and decreases by a factor of 0.9 every 5k steps, which regularizes the network and reduces over-fitting. We trained a few dozen different network architectures with variations on the number and size of the layers, and found minimal impact on the network's effectiveness. The output of the neural network has three dimensions, for each of the three desired outputs (P_S, R_S, and R_F); a sigmoid activation and weighted cross-entropy loss is used to predict the success probability P_S, while a linear activation and a squared error loss is used for the two regression outputs R_S and R_F. Loss for the success cost is only applied when a plan through a subgoal leads to the desired transition and the loss for the failure cost is only applied when it does not (Eq (3.10)).

An additional consideration of our training procedure is that the effective cost of mis-classifying a subgoal is variable and strongly depends on a number of factors. For example, if an action constitutes the only route to the goal, then the penalty for mis-classifying this action as unlikely to succeed is very high. The robot may be forced to explore every other option before returning. By contrast, mis-classifying in the other direction (a false positive) is therefore of relatively low cost. Computing the relative importance of correctly classifying an action would require solving the full Bellman equation and is therefore too expensive to compute. Instead, we utilize the approach in Stein, Bradley, and Roy [13], and provide a heuristic for computing this *misclassification cost* w_M at training time. For actions which *do not* lead to the goal, improper classification results in the exploration cost R_F described above, which is roughly the cost of exploring the space beyond the chosen subgoal. For actions that *do* lead to the goal, the robot might have to consider every other possible action, potentially traversing across the entire map before returning; therefore, the penalty is the cost of traveling to the furthest point beyond a subgoal that does not lead to the goal and back. The computed costs are then used as weights for the cross-entropy loss used to train the classifier. This heuristic makes intuitive sense: actions which are successful are naturally more important to correctly classify than those that do not. Finally, we also use *class reweighting* to compensate for the asymmetry in the proportion of actions that do and do not succeed. Below we characterize the fully the Partially Observable Temporal Logic Planner (PO-TLP) loss function:

$$L_{POTLP} = w_M L_{WCE}(P_S, \hat{P}_S, w_C) + w_C P_S (R_S - \hat{R}_S)^2 + (1 - P_S)(R_F - \hat{R}_F)^2, \qquad (3.10)$$

where w_M represents the mis-classification cost, w_C is the class reweighing term, and L_{WCE} is the weighted cross-entropy loss:

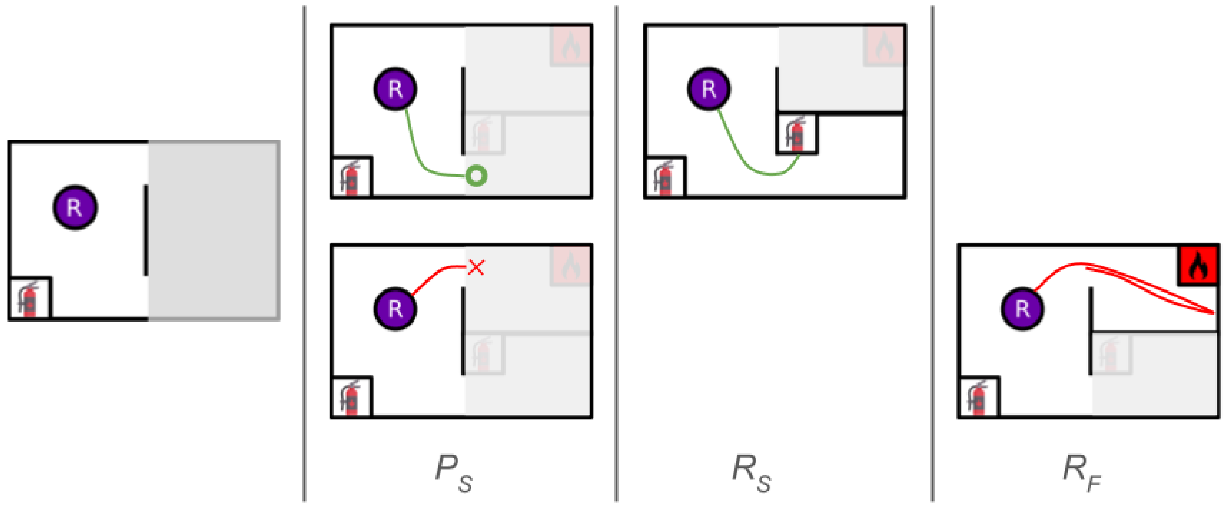


Figure 3.7: A depiction of how we compute training data from the underlying map. For the action of attempting to find a fire extinguisher while avoiding the fire, we identify the the frontier at the bottom will lead to a successful action, while passing through the one at the top will fail to do so. As such, we compute the cost of success by finding a path to the nearest extinguisher for the lower action, and the cost of failure for the higher upper one. Note that the cost of failure might require exploring the entire map, which we approximate by planning to the furthest region geometrically. We simultaneously compute the values for attempting to find the fire directly.

$$L_{WCE} = -w_C P_S \log(\hat{P}_S) - (1 - P_S)\log(1 - \hat{P}_S). \tag{3.11}$$

The labels for R_S and R_F are only relevant for successful or unsuccessful actions respectively. Therefore, in the above equations we only penalize them in those instances.

3.2.3 Collecting Training Data

To collect training data, we navigate the robot through environments using an autonomous, heuristic-driven agent in simulation, and teleoperation in the real world. We assume the agent has knowledge of propositions in its environments, so it can generate the feature vectors that encode actions for subgoals it encounters. As the robot travels, we periodically collect images and the true values of P_S (either 0 or 1), R_S, and R_F for each potential action from the underlying map. Figure 3.7 visualizes how we compute these values from the underlying map.

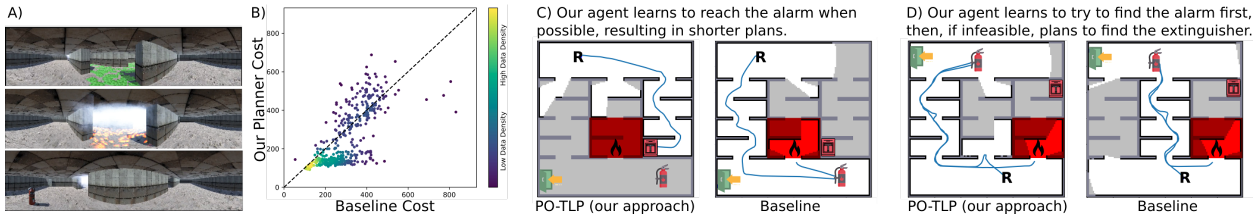


Figure 3.8: A comparison between our planner and the baseline for 500 simulated trials in the Firefighting environment with the specification $(\neg\texttt{fire}\ \mathcal{U}\ \texttt{alarm}) \vee ((\neg\texttt{fire}\ \mathcal{U}\ \texttt{extinguisher}) \wedge \Diamond\texttt{fire})$. The robot ("R") learns to associate green tiling with the alarm, and hallways emanating white smoke with fire (A), leading to a 15% improvement for total net cost for this task (B). Our agent learns it is often advantageous to search for the alarm (C), so in cases where the alarm is reachable, we generally outperform the baseline (highlighted by the cluster of points in the lower half of B). Our method is occasionally outperformed when the alarm cannot be reached (D), though we perform better in the aggregate and always satisfy the specification.

3.3 Experiments

To demonstrate the effectiveness of our approach, we perform experiments in both simulated and real-world environments. The planner from Ayala, Andersson, and Belta [111] solves a similar type of planning-under-uncertainty problem to those of interest in this paper, and likewise uses boundaries between free and unknown space to define the set of high-level actions available to the agent as it explores an occupancy grid world (albeit with a non-learned selection procedure and no visual input). Therefore, we compare the total distances traveled using each method.[2]

3.3.1 Firefighting Scenario Results

Our first environment is based on the firefighting robot example, simulated with the Unity game engine [134] and shown in Fig. 3.8. The robot is randomly positioned in one of two rooms, and the extinguisher and exit in the other. One of three hallways connecting the rooms is randomly chosen to be a dead end with an alarm at the end of it, and is visually highlighted by a green tiled floor. A hallway (possibly the same one) is chosen at random

[2]To be more comparable with our approach, our baseline modifies [111] by directly planning to regions in the LTS pertaining to transitions in the DFA and re-planning before reaching a subgoal at the same rate as our planner.

to contain a fire, which blocks passage and emanates white smoke. We collect data in this environment by running our baseline planner over a few dozen trials, periodically collecting labels for abstract actions defined by subgoals and predicates in the scene. We use this data to train our network to learn to associate the visual signals of green tiles and smoke to the hallways containing the alarm and the fire, respectively.

We test with four different task specifications—using the same network without retraining to demonstrate its reusability:

1. $(\neg\texttt{fire}\,\mathcal{U}\,\texttt{alarm}) \vee ((\neg\texttt{fire}\,\mathcal{U}\,\texttt{extinguisher}) \wedge \Diamond\texttt{fire})$: avoid the fire until the alarm is found, or avoid the fire until the extinguisher is found, then find the fire. To complete the task, the robot has the option of using either the fire extinguisher to fight the blaze, or simply pulling the alarm.
2. $\neg\texttt{fire}\ \mathcal{U}\ (\texttt{alarm} \wedge \Diamond\texttt{exit})$: avoid the fire until the alarm is found, then exit the building. Here we require the robot to find the alarm, removing the option of using the extinguisher.
3. $(\neg\texttt{fire}\,\mathcal{U}\,\texttt{extinguisher}) \wedge \Diamond(\texttt{fire} \wedge \Diamond\texttt{exit})$: avoid the fire until the extinguisher is found, then put out the fire and exit the building. In this case the robot must evacuate the building after extinguishing the fire.
4. $\neg\texttt{fire}\,\mathcal{U}\,\texttt{exit}$: avoid the fire and exit the building. Simple goal-directed navigation, where our robot (perhaps rationally) is instructed to leave the building.

Over ~3000 trials across different simulated environments and these four specifications, we demonstrate improvement for our planner over the baseline. Fig. 3.8 gives a more in-depth analysis of the results for specification 1. In Table 4.1 we compare the average cost and percent savings (with standard error) of the baseline and our proposed planner.

While our approach does not always outperform the baseline, it does in expectation for every specification, while still completing the task in every instance. Fig. 3.8B compares the cost incurred by both planners over ~500 trials for the first specification. For this task our method outperforms the baseline by 15% in total net cost. Fig. 3.8C and D highlight two instances of the environment, and the paths taken by our agents in satisfying the specification. Although not shown in this figure, we also compare our approach to the policy of

Table 3.1: Simulated Results in Firefighting Environment

Task	**Average Cost**			**Percent Savings**	
Specification	**Known Map**	**Baseline**	**Ours**	**Net Cost**	**Per Trial (S.E.)**
1	187.0	264.6	226.9	15%	14.4% (1.3)
2	392.1	696.0	461.2	34%	28.4% (1.0)
3	364.3	539.3	471.3	13%	6.1% (1.3)
4	171.2	269.1	203.3	25%	14.6% (1.0)

simply selecting the action with the highest estimated probability of success. In all cases, this greedy approach was outperformed—2.4 times the cost on average—by our planner as it fails to account for cost, leading to oscillations between actions in regions where the learning is uncertain.

3.3.2 Delivery Scenario Results in Simulation

We scale our approach to larger simulated environments using a corpus of MIT academic buildings containing *labs*, *classrooms*, and *offices*, all connected by hallways. Our dataset allows us to build occupancy grids, which we use to construct 3D simulations as shown in Fig. 3.9. In a case of simulation imitating life, simulated "professors" are located randomly in offices, "graduate" students in labs, and "undergraduates" in classrooms, which have differently colored walls in simulation. In order to provide our agent with a learnable signal, rooms that are occupied by delivery targets have their (simulated) lights turned on whereas other rooms are not illuminated. We also illuminate hallways.

Our agent must deliver packages to three randomly placed individuals in these environments: one each of a professor, graduate student, and undergraduate. We give the robot the specification $\Diamond\texttt{professor} \wedge \Diamond\texttt{grad} \wedge \Diamond\texttt{undergrad}$; instructing it to deliver a package to each individual in any order. Our agent is able to learn from the environments visual cues to navigate more efficiently. Over 80 simulations across 10 environments, the mean per-trial cost improvement of our agent compared with the baseline is 13.5% (with 6.1% standard error). Our net cost savings, summed over all trials, is 7.8%. Fig. 3.9 illustrates our results. We should note that, as with the *Firefight* scenario, we are careful to separate maps used for training from those used for testing.

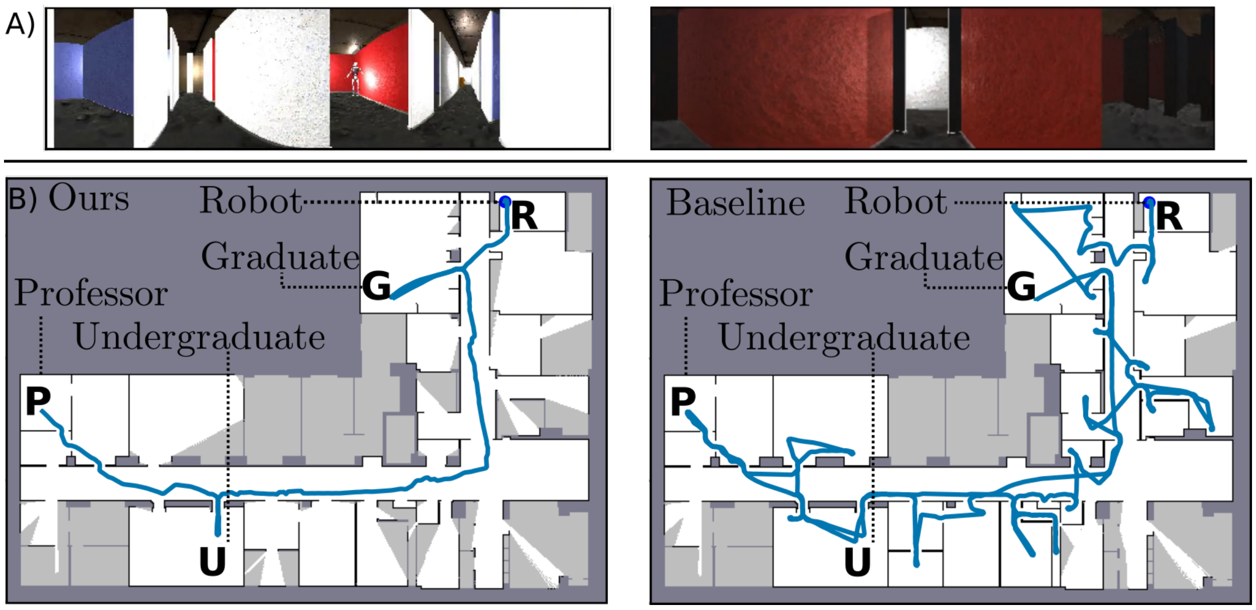


Figure 3.9: A) Visual scenes from our Delivery scenario in simulation. The rooms which can contain professors, graduate students, and undergraduates are colored differently and illuminated when occupied. B) A comparison between our approach (left) and the baseline (right) for one of several simulated trials for the task $\Diamond$`professor` $\wedge$ $\Diamond$`grad` $\wedge$ $\Diamond$`undergrad`.

3.3.3 Delivery Scenario Results in the Real World

Finally, we extend our Delivery scenario to the real world using a Toyota Human Support Robot (HSR) [135] with a head-mounted panoramic camera [136] in environments with multiple rooms connected by a hallway. The robot localizes and updates its map with a lidar and the *hector_slam* package [137], and streams images from a head-mounted panoramic camera [136]. The robot must deliver a package and a document to two people, either unordered ($\Diamond$`DeliverDocument` $\wedge$ $\Diamond$`DeliverPackage`) or in order ($\Diamond$(`DeliverDocument` $\wedge$ $\Diamond$`DeliverPackage`)). As in simulation, rooms are illuminated only if occupied. When the robot enters a room, the occupant specifies which object they need delivered, prompting a replan using an updated LTS of the environment.

These experiments were run in MIT academic buildings. Specifically, data were collected in Building 1, where different lights in rooms down a hallway were turned on or off depending on if a delivery target was in the room. Data were collected in this setting over a dozen trials. We tested our approach in a different building entirely, moving to MIT's building 36 (shown in Fig. 3.10). We ran 5 trials for each planner spanning both specifications and 3 different

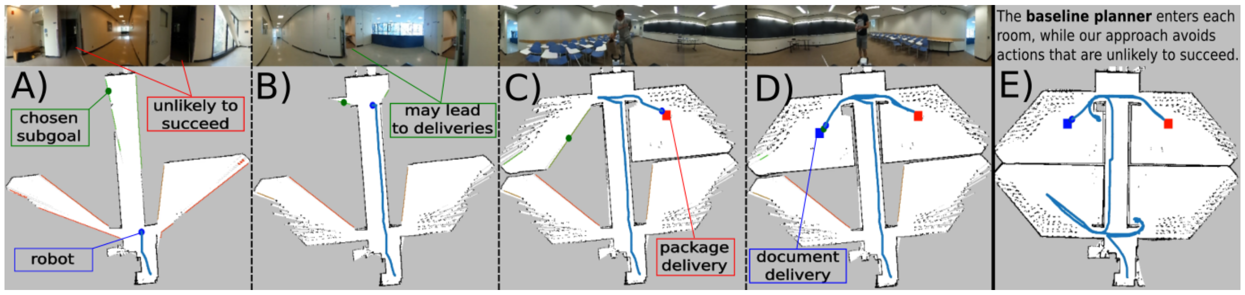


Figure 3.10: A comparison between our approach (A-D) and the baseline (E) for our real-world Delivery scenario. Our agent (blue dot) correctly predicts actions likely to fail (e.g., dark rooms in A) and succeed (e.g., completing a delivery in an illuminated room in B). Once the robot identifies delivery actions that lead to DFA transitions in known space, it executes them (C-D). Conversely, the baseline fully explores space before completing the task (E).

target positions in a test environment different from the one used to collect training data. We show improved performance over the baseline in all cases with a mean per-trial cost improvement of 36.6% (6.2% standard error), and net cost savings, summed over all trials, of 36.5%. As shown in Fig. 3.10, the baseline enters the nearest room regardless of external signal, while our approach prioritizes illuminated rooms, which are more likely to contain people.

The fact that we were able to learn a useful signal with such limited data in the real world highlights the benefits of our approach. We would expect that less data efficient methods would have struggled with the limited dataset, as well as the change in setting.

3.4 Discussion

In this chapter, we presented a novel approach to enable a robot to solve complex tasks in partially revealed environments. We model the underlying planning problem in this setting as a POMDP, which is particularly difficult to solve in large environments and complex task specifications. To enable efficient planning, the proposed method defines an abstraction over actions in physical and task space. Specifically, each abstract action designates what the robot will do in known space, where it will attempt to leave the known map, and what it intends to accomplish in unknown space. This abstraction allows us to represent tasks expressed to our agent in temporal logic as much shorter-horizon planning problems than they would otherwise be. However, the partially observable nature of the environment means that we cannot be certain any of these actions can be refined into a concrete plan.

Humans are an existence proof that predictions of feasibility and cost can be made in these settings from sensor input (in our case, vision), and that those predictions can lead to better decision making. Even when placed in an unfamiliar building, a human would know that, for example, hallways connect faraway regions of space, and classrooms are unlikely to contain coffee beans. We therefore define simple supervised learning problems to allow a robot to make predictions of the outcomes and costs of our abstract actions from panoramic visual input. We are careful to structure the inputs to our learned model such that it does not require re-training if the goal specification changes. Starting from the Bellman equation (Eq. (3.1)), we recognize that grouping future beliefs into two categories (success or failure of the current actions) allows us to derive a method to compute the optimal policy over actions for our robot to try given our network's predictions. However, even in our simplified representation, the resulting equation (Eq. (3.8)) is too expensive to tractably compute exactly.

To address this computational intractability, we adapt a stochastic search approach to our setting, which allows us to identify the lowest cost action in expectation for a given scene. We use our learned predictions to guide this search, therefore spending less time considering actions which are unlikely to succeed, or which are of prohibitive cost. Our planner is therefore able to reason about which abstract actions are likely to be refinable into a concrete plan at execution time, thus enabling it to account for the imperfections inherent in our planning abstraction. Once we have determined the best action according to our predictions, our agent computes a low-level trajectory in known space, and begins to travel down that path. As the agent travels, it receives new observations, allowing it to update its map, and potentially providing more information for its learned models. After a short period (on the order of a couple seconds), our agent uses this new information to re-plan from scratch, potentially choosing a new action. This planning loop continues until the robot completely satisfies the given goal specification (as highlighted in figure 3.1). Because our agent reveals the map as it travels, it is guaranteed to eventually satisfy any specification it is given, even if the learned predictions are misguided.

We demonstrated the effectiveness of our approach in three different settings. First, we considered a simulated environment of a building on fire. Our agent was given visual cues to identify where in the building it might find a fire extinguisher or an alarm, and learned to make predictions about the outcome and costs of different actions from visual input. The primary insight from these experiments was that our model was able to make useful predictions to guide search in this setting, even when we changed the goal specification. We then investigated how our approach scales to larger environments by testing our method in a simulated version of MIT academic buildings. We were still able to show improvement over the heuristic driven baseline in this setting for a given delivery task. Finally, we showed that our planner was useful in real world settings by implementing it on a Toyota HSR and testing it in building 36 at MIT. After training in a different section of campus, we showed that even in the real world with limited data, we are able to find efficient plans using our method.

Of course, despite the improvements we saw experimentally, there are some limitations to our approach, the impacts of which we outline here:

- **Inaccurate Abstraction:** One assumption we make in the training of our models is that we have access to ground truth labels for the feasibility and cost of different actions. Specifically, the labels for probability of success and cost of success are quite easy to attain given the ground truth map. Determining if an action might be successful is as simple as attempting to find a path to the appropriate labeled region in the map through the chosen frontier, and the associated cost can be determined from the length of that path. However, to get the cost of failure, we make an approximation. As outlined in section 3.2.3 and figure 3.7, R_F is computed by planning to the furthest region reachable in unknown space beyond the chosen frontier. However, there are many factors which may make this an incorrect estimate. One the one hand, there may be many branching paths through unknown space that make full exploration significantly more costly than planning to a single point. On the other hand however, in practice we often see an agent begin down a particular path, only to receive new information and reverse course upon re-planning. One way to more accurately estimate this value would be to track the actual experienced exploration cost using online learning methods, however it is unclear if this would lead to any observable improvement in planning.

- **Instantaneous predictions:** Another aspect of our planning approach which is somewhat limiting is the information our networks use to make predictions. As presented in section 3.2, our network is passed a vectorized representation of what the robot is trying to do in known space, as well as a camera image centered on the chosen subgoal. This image is a snapshot in time, and may not capture everything about the region the robot is considering exploring. Imagine for example the robot is traveling down a hallway, and sees an exit sign along its path. We would rightly expect it to predict that an action which attempts to find an exit in this direction has a high probability of success. However, as soon as the robot passes this sign, the next time it makes a prediction it will not have the same visual cue. As a result, the agent might lower its estimate of success. The worst case outcome here is that the predicted probability becomes so low that the agent changes its mind at the next planning cycle and reverses course.

  In general, non-local information may be relevant to decision making. Beyond the case described above, imagine our robot encounters a map of the environment it is in. We would like there to be some way for it to incorporate this information into its decision making. One potential way to include non-local information into our network predictions would be to include some method of either memory or message passing (or both) into our model architecture. For example, if we were to encode the environment as a Graphical Neural Network (GNN), and store images in the nodes, we might have a way to keep track of information across large distances [138]. The use of GNNs in scene encoding will be discussed further in the next chapter.

- **Planning cost:** A major contribution of this work is the ability to reduce long-horizon planning tasks into much more tractable problems. However, one limitation of the manner in which we compute costs in larger environments is the impact these calculations have on planning time. Specifically, in our simulated delivery experiments (section 3.3.2), we noticed that as the number of frontiers increase, the impact of computing the D value in equation 3.8 became limiting. Even with our stochastic planning approach, once the number of frontiers surpassed around 20, we were not able to expand enough of the search tree to outperform the baseline planner by much. It should be noted that we are even solving a simplified planning problem, as our agent is not considering geometrically how it will execute its actions. If we were to plan in the higher dimensional space needed to reason about grasps and arm trajectories, the planning problem would become even harder. In the next chapter we will consider how we can overcome the challenges of planning in known space for high-dimensional, complex tasks.

The novel research in this chapter demonstrates an ability to efficiently plan hierarchically in partially observed domains. In the scenarios considered here, the nature of the problem forces our agent to utilize a top-down planning approach. In general, this top-down strategy is susceptible to making decisions which lead our agent down dead-end paths, causing back-tracking. However, by informing our high-level planner of information initially not included in the higher levels of abstraction (panoramic RGB input), we are able to better guide decision making, leading to significant improvement over uninformed planning approaches.

We note that the limitations listed above are largely candidates for improvement. The method by which we collect data, architect our learned models, or compute trajectories in known space are all choices which may be made differently. In the next chapter we build upon the insights of this work to attempt to overcome the challenges of planning in known space for high-dimensional, complex tasks. Later on, in Chapter 5, we address the challenges of exploding environment complexity. Finally, we will consider opportunities for future work in Chapter 6, notably imagining how recent progress in foundation models might be incorporated into our approach.

Chapter 4

Learning Feasibility and Cost to Guide Task and Motion Planning

In the previous chapter, we demonstrated an ability to solve complex planning problems in partially-revealed environments, using a top-down hierarchical planning approach guided by visual input. However, we made a few notable assumptions when defining our planner that simplified the planning problem in some key ways. Critically, though we considered tasks where our robot had to "pick up" a fire-extinguisher or "deliver" a package to a human, the *sub-problem* of actually solving for each low-level trajectory executed by our agent was greatly simplified. We assumed we could grasp anything if we navigated to it, and deliver any package if the robot was in the vicinity of the target. As a result, we could abstract away much of the complexity of planning in high-dimensional domains, and solve for the cost of executing actions in known space with discretized grid search. Of course, this approach is not always feasible in reality, as the computational complexity of the combined task and motion planning problem is in general greater than that of sequential motion planning [19]. This is at least in part due to the fact that there are many contexts where the geometry of a given scene limits which actions are immediately feasible. Just as high-level actions which enter unknown space can fail, so too might those in known space. Grasps could be infeasible due to an obstruction, passageways may be too narrow to find a plan, or a platform might be too cluttered to find a valid place to put an object. Much in the same way that we could not easily refine the exploratory actions during planning, attempting to solve for low-level, high-dimensional trajectories naively in this context is also computationally challenging.

Despite considering multi-step planning specifications, another key assumption we made in the development of our Partially Observable Temporal Logic Planner (PO-TLP) was to treat the task purely as a sequential motion-planning problem (as defined in Chapter 2.1). As a result, our planner never had to consider how our robot's interaction with the world might affect anything beyond the agent itself (e.g., activating the alarm did not put out the fire, which could have affected how the agent then traveled to the exit). PO-TLP relied on the fact that we could train a network with (among other things) local sensory input, and make predictions of feasibility and cost for all future actions just as effectively as for the next action to take. For tasks beyond navigation however, any decision the robot makes can (and often does) have an impact on the environment itself. For example, if the robot moves one object to a new location, then attempts to grasp another object which was previously obstructed, the fact that the first object was moved clearly affects the feasibility of the second action. In this case, if we rely on an image of the scene to predict feasibility, or otherwise fail to account for the updated state of the world before acting, our agent would be unable to predict how moving the obstruction would impact its plan.

Consider, for example, a mobile robot attempting to "cook" multiple objects in a kitchen environment. Solving this problem hierarchically involves considering both the discrete sequence of actions (e.g., "pick up object 'A', navigate within the room, then place 'A' on a new platform"), and the constrained continuous and discrete parameters of those actions (a reachable grasp on A, a collision-free trajectory to move between configurations, a platform to place on). Problems of this nature can quickly become computationally challenging as the task complexity increases. The more objects the robot is assigned to 'cook', the longer the task horizon grows, the wider the breadth of options becomes, and the more sub-problems must be solved [19]. Moreover, as more objects are moved, the impact of constraints shared between sub-problems (e.g., one moved object impacting the placement of another) also become greater [19]. In this chapter, we build upon the intuition we found useful in partially revealed domains, and again aim to learn models to guide search to refine high-level, abstract plans, this time accounting for the increased complexity of Task and Motion Planning (TAMP).

One common approach for solving TAMP problems like the one described above involves defining external approaches for solving for specific sub-problems in a task (e.g., a sampler to find feasible grasps or collision-free motion planner for trajectories between robot configurations). As discussed in Chapter 2.1.2, a TAMP solver will then alternate between finding a plan at each level of the hierarchy. At the upper level, the planner returns discrete high level plans composed of abstract actions, then finds the continuous parameters of those plans by solving the relevant sub-problems [19], [36] at the lower level. In this approach, there is an implicit assumption that any task plan found at the upper level of the hierarchy might be feasible, and so it is worthwhile to try to solve its associated sub-problems. This is unfortunately not always true in reality. Imagine a robot attempting to pick up a mug at the back of a cluttered shelf. No matter how many grasps are sampled, there may be no trajectory which avoids all potential collisions with obstructing objects. Unsurprisingly, if a problem instance is particularly constrained (e.g., the object we are trying to grasp is buried behind many obstructions), high-level plans which do not consider moving those objects first may ultimately be infeasible. Often times TAMP solvers may attempt to refine infeasible high-level plans numerous times, failing to find a valid solution before a sub-problem is ultimately abandoned.

Clearly, a major challenge in this strategy is determining when to attempt to solve a particular sub-problem that is a part of one high-level plan, versus spending computation time solving a different sub-problem contained within a separate, potentially more feasible, plan. Even with clever approaches such as the adaptive algorithm in Garrett, Lozano-Pérez, and Kaelbling [36] or the Monte-Carlo Tree-Search (MCTS) based planner proposed in Ren, Chalvatzaki, and Peters [37], due to the combinatorial nature of task planning, coupled with the complexity of motion planning, problems can become computationally intractable very quickly as state and action spaces grow [19]. Particularly in problem settings where the chosen black-box samplers or solvers fail frequently (e.g., attempting to place an object on a cluttered platform), repeated failed queries to these solvers can cause planning times to balloon. This follows the intuition we built in the previous chapter. In instances where the principle of downward refinement is a bad assumption in general, we would expect hierarchical planning to be less efficient when done naively. Unfortunately, knowing ahead of time which sub-problems have feasible (or optimal) solutions is as hard as the original planning problem itself. There is no practical way to avoid occasionally attempting to solve sub-problems that do not actually have feasible solutions, but we would like to be able to identify ahead of time the expected cost of doing so, and subsequently guide search to minimize wasted computational effort.

If we as humans are able to make predictions about which objects we can pick up, and where we can feasibly place them, so too should our robots. There has been significant recent progress in accelerating planning for TAMP problems using learned models. Most relevant to this work are contributions which attempt to accelerate search from experience [46]–[53]. Many of these approaches learn to either ignore certain parts of a problem domain, or guide the search for high-level plans, but do not address how to learn within the planning process itself to accelerate finding continuous action parameters. Those that do [46] often require thresholds to prune search tree branches based on pre-trained feasibility predictions, which can perform will in aggregate across many planning problems, but lead to failures in certain cases with no way to recover. Moreover, given the fact that simply finding a satisfying solution is so difficult in these settings, often times considering metrics like execution time is an afterthought.

Our objective is to find plans which solve TAMP problems efficiently, both in terms of wall-clock time spent searching for the plan, as well as with respect to the time it takes to execute the plan on a robot. To that end, we propose an approach for learning to predict the outcome and cost of solving particular sub-problems used by existing TAMP algorithms, then use these models to accelerate planning. As in Chapter 3, the models we learn allow us to predict when the downward refinement property of our abstraction holds, and how expensive the refined plan is expected to be. In this chapter we propose two contributions. The first is a method to compute the expected cost of attempting to solve the relevant sub-problems within a high-level plan in a TAMP problem. Specifically, for each sub-problem (like grasp sampling or inverse kinematics) relevant to our robot, we train a model to predict outcomes (either success or failure) and costs (in both planning and execution time) for different inputs. This contribution is inspired by the work in Chapter 3. Second, we propose a novel stochastic planning approach to better take advantage of

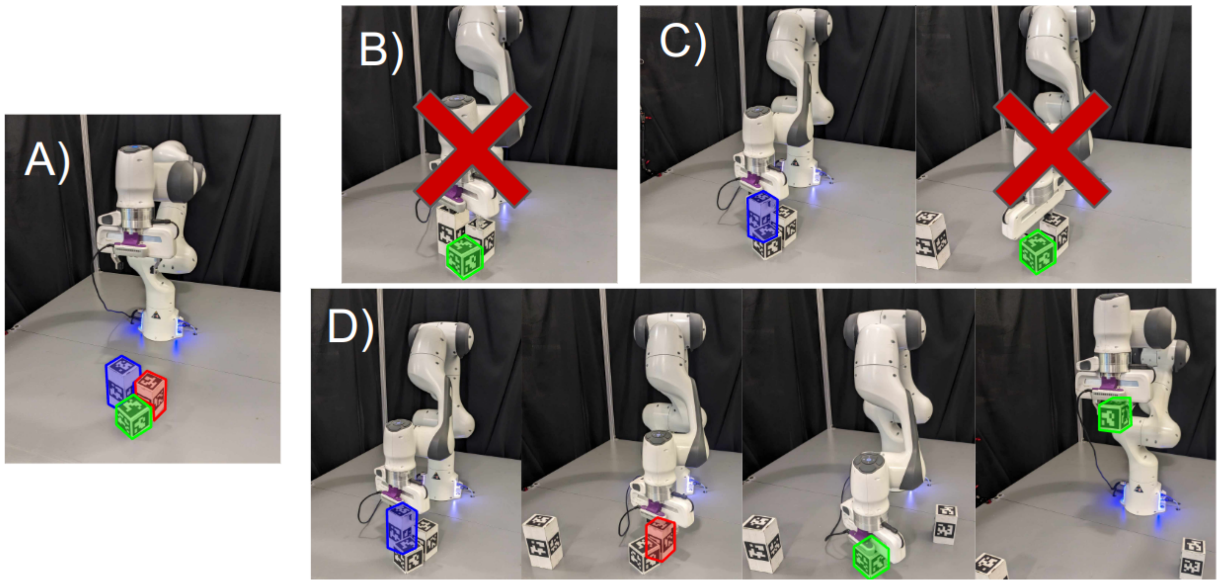


Figure 4.1: **A)** A Panda agent is tasked with grasping the block highlighted in green. **B-C)** A naive abstract plan might be to grasp the green block directly, ignoring some or all potential obstructions. When the sub-problems associated with these plans are attempted, computation will be wasted solving for motion plans that are infeasible due to collision. **D)** Instead, our approach utilizes learned models to guide planning by taking into account the feasibility and expected cost of sequences of sub-problems. As a result, our agent efficiently finds a plan which moves the relevant obstructions first.

our ability to estimate these outcomes and costs to accelerate planning. Our planner uses the learned model's estimates to guide search, then takes advantage of the UCT algorithm to account for potential inaccuracies in our models online. We propose a method by which our models can be evaluated to return the expected cost of a grounding and executing a full high-level plan, without having to immediately solve each sub-problem, enabling our planner to reason about potentially re-solving upstream sub-problems during search.

This chapter is organized as follows. We first outline how we build a planning abstraction for TAMP problems using PDDLStream, highlighting how approaches which do not account for the infeasibility of abstract plans struggle in many classical TAMP problems. Second, we derive how we estimate and refine online the expected cost of grounding and executing an abstract action plan using these predictions. Then, we describe the construction and training of simple models offline to predict the outcome and costs of individual sub-problems in a TAMP domain. Next, we demonstrate how this cost estimate is incorporated into a novel planner to more efficiently guide search. Finally, we implement our approach using the PDDLStream framework defined in [36], and demonstrate an improvement in planning time on two different, simulated platforms over a heuristic driven baseline, as well as on a real robot [37].

4.1 An Abstraction for Task and Motion Planning

As discussed in Chapter 2, the combined TAMP problem jointly considers elements of high-level task planning [31], [32] and low-level motion planning [33] in an attempt to solve hybrid discrete-continuous, multi-modal planning problems [19]. Solutions for TAMP problems take the form of a sequence of parameterized actions $\pi = [a_1, a_2, ..., a_n]$ that define a plan, where parameters satisfy each action's constraints [19]. One approach for representing a TAMP problem—which we use in this work—is an extension of the Planning Domain Definition Language (PDDL) called PDDLStream [36].

A PDDLStream problem (P, A, S, O, I, G) is specified as sets of predicates P, actions A, streams S, objects (referred to elsewhere in this thesis as symbols) O, an initial state I, and a goal state set G. The initial and goal states of a PDDLStream problem are sets of facts: instances of boolean functions called predicates $p(x) \in P$, which are parameterized by tuples of objects $x \in O$. For example, the fact that certifies if the robot is at a given pose is an instantiation of the predicate $AtPose\ ?p$, and is either true of false for different pose objects $?p$. Actions $a \in A$ are defined by two sets of predicates: preconditions and effects, and are parameterized by object tuples x. For a given input x, if the preconditions evaluate to true, the action is legal, and the effects specify which facts will change value in the subsequent state. Below we see a PDDL encoding for a *pick* action:

```
:action pick
    :parameters (?o - obj ?p - pose ?g - grasp ?q - conf ?t - traj)
    :precondition (and (iskin ?o ?p ?g ?q ?t)
                       (atpose ?o ?p)
                       (handempty)
                       (atconf ?q)
                       (canpick)
```

```
                        (not (usedGrasp ?o ?p ?g))
                        (GraspAtPose ?g ?p)
                        )
    :effect (and (atgrasp ?o ?g)
                 (canmove)
                 (not (atpose ?o ?p))
                 (not (handempty))
                 (increase (total-cost) 100))
```

For certain actions, preconditions may include facts and objects that are either cumbersome or impossible to add to initial state I. For example, one of the preconditions for the *pick* action defined above is a collision-free configuration `?q` which grasps the chosen object. The fact (`IsKin ?o ?p ?g ?q ?t`) indicates that `?q` is kinematically feasible and collision free. If a solution to the task exists, in order to guarantee it can be found by a PDDL solver, we might attempt to enumerate all possible feasible configuration objects for all possible grasps (or at least a discrete subset of these which reasonably covers the configuration space). However, for a 20-DOF robot (e.g., a PR2), doing so without creating a potentially intractably large problem is impossible. To account for this, PDDLStream problems include object generators called *streams* $s \in S$, which allow a planner to represent potentially relevant sub-problems, such as identifying relevant robot configurations, without tabulating all solutions. Streams consist of sets of (1) input and (2) output objects, (3) domain predicates which must be true in the input, (4) action predicates to be certified if the queried stream is successful, and (5) an external function that attempts to solve a specific sub-problem when the stream is queried. The stream that certifies the precondition (`IsKin ?o ?p ?g ?q ?t`) is given below:

```
:stream inverse-kinematics
    :inputs (?o - obj ?p - pose ?g - grasp)
    :domain (and (IsPose ?o ?p) (IsGrasp ?o ?g))
    :fluents (AtPose)
    :outputs (?q - config ?t - trajectory)
    :certified (and (IsConf ?q) (IsTraj ?t) (IsKin ?o ?p ?g ?q ?t))
```

When an action has a precondition which can only be certified by a stream, that stream can be queried in an attempt to solve the associated sub-problem, and determine if said precondition can be certified. The *inverse-kinematics* stream above takes in an object, its current pose, and a valid grasp (relative end-effector position) of that object. The `:fluents` field specifies addition inputs, in this case, all AtPose facts in the current state [1]. When queried, the stream calls an IK solver (written in some other language like Python or C++) which returns two new objects: a collision free configuration `?q` and a short trajectory `?t` which represents the closing of the gripper on the object of interest. If the sub-problem is successfully solved, the stream certifies that `?q` is a robot configuration, `?t` is a trajectory, and that `?q` and `?t` are collision free when object o is at position `?p` for grasp `?q`. In that case, the new objects are added to the domain, (`IsKin ?o ?p ?g ?q ?t`) is added to the state I, and the precondition to the `pick` action is satisfied. Conversely, if the sub-problem is not solved, the fact is not certified, the action's preconditions are not met, and it cannot be included in a valid plan.

It is precisely this automatic generation of the objects—only when they are useful to solving the sub-problems in a potentially complete task plan—that is the power of interleaved TAMP solvers like PDDLStream. Note, however, that though a stream might produce a valid solution for its inputs, that output may still not be useful to the overall problem. The IK stream for example requires a grasp as an input, but many sampled grasps, and corresponding objects, may never lead to a collision free configuration depending on the object's pose, or the positions of other objects in the scene as specified by the `fluents`. A stream may therefore need to be queried many times with potentially many different inputs before generating an solution which jointly satisfies all constraints in a TAMP problem. We refer the reader to [36] for a more detailed description.

PDDLStream encodings define a hierarchical planning problem, and there are several solvers that have been developed for TAMP problems specified in this way. The simplest approach requires defining the *level* of a fact, which corresponds to the number of chained sub-problems that would need to be solved in order to certify said fact. For example, the level of the `IsKin` fact would be 2, considering a grasp needs to be sampled (defined by a different

[1] *Fluents* are an artifact of the PDDLStream software, which allows us to pass an unspecified number of arguments to a stream.

stream) in addition to the IK stream. Starting with level-0, we can first attempt to certify all facts at the given level, then attempt solve the PDDL problem from the resulting state. If that attempt fails, we increase the level, and try again. This is the approach defined in the *Incremental* algorithm [36], [38], and has two negative side effects: 1) spending time solving sub-problems which are ultimately irrelevant to any plan, and 2) therefore producing many irrelevant facts, which makes the discrete search for plans more computationally expensive.

More sophisticated approaches improve on the Incremental approach by delaying the step of solving sub-problems by optimistically assuming that any time a stream is needed to certify an action predicate, it can be queried successfully. In practice, these approaches add what can be referred to as *optimistic-objects* to the planning domain according to the current *level*, which correspond to the outputs of streams if their associate sub-problems were to be successfully solved. Then, PDDL solvers generate abstract plans π, which are composed of actions that may be parameterized by these optimistic-objects. For a given abstract action plan π, we can generate the sequence of sub-problems s that must be solved to ground the optimistic parameters in π: referred to as the stream-plan ψ. If each $s \in \psi$ is able to be solved for satisfying output objects, then the concrete action plan solves the original TAMP problem [36], [37]. Unsurprisingly, this optimistic assumption struggles in certain settings; in particular when many of the sub-problems in a scene are infeasible. In such scenarios, optimistically assuming any abstract plan can be refined to a concrete one leads to solvers spending time unnecessarily attempting inherently infeasible sub-problems. In the following section, we will highlight a scenario where this is the case, and consider how we can estimate cost to guide search in TAMP.

4.2 Reformulating the Task and Motion Planning Problem

A principal challenge in solving TAMP problems is determining which of a set of abstract action plans (found at higher levels of the hierarchy) we should attempt to ground to concrete parameters. If we spend computation on the "wrong" action plan, we waste time solving sub-problems that are infeasible, or otherwise irrelevant to a complete plan. Therefore, our planner must decide in some way which plan it should attempt to ground, and so we need a method to compute and compare the expected costs of different high-level plans.

For the sake of grounding this discussion, let us imagine a simple task, wherein a table top manipulator is trying to move one of three differently sized blocks from one platform to another (similar to the setting highlighted in figure 4.1, but more specifically that in figure 4.2). We will assume our agent has certain skills, for example, the ability to pick up objects, place them down, and move between robot configurations. The sub-problems the robot must solve to execute those actions and accomplish its task are to sample grasps and placement poses, solve Inverse Kinematics problems, and to compute motion plans between configurations using an RRT solver [23]. The full domain and stream file for this scenario can be found in Appendix A.1. We task our robot with grasping the smallest block, constrained to only use top-down grasps. In Figure 4.2-A, we can see that the red block obstructs any attempts to pick up the smaller target (green) block due to its position (the blue block is irrelevant), and so may need to be moved out of the way. Next, we compare two possible abstract plans our agent might consider (visualized in Fig. 4.2 and expanded below). The first (see π_1 below) would be to pick up the green object directly, then place it on the goal platform. Here we see this abstract action plan, and corresponding sub-problem sequence (bolded values indicate parameters that must be solved for):

Action Plan π_1

1. Move(conf1, **conf2**, **traj1**)
2. Pick(obj2, **grasp1**, **conf2**, **pose1**, . . .)
3. Move(**conf2**, **conf3**, **traj2**)
4. Place(obj2, **grasp1**, **conf3**, **pose2**, . . .)

Sub-problem Sequence ψ_1

1. **grasp1** = Grasp(obj2)
2. **conf2** = IK(obj2, **grasp1**, pose1)
3. **traj1** = RRT(conf1, **conf2**)
4. **pose2** = Place(obj2, surface2)
5. **conf3** = IK(obj2, **grasp1**, **pose2**)
6. **traj2** = RRT(**conf2**, **conf3**)

As we can see, we have to solve 6 individual sub-problems to fully parameterize this action plan. Another option would be to first move the red block out of the way, then move the green block (π_2):

Action Plan π_2

1. Move(conf1, **conf2**, **traj1**)
2. Pick(obj2, **grasp1**, **conf2**, **pose1**, …)
3. Move(**conf2**, **conf3**, **traj2**)
4. Place(obj2, **grasp1**, **conf3**, **pose2**, …)
5. Move(**conf3**, **conf4**, **traj3**)
6. Pick(obj1, **grasp2**, **conf4**, pose3, …)
7. Move(**conf4**, **conf5**, **traj4**)
8. Place(obj1, **grasp2**, **conf5**, **pose4**, …)

Sub-problem Sequence ψ_2

1. **grasp1** = Grasp(obj2)
2. **conf2** = IK(obj2, **grasp1**, pose1)
3. **traj1** = RRT(conf1, **conf2**)
4. **pose2** = Place(obj2, surface2)
5. **conf3** = IK(obj2, **grasp1**, **pose2**)
6. **traj2** = RRT(**conf2**, **conf3**)
7. **grasp2** = Grasp(obj1)
8. **conf4** = IK(obj1, **grasp2**, pose3)
9. **traj3** = RRT(**conf3**, **conf4**)
10. **pose4** = Place(obj1, surface2)
11. **conf5** = IK(obj1, **grasp2**, **pose4**)
12. **traj4** = RRT(**conf4**, **conf5**)

For this second action plan, the associated sub-problem sequence is much longer. As a result, most TAMP solvers—which rely on heuristics based in part on plan length—will be heavily incentivized to spend time searching for parameters for the first plan, which moves the green block directly. However, due to the geometry of the scene, we can see the plan which moves the red block first is far more likely to lead to a concrete plan despite its length. We visualize the two scenarios in figure 4.2. How can we better represent the true cost of grounding an abstract plan in order to better guide search?

The total time it takes for the robot to solve a given task can be split into time spent *thinking* and time spent *acting*. Our objective is therefore to find plans which solve TAMP problems efficiently, both in terms of wall-clock time spent searching for the plan, as well as with respect to the time it takes to execute the plan on a robot. In Chapter 3, we considered a problem where the feasibility and cost of executing a plan was uncertain, but planning itself was inexpensive (on the order of a few seconds or less). Conversely, in TAMP, we often consider the execution of a plan to be predictable, while the act of finding a good plan itself

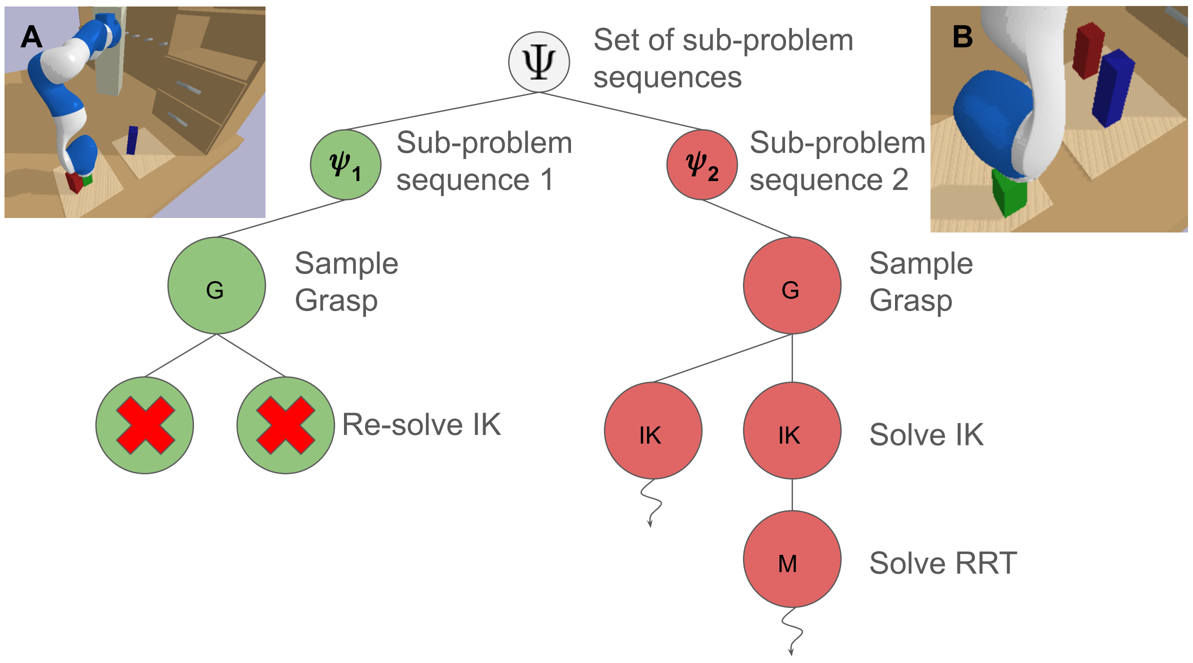


Figure 4.2: **A)** A table-top manipulator is tasked with holding the shorter green block, but the red block obstructs its grasp. **B)** By moving the red block out of the way, we can pick up the green one. **Center)** We visualise a subset of each stream plan. Notably, grasping the green block directly as in ψ_1 is a shorter plan, however attempting to find a collision free configuration for any sampled grasp is infeasible. Heuristics in TAMP approaches will nevertheless encourage search to attempt to ground this plan as it is shorter than ψ_2. We would like use geometric information to guide search to be more efficient.

might take several minutes. The source of this extra cost lies in attempting to solve the relevant sub-problems. For example, each attempt to solve an inverse kinematics problem might take around half a second, and the timeout for a motion planning problem (using the RRT) might be set to 2 or 3 seconds. Thus, if our search attempts to solve for many different motion plans, any of which may fail, search time will be a significant contributor to the sum of planning and execution costs. We can therefore model attempting to solve a sub-problem in the same way we did executing actions in unseen space in Chapter 3; whereas before we refined an abstract plan by taking the action, now we do so by solving the sub-problem sequence. In both cases, there is uncertainty regarding feasibility: *"does this hallway lead to the fire-extinguisher"* $\rightarrow$ *"does the IK problem have a valid solution"*, and cost: *"how much time will it take to reach the extinguisher"* $\rightarrow$ *"how long will it take to find the trajectory between these configurations."*

With that in mind, we once again recognize that we can split the outcomes of actions into cases where we either succeed or fail to solve a given sub-problem. Taking inspiration from the work in Chapter 3, the cost of attempting to solve a single sub-problem can be written as follows:

$$Q_p(\psi_t) = P_{S_t} R_{S_t} + (1 - P_{S_t}) R_{F_t}, \tag{4.1}$$

where $Q_p(\psi_t)$ is now the cost of solving sub-problem t in sub-problem sequence ψ. As before, P_{S_t} is the probability an action succeeds, and in this case that means solving a sub-problem. However, now R_{S_t} refers to the cost of successfully solving the sub-problem in wall-clock time, and R_{F_t} represents the cost of attempting to find a solution and failing. Note that we no longer include the D term, as the entirety of the action occurs in "known" space.

4.3 Computing the Cost of Solving and Executing an Action Plan

The above equation defines the expected cost of attempting to solve a single sub-problem in a sequence. In order to evaluate the expected cost of a sequence of sub-problems ψ, we must consider the subsequent costs both in the case where we solve the sub-problem (we attempt to solve the next one in ψ), and when the attempt fails. In Chapter 3.1.3, we derived from the Bellman equation a method to estimate the cost of a sequence of stochastic actions with binary outcomes, and re-state that equation here:

$$\begin{aligned} Q(b_h, a_{sz'z''}) = D(b_h, a_{sz'z''}) + \underline{P_S(b_h, a_{sz'z''})} \times \left[\underline{R_S(b_h, a_{sz'z''})} + \min_{a_S \in \mathcal{A}(b_{h_S})} Q(b_{h_S}, a_S)\right] \\ + \left[1 - \underline{P_S(b_h, a_{sz'z''})}\right] \times \left[\underline{R_F(b_h, a_{sz'z''})} + \min_{a_F \in \mathcal{A}(b_{h_F})} Q(b_{h_F}, a_F)\right], \end{aligned} \tag{4.2}$$

There are three notable differences between the equation above from Chapter 3, and the cost we are trying to compute for finding concrete values of abstract action plans. First, this is not a Partially Observable Markov Decision Process (POMDP) in that we know exactly our subsequent state if an action succeeds or fails. Thus, we do not reference the notion of a *belief state*. Second, we note that the sequence of actions (in this case solving sub-problems) has already been determined by the high-level planner, meaning we always know which sub-problem we will attempt to solve next if we succeed in solving the current one. These two differences remove the need to take a minimum over the costs of future actions: $\min_{a_S \in \mathcal{A}(b_{h_S})} Q(b_{h_S}, a_S \equiv Q_p(\psi_{t:T})$. Finally, the third difference is that if we fail to solve a sub-problem, the full sequence might still be feasible if a different solution was found to a sub-problem up-stream. For example, if we fail to find a motion plan between two robot configurations, we can re-sample a grasp on the chosen object, which might yield a different configuration for which a valid trajectory does exist. Therefore, the cost of failing to solve a particular sub-problem must involve re-solving the sub-problems from step 0 up to that point in the plan: $\min_{a_F \in \mathcal{A}(b_{h_F})} Q(b_{h_F}, a_F \equiv Q_p(\psi_{0:T})$ We re-write Equation 3.8 here with our new notation, recursively representing the expected planning cost Q_p of stream plan ψ, beginning at step t until the final step T, as:

$$Q_p(\psi_{t:T}) = P_{S_t}(R_{S_t} + Q_p(\psi_{t+1:T})) + (1 - P_{S_t})(R_{F_t} + Q_p(\psi_{0:T})), \tag{4.3}$$

where $Q_p(\psi_{t:T})$ represents the total cost (in seconds) it would take to solve each sub-problem $s_t \in \psi$ (but not yet execute any generated trajectories), beginning at step t in the plan. Because we are no longer taking the minimum of a set of actions, we should be able to re-formulate this equation without requiring recursive values. Notice that $Q_p(\psi_{0:T}) \equiv Q_p(\psi_{0:t}) + Q_p(\psi_{t:T})$, meaning that the term $Q_p(\psi_{t:T})$ exists on both sides of our equation. We can manipulate Eq. 4.3 algebraically, and re-write our expected cost:

$$Q_p(\psi_{t:T}) = (R_{S_t} + Q_p(\psi_{t+1:T})) + \frac{1 - P_{S_t}}{P_{S_t}}(R_{F_t} + Q_p(\psi_{0:t})). \tag{4.4}$$

Due to the stochastic nature of some external solvers, certain sub-problem can be queried infinitely many times, and may eventually yield a successful output, particularly given different inputs. Eq. 4.4 represents this intuition, as the expected planning cost for a particular step in the recursion is simply the cost of solving a sub-problem, plus the number of times we expect to fail to solve s_t, times the expected cost of each failure. The ratio $\frac{1-P_{S_t}}{P_{S_t}}$ represents one less than the expected number of attempts before success in the geometric distribution parameterized by P_{S_t} (assuming independent samples). Finally, we can unroll the recursive steps in our computation, and write the expected cost of successfully solving each sub-problem in ψ as follows:

$$Q_p(\psi_{t:T}) = \sum_{\tau \in t:T} \left(R_{S_\tau} + \frac{1 - P_{S_\tau}}{P_{S_\tau}} (R_{F_\tau} + Q_p(\psi_{0:\tau})) \right). \tag{4.5}$$

Notably, we have not yet considered the cost of executing the plan. In our formulation, this cost is only relevant for complete plans, as we do not begin execution until we find a full concrete plan. We introduce an additional term R_M, to represent the cost of executing any motion plans generated by solving a given sub-problem. The total cost of execution is therefore $Q_e(\psi_{t:T}) = \sum_{\tau \in t:T} R_{M_\tau}$. Finally, we add the planning cost Q_p to the cost of executing all generated trajectories in a plan, Q_e, to recover the total cost of planning and execution Q:

$$Q(\psi_{t:T}) = Q_p(\psi_{t:T}) + Q_e(\psi_{t:T}). \tag{4.6}$$

Using Eq. (4.6), we are able to estimate the expected future cost of planning and execution from any point in a given ψ. In the following section, we discuss how we learn the terms needed to compute Eq. (4.6).

4.4 Learning to Model External Functions

A major challenge for efficient TAMP strategies is the very thing that makes planning possible at all: the separation of the abstract discrete plan and the low-level geometric plan. The high-level planner is intentionally spared the challenge of reasoning about low-level geometry, which enables it to produce candidate abstract action plans quickly. However, a side effect of this is that these plans may be geometrically infeasible, often in a way that is obvious to a human observer. Consider again the case of moving blocks from one platform to another. Given enough experience, we should be able to reason that any plan which grasps the shorter green block is unlikely to succeed, or will at least be very costly to find. This

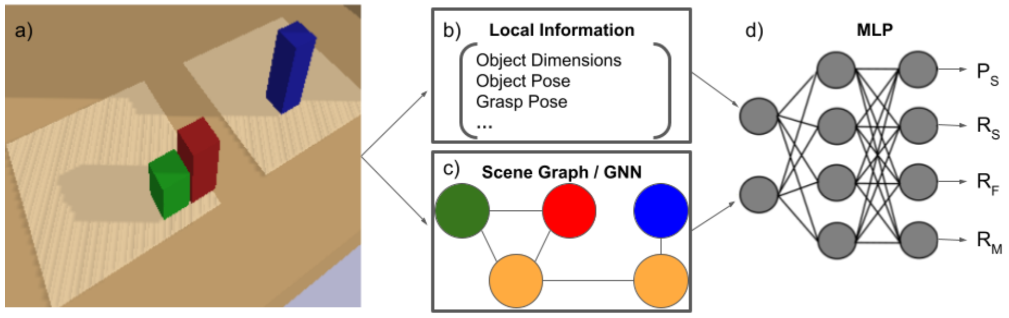


Figure 4.3: An example of how we decompose a scene **(a)** into inputs for our various learned models. First we vectorize the local information that is relevant for a particular sub-problem **(b)**. Then, we embed the global information into a Graphical Neural Network (GNN) [138] **(c)**. After message passing, we concatenate the global feature vector of the graph with the local information and pass that through an Multi-Layered Perceptron (MLP) to generate the properties used to compute cost **(d)**.

intuition is what leads us to our approach. If we can inform the higher levels of abstraction about low-level information that is relevant for planning, we can improve upon naive decision making. By learning to predict the feasibility and costs of solving different sub-problems, we can compute Eq. (4.6) and guide search for a concrete action plan.

We train offline separate models for each sub-problem associated with a stream $s \in S$ in a domain, mapping an individual stream's inputs to estimates of the outcome and costs of querying for and executing the associated sub-problem. Each model has two types of input: local information that is defined in the domain of the stream (e.g., the start and goal configurations passed to a motion planner), and global information (e.g., the poses and dimensions of other objects in the scene). Global information, derived from a scene graph, is embedded in a Graphical Neural Network. The GNN is composed of a node model, an edge model, and a global model, each of which shares the same architecture: two fully connected layers of size 128, with leaky ReLU activation after the first layer. For each object in our scene graph, we pass the pose and dimension features through two fully connected layers, outputting a node feature vector that is N x 128. We similarly embed the edge features (the translation between two objects), drawing edges according to which objects are supported by the same surface, then perform one message passing step between the node and edge models. Next, we pass each node through the global model, and concatenate the softmax of the output with the vectorized local stream information. We pass this vector through a 4 layer MLP to produce the model's output.

Given local and global information, each network produces four outputs: P_S; the probability of solving the sub-problem for the given inputs, R_S; the expected cost of solving the sub-problem successfully in wallclock-time, R_F; the expected cost of failing to solve the sub-problem, and R_M; the expected time it would take to execute any motion plan output (if a sub-problem does not generate a trajectory, we set this value to 0). With these four properties, we have the ability to compute the expected cost of attempting to solve, then, if successful, execute any trajectory generated by a given sub-problem. We compute the cross-entropy loss to train the probability output, and the mean squared error for regressing the cost predictions. All models were trained using the Adam optimizer (with default parameters) over 10 epochs with a batch size of 32. The labels for our four outputs can be generated by tracking the outcome and costs each time a stream is queried during search. In this way, we can generate training data by running any PDDLStream based planner. Figure 4.3 depicts a representation of the network modeling an inverse kinematics solver.

There is one additional subtlety to the training process that we must note. So far, we have considered the problem of attempting to predict the outcome and costs of actions given particular inputs (e.g., predicting our four values for an inverse kinematics problem given an input object, pose, and grasp). However, as we have mentioned, for longer sub-problem sequences, the inputs to a given sub-problem may not themselves yet have been solved for, in which case we do not have a full set of inputs. We would still like our models to be able to make predictions in this instance, and be able to do so with incomplete information. To do this, we pass along with all input a value which represents a flag: *zero* if the input value is unknown, and *one* if it is known. Additionally, in the case where the value is unknown, we pass a vector of zeros, appropriately sized, instead of the true value. For example, if we do not yet know the Pose of an object we intend to grasp, we will pass our IK model a vector of seven zeros along with a single zero as a flag. In this case our model recognizes the absence of a true pose, and returns the expected values of our outputs for the chosen object and grasp for an unknown pose.

In order for our models to be able to make reasonable predictions in these cases, we must present them with relevant data at training time. Therefore, during data collection we

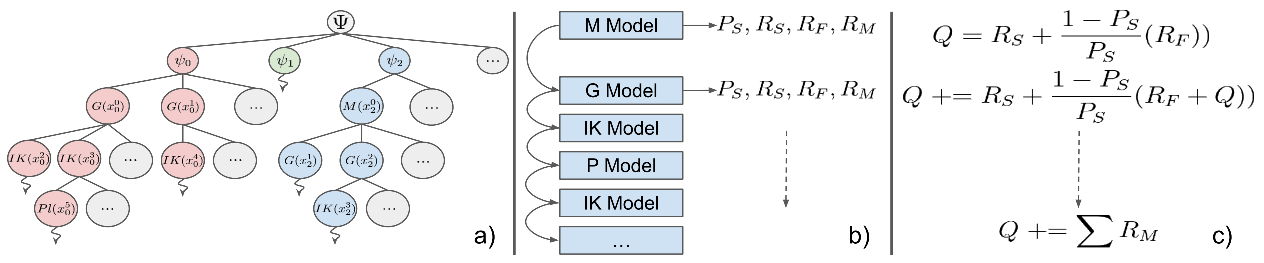


Figure 4.4: **A)** The root node in our tree represents a set of sub-problem sequences Ψ, and each child is a sample ψ from that set. For all subsequent nodes, the available actions consist of attempting to solve a sub-problem, in this case: G: sample grasp, Pl: sample stable placement, IK: solve for kinematically feasible configuration, and M: solve for collision free motion plan. **B)** At each node, we use learned models to predict the feasibility and costs of attempting to solve all remaining sub-problems in the sub-tree. **C)** We use these predictions to estimate the expected cost of finding satisfying solutions for the remaining steps in ψ from each node using eq. (4.6), defined in Sec. 4.3.

record how often a sub-problem would have full or partial inputs during search. Then, during training, we intermittently "block out" particular input values according to the observed frequency. By carefully weighting our training data in this way, we ensure our models see partial input prediction problems with the same regularity that they would expect at test time and are still able to make useful predictions when faced with such scenarios.

4.5 Planning with our Models

We now have a collection of learned models as well as an equation to estimate the cost of solving a full sub-problem sequence using the predictions made by those models. In Chapter 3, we formulated search for the optimal action plan as a stochastic search problem using Partially Observable UCT (PO-UCT), and we may be tempted to re-use that approach here. However, there are some subtle differences which allow us to do better than planning purely with our learned models. Notably, in order to refine an abstract action in Chapter 3, our robot would have to travel potentially large distances into unknown space. As a result, we would fully plan using our predictions of feasibility as the dynamics model in task space, then take an action and re-plan. However, in the TAMP formulation, we can attempt to solve a sub-problem comparatively quickly, and so have the ability to do so as a part of the search for a concrete plan. In *eTAMP* [37], the authors propose using Progressive Widening for Upper Confidence Bounded Trees (PW-UCT) [139], [140] to search for parameters that satisfy the constraints of an action plan. We build upon this approach for our planning algorithm, using our models to guide search, and so outline it briefly here.

4.5.1 Searching for Abstract Stream Plans

The first step in our approach is to generate several abstract action plans $\pi \in \Pi$ — sequences of actions which would represent a successful plan if all preconditions are met — from a PDDLStream problem using a top-k planner [141]. As mentioned in Section 4.1, some or all of an action's preconditions may be certifiable only by a stream. However, during the search for abstract action plans, we do not explicitly attempt to solve the sub-problems associated with those streams, as this would be prohibitively expensive. Instead we solve for abstract plans π, and compute the associated stream-plan ψ for each. Given a set of k stream plans $\psi \in \Psi$, we can begin to attempt to search for these parameters.

One approach for this search would be to simply query Equation (4.6) for each stream plan given the initial state of the problem, and guide search using these estimates only. However, during search in non-trivial domains, we may need to consider different potential solutions to the same sub-problem (e.g., sample multiple different grasps on a block). Because the output of a sub-problem may depend on its input, there can and will be different cost and feasibility estimates for the same step in a stream-plan depending on the parameters that are passed to the model (which depend on the solutions to upstream sub-problems). Therefore, when a new solution to a sub-problem is found, we can update our predictions for the remainder of a stream plan for a more accurate estimate. As we progress in our search for the parameters of a plan, some predicted values will vary, and so too will the remaining estimated expected cost. The differences in estimates can help us guide search, and we account for the associated uncertainty with PW-UCT.

4.5.2 Searching within Stream Plans

There are four distinct steps in a PW-UCT search problem (as there were for our PO-UCT in Chapter 3). The first is *selection*, where the existing tree is navigated according to the

UCT heuristic (Eq. (4.7)) to find a node to expand. Next, in the *expansion* phase, a child from the selected node is generated and appended to the tree. In the *simulation* step, we continually add nodes from the newly expanded child until either an expansion fails, or we successfully reach our goal. Finally, we update the statistics (total node visits and accrued reward) of all visited nodes via *back-propagation.* This process is repeated until a solution is found.

Our tree is built as follows. At the root node (level 0), the available actions correspond to selecting one of the stream plans returned by the top-k planner. After this choice, each level-1 sub-tree is associated with different evaluations for that stream plan. Each level in a sub-tree corresponds to a specific stream, and each node in a level to a solution of the associated sub-problem. Because sub-problems can be re-solved and potentially produce novel results, there are infinitely many actions from each node (although a tree will only get as deep as the length of a stream plan). Refer to Figure 4.4 for a depiction of a growing search tree.

4.5.3 Guiding Search with Learned Cost

During traversal from the root to a leaf in the selection phase, the UCT equation (4.7) is used to choose the next node:

$$\underset{v_i}{argmax}\ Q(v_i) + c\sqrt{\frac{2ln(N(v))}{N(v_i)+1}}, \tag{4.7}$$

where v_i represents a child of node v, $N(v)$ denotes the number of times a node has been visited, and $Q(v_i)$ gives the online estimated value of a particular child.

The UCT equation (4.7) relies on an estimate of Q in order to guide search. In *eTAMP*, the authors propose a heuristic based on the depth of the search tree, and accrued reward [37]. Such heuristics, while potentially useful, require domain knowledge, may necessitate tuning, and cannot adapt to different streams within a plan. Implicitely, they assume all unsolved sub-problems are feasible, which we have demonstrated is an unreasonable assumption in most robotics problems.

Instead, we use our learned models, applied to each step of the remaining stream plan, to more efficiently estimate Q and guide search. We evaluate Eq. (4.6) for each visited node in order to get an estimate for remaining expected cost. Specifically, we use the negative of the final output from Eq. (4.6) as the estimate for Q. If, in the application of Eq. (4.6), we encounter a stream input that has not yet been solved for, we pass the model a zero-vector of the same shape (along with a flag in the input) in its place to make a prediction without that information, and remove any associated edges in the scene-graph GNN embedding (as discussed in Section 4.4).

If the node selected by our learned UCT estimation is a leaf node (meaning it has no children), the associated sub-problem is attempted, and, if solved, a new node is added to the tree. Then, we re-query our learned models for the remaining sequence of streams in ψ, using any new output generated by solving the previous sub-problem, and compute the learned Q-value for the new node given those new inputs. From there, the simulation and back-propagation steps are taken, and selection begins again. By growing the tree in this way, we are biased to evaluate sub-problems that are determined by our learned cost to be the most likely to lead to a satisfying plan in the shortest amount of time, balancing exploiting high value branches, and exploring new solutions to account for potential inaccuracies in the learning.

4.5.4 Progressive Widening with Learned Cost

One additional challenge in stochastic search for continuous parameters is the question of when we choose to compute a new solution to a sub-problem rather than expanding the search tree from an existing solution at a child node. As mentioned previously, each stream could be called infinitely many times and potentially produce new outputs. For example, each time a grasp sampler is queried, we should expect it to return a new grasp object. As such, during tree traversal, we must decide at each node if we are going to query the stream for a new set of groundings for its output and create a new child node, or if we are going to select a child node with an existing assignment. If we rely on the standard UCT heuristic, we would always sample new nodes during the selection phase of tree traversal, due to the fact that each new action has never been tried before. To avoid this problem, traditionally, PW-UCT search defines a formula for sampling new child nodes based on a pre-set schedule [139], [140]. In Ren, Chalvatzaki, and Peters [37], this decision is governed by the progressive-widening inequality [140]: $N(v)^{\alpha} > (N(v)-1)^{\alpha}$, where $N(v)$ represents the number of visits to a node v, and α is an exploration constant. If the condition is true, then a new child node is created from node v by querying the associated stream. If not, and multiple children exist from a particular node, the UCT equation (4.7) selects the next node as described above.

We propose one further alteration to PW-UCT, taking advantage of our ability to estimate the expected cost of the remaining sequence of streams. Instead of relying on the progressive-widening heuristic as in Ren, Chalvatzaki, and Peters [37], we simply consider the act of sampling a new node as another action in Eq. (4.7). If the UCT heuristic for querying the current stream again—according to the estimated cost from (4.6)—is higher than that of any the available children, we do so, and add a new child to the current node. This allows us to completely forgo the need to rely on the hand-tuned progressive-widening heuristic, and guide our search purely through the predictions of our learned models. Because the exploration factor in Eq. (4.7) ensures we will eventually re-sample every node, we retain the guarantee that we will eventually expand our search breadth with this method.

It should be noted that we also use this approach to decide when to query a new stream plan ψ from the root node. In that case, the new plan that is selected is the one with the lowest estimated cost of those in Ψ. As a result, we are able to explore the highest value (lowest cost) stream plans first, and only consider new ψ's when the value of exploration is high as determined by UCT. Once a full sequence of sub-problems has been successfully solved, we have found a concrete plan which satisfies the original TAMP problem, and can begin execution. In the next section, we demonstrate the benefits of using our approach experimentally.

4.6 Experimental Results

Table 4.1: **Experimental Results**: All units in seconds (% cost reduction)

	Kitchen Domain		**Unpack Domain**		**Real Domain**	
Cost $\downarrow$ **(**$\uparrow$**)**	**eTAMP**	**Ours**	**eTAMP**	**Ours**	**eTAMP**	**Ours**
Plan Time	98.3	39.5 (60%)	105.8	54.3 (49%)	66.7	26.2 (61%)
Motion Time	86.4	82.1 (5%)	46.7	46.0 (1%)	21.3	20.9 (1%)
Total Time	184.6	121.7 (34%)	152.6	100.4 (34%)	88.0	47.1 (46%)
Expansions	149.8	55.2 (63%)	707.2	125.9 (82%)	91.2	11.0 (88%)

To highlight the capabilities of our learned planner, we implement our approach in two simulated scenarios, as well as on a real robot. To make comparison straight-forward, the simulated experiments use two problem settings tested in *eTAMP*; specifically, their 'kitchen' and 'unpack' domains [37]. We compare our planner against the heuristic-driven planner defined in *eTAMP*, using the hand-defined parameters, tuned for each environment as specified by the authors. In each instance, we demonstrate that our planner is able to out-perform the baseline [37] in both planning time and the number of search nodes expanded, while finding plans of equivalent motion cost as shown in Table 4.1.

4.6.1 Kitchen Domain

In the 'kitchen' domain, we consider a simulated PR2 robot, shown in figure 4.5 with five available actions: pick up an object, place an object on a platform, move between configurations, cook an object, and clean an object. An object is cleaned when placed on the sink and cooked when placed on the stove. The agent is then tasked to first 'clean', then 'cook' all blocks initially placed on a table, being careful to avoid overcrowding any platform. We consider four distinct sub-problems, specifically a grasp sampler, a stable placement sampler, an inverse kinematics solver, and an RRT motion-planner between configurations. The full domain and stream files for this setting are presented in Appendix A.2.

For this task, most action plans considered by the planner are feasible, though deciding which sub-tree to explore in search is difficult. Notably, the order in which blocks are moved

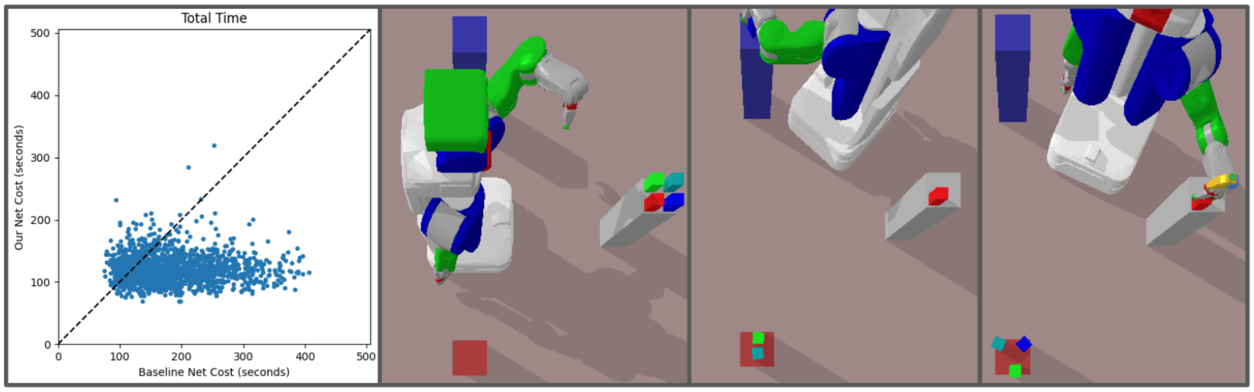


Figure 4.5: A comparison between the total cost of planning and execution of the baseline planner [37] and our learned TAMP planner for 1600 trials in the 'kitchen' domain. Each point in the scatter-plot represents the outcome of a single trial. The images on the right demonstrate potential failure modes in this domain. If the first few blocks are placed poorly on the stove, it may be impossible to safely place all four blocks there without risking collision. Our approach allows us to predict when a block placement will lead to failure later in a plan, reducing the time spent planning in these sub-trees.

is generally irrelevant to the feasibility and cost of the problem. Therefore, at the root node, our learned models predict approximately the same cost for each possible sequence of sub-problems ψ. However, as the search tree grows in depth, and blocks are added to the final platform, we are able to improve upon naive search by considering how cluttered the surface is, and if a particular sampled pose is feasible. Because we are able to more accurately estimate the cost of querying our solvers as more sub-problems are solved, we can guide search to select the sub-problem in the tree that is most advantageous to re-sample.

In this domain, we ran 1600 trials for both the baseline planner [37] and our learned planner (trained on 100 trials worth of data collected by running the baseline planner), initializing each trial with a new random seed. We recorded both the planning and execution time, and plotted these values for the individual trials (along with their sum) in Fig. 4.5 and Table 4.1. As shown in the table, we demonstrate a mean reduction in total time of approximately 63 seconds (a 34% improvement). We demonstrate that our approach can find plans of similar execution time to the baseline with fewer average node expansions (150 vs 55). This domain highlights the importance of updating our predictions of cost as well as feasibility when determining which streams are actually evaluated in TAMP solvers, even in contexts where most abstract plans are potentially feasible.

4.6.2 Unpack Domain

In our second environment, we consider a table-top manipulator with the ability to pick and place objects on different platforms. As before, to pick or place an object our agent must sample grasps or placement poses, find collision-free configurations, and compute safe trajectories between these configurations. One additional difference in this domain from the "kitchen" scene is that we define separate streams for motion planning while the robot's hand is empty, and when it is holding a block. In this domain, our goal is simply to move a specified object from one platform to the other. However, depending on the configuration of the other objects in the scene, this may not be immediately feasible. To solve the task, the robot must determine which objects must be moved out of the way before it can safely pick up the target object, then find a plan which does so.

Whereas in the 'kitchen' domain, nearly every high-level plan could be valid depending on the object groundings, in the 'unpack' domain, many of the abstract action plans returned by the top-k planner are infeasible depending on the orientation of the blocks in the scene. For example, if the taller blue block sits beside the green block, the planner will be unable to find a configuration to grasp the green block without coming into collision with the blue one. As such, any calls to our IK solver will fail, and the ability to reason about which queries to to an external planner will or will not succeed can be very impactful in terms of accelerating planning. We highlight a few examples of this in Fig. 4.6.

During training, over 200 trials, we considered instances with either one, two, or three blocks in the scene, with the initial poses of the blocks randomly selected (but closely clustered). We then evaluated our planner for the case of three blocks, running 400 trials for both the baseline planner [37] and our learned planner. Once again, we recorded both the planning and execution time, and plotted these values for the individual trials in Fig. 4.6 and Table 4.1. As shown in the table, we demonstrate a mean improvement in total time of approximately 52 seconds (a 34% reduction in total time). In addition to comparing the wall-clock time of planning and execution in this environment, we also highlight the difference in the number of nodes expanded by our planner vs the baseline planner. We average approximately 126 nodes expanded per search, whereas the baseline explores on average 707

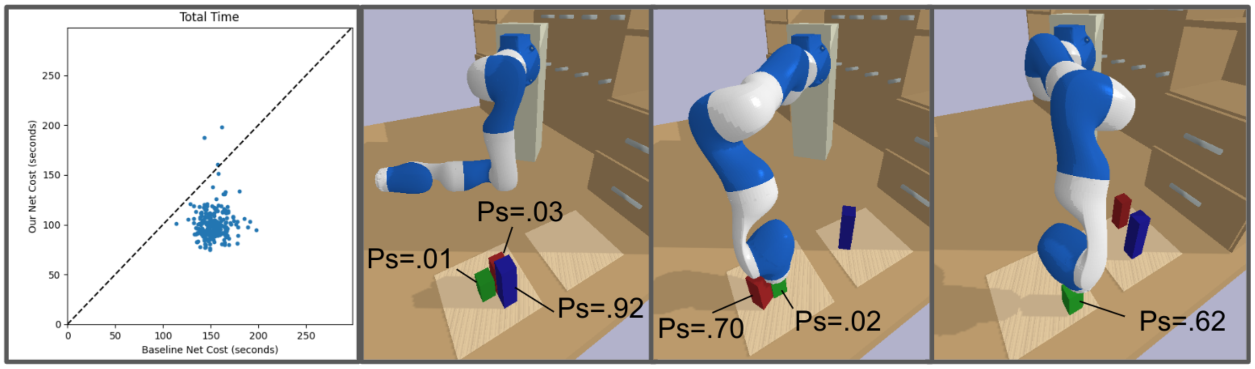


Figure 4.6: Comparing the total cost of planning and execution for 400 trials in the 'unpack' domain, where once again each point in the scatter-plot represents the outcome of a single trial. We also highlight some example scenes, where we show the feasibility predictions given by our IK model for attempting to grasp each block on the first platform. As more blocks are cleared, the predicted likelihood that we can successful find a collision-free configuration to grasp the green block increases.

nodes per trial. We might expect that by expanding fewer nodes, we produce less efficient plans. On the contrary however, our execution time is approximately the same as for the baseline (in fact we show a slight improvement), indicating that our approach is able to avoid spending computation in stream plans that are infeasible without diminishing plan quality.

We further compare our approach to one (not included in the table) which uses predicted feasibility as a threshold to prevent the planner from exploring low-probability actions [46]. In that work, the authors report an improvement of 63% over the same baseline in terms of motion planning time only in the 'unpack' domain, which does not include time querying the top-k solver for Ψ. Using our approach, we found a savings of approximately 67% in this metric. Moreover, because this approach thresholds certain sub-trees from ever being considered, the planner fails to find any plan $\sim 8\%$ of the time, whereas we found no failures over 400 trials. We do note that we did not re-implement and test this approach ourselves, and are relying on the reported values.

4.6.3 Real World Experiments

Finally, we implement our planner on a real Panda manipulator (see figures 4.1 and 4.7), testing a modified version of the 'unpack' problem, where the robot is tasked with grasping a particular block in a crowded grouping, potentially having to remove obstructions before it

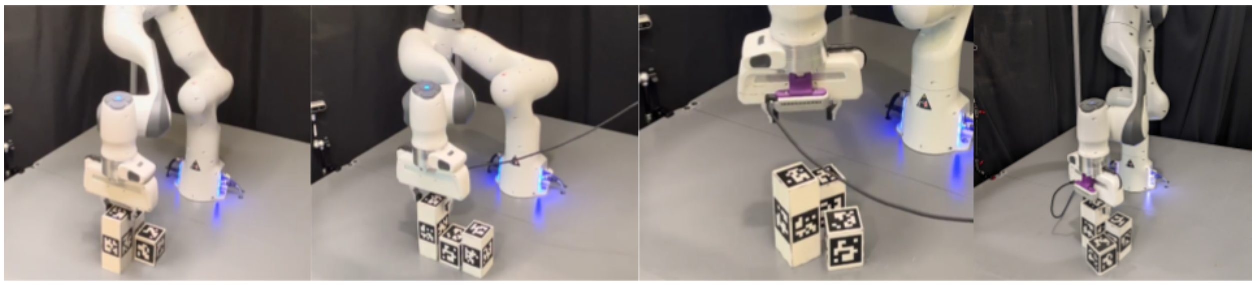


Figure 4.7: Here we highlight four (4) example initial configurations for our real world experiments. Notice the RealSense camera affixed to the Panda robot, enabling real time updates to pose estimation of the blocks.

can safely reach its target. A Panda arm is equipped with a parallel gripper, and a RealSense camera, which it uses to identify objects, their positions, and their shapes prior to planning. We define grasp and placement samplers, as well as an RRT motion planner, and an IK solver as the relevant sub-problems. See Appendix A.1 for the full domain and stream files used for this experiment. We trained our models for each sub-problem on data from 100 simulated trials, then test on the real robot.

Across 10 real world trials, we compare our planner to the baseline approach for identical initial conditions. For each trial, first the robot identifies the blocks in the scene and their poses. Then, we search for a plan to grasp the selected block using our approach. Finally the agent executes this plan, using the mounted camera to account for perception errors during execution as needed. In each trial, our planner outperforms the baseline with respect to planning time and nodes expanded, while producing plans that are of equivalent quality in terms of motion cost. An example of this scenario is shown in Fig. 4.1 and Fig. 4.7, and we report the results of the trials in Table 4.1.

It is notable that our learned models for this real experiment were trained in simulation. This is possible in part because the method by which we encode our scene is purely based on the scene graph representation defined in section 4.4, where each node only contains vectorized information related to an object. Alternatively, if we had informed our models with image data, it would be much more difficult for our models to generalize from the sim environment to the real world.

4.7 Discussion

In this work, we proposed a novel approach for planning in long-horizon TAMP problems. In our method these abstract plans come from a PDDLStream encoding of a TAMP problem, which separates discrete decisions like: “which block should the robot pick up” from low-level continuous ones such as: “how should the robot pick that block up.” The insight of these kinds of approaches is that we should delay attempting to solve for low-level trajectories until a high-level plan which might solve the full problem is found. In general, this is a good idea; solving sub-problems in TAMP can be expensive, and we do not want to waste time solving ones that will never be useful to a full plan. However, we often encounter circumstances where the geometry of the scene implies that many sub-problems are infeasible, for example: trying to place an object on a crowded table.

What makes traditional planners inefficient in these contexts is the fact that they make an implicit assumption that we can potentially refine every high-level plan to a low-level trajectory. In reality, we should be able to learn from experience which sub-problems are likely to fail depending on their parameters, and the associated costs of attempting to solve them. Building upon the intuition of our PO-TLP from Chapter 3, we propose learning models to guide search to refine high-level, abstract plans, allowing us to reason about the imperfection within our TAMP abstraction. A major difference in the approach in this work is that we attempt to minimize both the cost of execution and of planning, which has a far greater impact on TAMP problems. To that end, we derived an equation (Eq. (4.2)) to use these models to compute the total cost of attempting to ground, then execute a given abstract plan.

The underlying problem in this work is not a POMDP as we saw in Chapter 3. Therefore, we do not need to account for uncertainty in the same way, and are more easily able to solve this equation efficiently over a set of several abstract action plans. As we attempt to search for concrete action plans, we can repeatedly re-solve for the expected total cost from each node in our search tree, updating our learned predictions (and therefore cost estimates) as upstream sub-problems are solved. A major contribution of this work is our stochastic planner, which executes this search, and guides our agent to attempt to solve sub-problems which are components of plans that are on the whole more likely to succeed (and be of lower cost) than alternatives.

We demonstrated that our strategy is effective across various problem settings, achieving improved performance with respect to planning time across three domains, including one on a real Panda robot. In the first simulated environment, we showed that our planner is able to succeed in settings where most high-level plans are potentially refinable, though early choices of action parameters affects that feasibility greatly. We then considered a domain where many high-level plans are certain to be infeasible depending on the initial scene geometry, and were able to show our approach outperformed heuristic driven baselines there as well. Finally, we showed our method can be implemented on a real robot and produce low-cost plans efficiently.

As expected, though we saw impressive results experimentally, there are several limitations to our approach, which we discuss below:

- **Limitations in learning:** in order to make predictions about the feasibility and costs of different actions, we utilize a GNN to encode the geometry of the scene to our learned predictors. Our network takes in a vectorized representation of objects, which was useful for understanding properties relevant to the settings we considered. However, this encoding did discard information which might be relevant to other problems, such as semantics, which could require visual input to deduce. In order to make predictions from visual input at future steps of a plan, we would need some way of updating the original image of the scene to incorporate the actions the robot plans to take. There has been some work in this space [142], though we do not address it here. Future work might also consider how we might address the issue of learning models which generalize across environments, which we discuss in greater detail in Chapter 6.

- **Limitations in planning:** Our planner uses its learned models to guide search for grounding abstract plans, but makes no effort to use those models to aid in finding those candidate plans in the first place. This leads to a "warm-up period" where the first several seconds of planning are spent finding abstract plans, many of which our models quickly identify as very expensive, and so are largely ignored. Work by Khodeir, Sonwane, Hari, *et al.* [81] has a method for using feasibility predictions to guide search for the abstract plans, which we can incorporate into our approach for a more efficient overall planner.

- **Limitations in execution:** Our planner makes a few key assumptions, principally that any plans which are found are executable. In the simulated scenarios we considered, this was a reasonable assumption. However, using the real robot, we found errors in perception led to failed execution rather frequently. Specifically, we saw grasps where the block slipped from the gripper, the gripper hit the intended block during the grasp, or another block was knocked to a different location during pick-up, leading to a failed future grasp. As a result, we were forced to utilize the RealSense camera during execution to get an update on the target block's pose, which improved execution significantly, but was still imperfect. If failure of a found plan is possible, we should account for it in our computation of cost. This is certainly possible, and is one avenue for future work.

The novel research in this chapter demonstrates an approach for efficient decision making in TAMP domains. TAMP problems require a solution method which breaks down the problem hierarchically, separating discrete decisions like which object to grasp from the continuous parameters grounding high-level actions. This hierarchical approach is necessary, as solving the full TAMP problem at once is too complex for sufficiently interesting tasks, however it introduces a challenge where the high-level planner is divorced from the low-level information which determines if its decisions are ultimately feasible. To account for this, we learn properties related to these high-level actions (in the form of feasibility and cost of their sub-problems), which allow our planner to explicitly reason about the imperfections of its planning abstraction.

We have shown this general approach to lead to more efficient planning now in two different settings. First in partially observed environments (Ch. 3), and now here in the domain of TAMP. However, we have not yet considered how we can build our planning abstractions directly from perception, nor have we addressed the implications of increasing scale on our planners. Unsurprisingly, these elements introduce challenge to our system which must be addressed. In the next chapter, we look at these two factors, extending TAMP to real-world, building scale mobile manipulation.

Chapter 5

Building and Planning within Large-Scale Hierarchical Abstractions

The goal of the work in this thesis is to develop approaches which enable robots to solve complex tasks in large-scale, real world environments autonomously. Over the previous two chapters, we proposed a pair of related approaches for planning in different settings. For each problem we learned how to accelerate planning within imperfect abstractions, first for the case of complex tasks in partially revealed environments (Chapter 3), then in high-dimensional Task and Motion Planning (TAMP) problems (Chapter 4). Notably however, each approach for planning we have discussed so far can be limited by environmental complexity. In Chapter 3, as the maps grew in size, we spent proportionally more time planning trajectories at the lowest level, leading to fewer nodes expanded in the search tree, and worse plans. We addressed planning in known space to some degree in Chapter 4, where the TAMP abstraction enabled greater planning efficiency. However, we saw that as we added more objects to the scene, the branching factor of search increased proportionally. In either case, eventually the space of possible plans becomes too large to search effectively in a reasonable timeframe. If we want to extend our TAMP work to larger, more complex environments, we need a way to build abstractions from perception that lead to tractable planning problems.

The work in this chapter therefore aims to enable autonomous agents to solve large-scale TAMP problems in real-world environments. As discussed in Chapter 4, in order to solve a TAMP problem efficiently, an abstract planning domain is needed, which accurately represents the robot's environment and the available actions. We did not address in the previous chapter how we can build these abstractions on a real robot, and so consider that problem in part here. Recently, significant progress has been made in the area of generating hierarchical metric-semantic representations of the world using 3D scene graphs [72], [73]. These environmental abstractions lend themselves well to large-scale planning problems, as they are capable of storing both higher-level abstractions such as objects along with the connectivity of regions needed for task planning, as well as the low-level metric information required to check kinematic feasibility of different actions. The incentives when building these representations align with including as much information as possible into the scene graph. Every object included in the scene makes our graph more accurate, and thus a better representation of reality for any downstream use. Similarly, the higher the resolution with which we can represent the environment, the better chance an agent has to localize accurately.

However, as a planning problem instance grows in the number of objects and regions, so too does the complexity of finding a plan. TAMP is PSPACE-Hard [45], so problems can become computationally intractable very quickly as the sizes of the state and action spaces grow [19]. To create tractable planning problems when converting a 3D scene graph into a planning domain, it is critical to leverage the graph's structure and identify which elements of the environment are potentially relevant to the given planning problem. Consider, for example, a robot responding to a Chemical, Biological, Radiological, Nuclear, and Explosive (CBRNE) scenario, receiving instructions to inspect and neutralize dangerous objects scattered in a large area, represented as a scene graph. Let us stipulate that our robot can pass near an object only after it has neutralized and cleared it, and that the robot may be instructed to avoid particular regions entirely. Depending on the geometry of the scene and the specified goal, for the set of tasks available to the agent, only a subset of these dangerous obstacles and regions may ultimately be relevant to finding a plan. However, for a robot building a scene graph representation from perception, it is not at all obvious which elements of the graph should be added to a planning domain to ensure a valid plan for can be found and executed *a priori*.

Previous approaches to the problem of inferring a task-relevant planning domain have

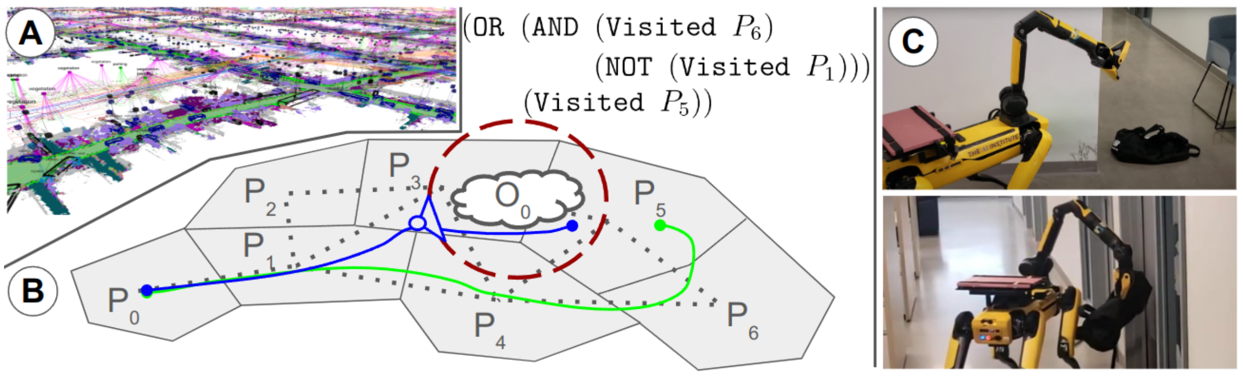


Figure 5.1: An illustration of how we derive and encode tasks in our planning representation from a 3D scene graph. **(A)** An isometric view of a Hydra scene graph generated from the KITTI dataset, giving an insight to the scale of the environment. **(B)** A simplified version of this scene, where the agent is tasked with either visiting Place 6 while avoiding Place 1, or visiting Place 5. We see that Place 5 is partially obstructed by a suspicious object, so the agent must consider either avoiding it (green trajectory), or inspecting and neutralizing the object (blue trajectory) to reach its goal. **(C)** A mobile robot (which we used to build our scene graphs) executing a plan in the real world, inspecting an object in the top frame, and moving an obstruction out of its path on the bottom.

relied on representations of connectivity in the scene graph to prune superfluous elements [50], however, these efforts have been limited to specific kinds of task planning problems. The reason for this is that the pruning approaches employed by these methods often remove information necessary for checking the geometric feasibility of plans, or they implicitly limit the types of goals that can be specified. For example, Agia, Jatavallabhula, Khodeir, *et al.* [50] consider pruning all elements which are not specifically referenced in the goal or do not have an "ancestor" which is so referenced. This leads to a very sparse graph, which is unfortunately only useful for tasks and environments where all navigation actions are assumed to be geometrically feasible. Alternative approaches for reducing the planning problem size involve attempting to learn the relevance of planning objects, then incrementally adding objects to the domain according to the learned relevance score until the problem is solvable [49]. Unfortunately, this approach requires training on numerous similar planning problems, and is difficult to generalize to tasks at large scales in the real world.

In this chapter, we propose a novel approach to both enable and accelerate TAMP in large environments (Fig. 5.1). Our first contribution is a three-level hierarchical planner for planning in large domains derived from 3D scene graphs. Specifically, we define 1) a high-level discrete planner which reasons over a carefully pruned planning domain, 2) a mid-level navigation planner which utilizes the structure of the scene graph to accelerate planning over long distances, and 3) a low-level planner which solves for trajectories, guided by the abstract plans found at the higher levels. Subsequently, we present the formulation of a sufficient condition for removing symbols from a planning problem while maintaining feasibility, which can greatly reduce computation when planning. This condition shows that many of the places in a 3D scene graph can be ignored when formulating planning problems that factorize according to our three-level hierarchy. We then introduce a technique for reasoning about whether the sparsified domain matches the original intent of the planning task, which reveals extra constraints that must be imposed on the motion planner. Finally, we develop a method to further accelerate planning by incrementally identifying objects in the scene as relevant during search according to how the geometry of the scene affects the feasibility of certain high-level plans.

As stated above, the contributions of this chapter allow a robot to solve mobile manipulation tasks in large environments. Specifically, we address task and motion planning problems where the robot does not have a complete description of the world ahead of time. As such, we not only have to worry about how to build a model of the world from real perception, but we must also ensure that our representation is not so large and full of detail as to make planing difficult. The techniques described here are therefore focused on building an abstract model that allows efficient TAMP without sacrificing completeness. Our planner enables an agent to take in a complex command specified in the Planning Domain Definition Language (PDDL) and return a plan to execute a series of actions which satisfy that goal efficiently over long horizons. These actions can be as simple as goal directed navigation, or as intricate as inspecting objects, or moving them across large distances. The approach presented here is most useful as the complexity of both the given task and the environment grows, as we are able to ignore information in the scene which is irrelevant to finding a good plan. In the following sections, we show the effectiveness of our contributions by demonstrating faster planning when compared to a baseline across two hand-crafted domains, two scene graphs built from real perception and planning in simulation, and finally a real-world mobile manipulation task on a Spot robot.

5.1 Task and Motion Planning in 3D Scene Graphs

In this section, we introduce how we encode TAMP problems, review the 3D scene graph structure that we leverage for grounding planning problems, and finally propose a CBRNE-inspired planning domain as an example of formulating a planning problem based on a 3D scene graph.

5.1.1 Task and Motion Planning Preliminaries

We introduced the concept of TAMP in Chapter 2.1.2, and further discussed some of its challenges in Chapter 4. Here we briefly re-define a few terms (and introduce several new ones) that are particularly relevant to the contributions presented later in this chapter. Given the scale and density of information in 3D scene graphs, many planning approaches will represent the problem of decision making within them as a top-down task planning problem. As previously discussed, a common formalism for encoding task planning problems is the PDDL. In a PDDL problem, a set of facts defines a*state* $\mathcal{I}$, where each fact is an instance of a boolean function called a predicate $p(\bar{x}) \in \mathcal{P}$. Each predicate is parameterized by a tuple of symbols $\bar{x} = [x_1, \ldots, x_k]$ from a given set of symbols $x \in \mathcal{O}$, where each symbol x_i is a discrete representation of a state variable. Actions $a(\bar{x}) \in \mathcal{A}$ define how we transition between states, and are also parameterized by symbols. These parameters are expressed as two sets of predicates: preconditions $\text{Pre}(a_i)$ and effects $\text{Eff}(a_i)$. Preconditions determine if we can take an action from a particular state $\mathcal{I}$, while effects define the set of facts that are added ($\text{Eff}^+(a_i)$) or removed ($\text{Eff}^-(a_i)$) from the state $\mathcal{I}$ when that action is taken. As before, we define a planning *domain* by a lifted sets of predicates $\mathcal{P}$ and actions $\mathcal{A}$. We further define a problem *instance* $P = (\mathcal{P}, \mathcal{A}, \mathcal{O}, \mathcal{I}_0, \mathcal{G})$ by combining a domain with an initial state $\mathcal{I}_0$ and a set of goal states $\mathcal{G}$, parameterized by symbols $\mathcal{O}$.

Solutions to PDDL problems take the form of a sequence of parameterized action instances $\pi = [a_1(\bar{x}_1), a_2(\bar{x}_2), ..., a_n(\bar{x}_n)]$ [19]. For any action sequence, there is a corresponding sequence of states $\mathcal{I}_\pi = [\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_n]$, leading from the initial state to a goal state that can be constructed from each action's effects. We will use the fact that only a subset of state symbols are needed for each action to enable a factorization of the planning problem in Section 5.2.1. For an action plan π, its corresponding state plan $\mathcal{I}_\pi$ is *valid* if $\mathcal{I}_i \in \text{Pre}(a_{i+1})$ for $i = 0, ..., N-1$, and $\mathcal{I}_N \in \mathcal{G}$. A range of solvers [28], [29] can solve tasks specified in PDDL, and any state plan found by such a solver is valid by construction. A feasible planning problem is one for which there is a valid solution.

The continuous nature of TAMP problems, coupled with the scale of environments we consider here, make discretizing and encoding a planning problem directly in pure PDDL difficult. Those approaches that attempt to do this do not attempt to solve for low-level trajectories at planning time, and instead assume the can be refined later. This assumption allows for efficient planning, but simply does not match reality; there are many instances where the geometry of the scene might might certain actions infeasible. We therefore again use PDDLStream [36], as we did in Chapter 4 to represent and solve TAMP problems. As stated in the previous chapter, a PDDLStream problem instance $(\mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{O}, \mathcal{I}_0, \mathcal{G})$ represents the discrete search portion of a TAMP problem in PDDL, as a set of predicates, actions, symbols, initial state, and set of goal states, but also introduces the notion of *streams* $s \in \mathcal{S}$, which can query external solvers (e.g. a motion planner) during search to produce new symbols and facts within the problem instance. Streams make the problem encoding more efficient, as they obviate the need to crete an object and evaluate the predicates for all possible continuous values of a symbol. PDDLStream solves[1] problems by first finding an optimistic solution that satisfies the domain's symbolic constraints – a *task skeleton* – and then attempting to solve for feasible continuous parameters. We once again refer the reader to Garrett, Lozano-Pérez, and Kaelbling [36] for a detailed description of PDDLstream.

In a TAMP problem, each symbol x_i can represent a continuous value (e.g., a pose), and the grounded parameters of an action depend on these values. From these parameters, we can derive a *motion sequence*, which specifies how a robot executes an action. For example, from an action plan composed of a sequence of `move` actions, the corresponding motion sequence would be composed of the trajectories that were solved for by the motion planner and describes the continuous values of the parameters $\bar{x}_i$ for each `move` action. Executing that sequence involves multiple calls to a trajectory controller, where two motion sequences

[1]Specifically, the PDDLStream *adaptive* solution algorithm.

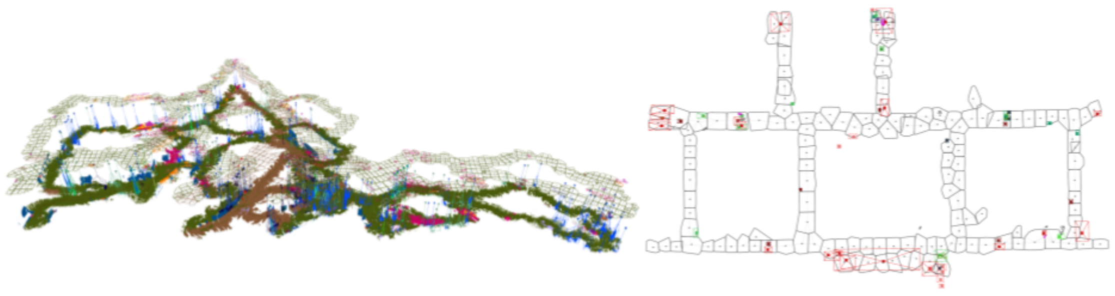


Figure 5.2: Two examples of a Hydra Scene Graph from different environments. One the left we see an isometric view of a scene graph built outdoors over a large length scale. At the lowest level we see the metric semantic mesh, upon which we cluster places and objects to form higher levels of abstraction. On the right is a top-down view of the places and objects of scene graph built in Building 45 at MIT. Both examples were built with a Spot robot.

are equivalent if they result in the agent acting identically.

5.1.2 Building 3D Scene Graphs from Perception

We desire robots which are able to build planning representations entirely from real world perception. We assume our robot is equipped with a prior model of its own state and a motion controller, as well as the ability to use its sensors to build a dense geometric model of its environment and the objects in it. To derive a discrete, symbolic model of the geometry, we take advantage of recent work in 3D scene graph mapping [72] that infers a discretization of the geometry and the objects in the geometric model. While our approach is compatible with a range of scene graph implementations, our definition of a 3D scene graph, directly based on Hydra [72], [75], consists of several layers of increasing abstraction (see Fig. 5.1). Each layer consists of a collection of nodes representing location and other attributes, with edges connecting nodes within the same layer representing relative spatial constraints and edges between different layers representing an inclusion relationship. The lowest layer of the hierarchy is a semantically-annotated *mesh* of the scene geometry. The next layer contains *objects* and their locations identified by a semantic image segmentation. The *places* layer represents navigable regions of the environment based on semantic and geometric properties of the mesh. Places are clustered into groups based on geometric and semantic information, and these groups become nodes in the higher-level *regions* layer (e.g., rooms in an indoor environment). Hydra can construct this map representation in realtime from RGBD sensor data while accounting for odometry drift, enabling large scale, consistent, and information-rich maps.

Previous work on 3D scene graphs has mainly focused on indoor uses. These representations rely on the Generalized Voronoi Diagram (GVD) [143] to generate places, an abstraction of 3D spatial connectivity, which are not well suited for ground robot navigation. We use an alternate formulation of 2D places in our navigable scene graph, where each place represents a 2D polygon with consistent terrain classification, representing an area the robot may traverse (Fig. 5.1B). As the resolution of the places is much coarser than the mesh resolution, planning over sequences of places can be much faster than planning over the mesh itself, while still retaining important geometric information. In section 5.3.4, we pilot a Spot robot around MIT's building 45 and build a Hydra scene graph. Figure 5.2 highlights the different layers of the resulting representation.

5.1.3 Inferring the Planning Domain from Scene Graphs

Given a Hydra scene graph, we are now able to introduce a framework for deriving a TAMP problem instance. We will use this abstraction to to demonstrate the salient aspects of solving planning problems based on large-scale environments. In general, we define a symbol $x \in \mathcal{O}$ for each node in a scene graph, as well as a symbol corresponding to the robot. This means each *object*, *place*, *region*, etc. are all represented as symbols which can be combined to form predicates in the domain. Specifically, we define six classes of predicates, derivable from a Hydra scene graph, to include in our problem and which may be relevant for planning:

1. Type information derived from the nodes of the graph, where each node corresponds to a unary predicate: `(Configuration ?c)`, `(Place ?p)`, and `(Object ?o)`, etc.

2. Agent or object predicates that define the state of the robot and objects: `(AtConfig ?c)`, `(AtPlace ?p)`, `(AtRoom ?r)`, etc.

3. Connection predicates defined by edges in the same level of the graph: `(Connected ?n1 ?n2)`.

4. Inclusion predicates indicating edges connecting nodes of different levels of the graph: `(PoseInPlace ?c ?p)`, `(PlaceInRoom ?p ?r)`, etc.

5. Preconditions of actions that are certified by solving a stream's associated sub-problem. For example, a *move* action may require that a trajectory has been found between two configurations: `(Trajectory ?c1 ?t ?c2)`.

6. Finally, problem-specific predicates defined by the user to specify goal states and problem constraints such as which places to visit or which objects to collect.

As a running example, we define an example problem using these predicates, motivated by CBRNE scenarios, which we name the "Inspection Domain." In this setting, an agent can be commanded to visit or avoid certain places, and inspect and neutralize objects that have been marked as suspicious. The robot cannot move past a suspicious object until it has been inspected and neutralized, which may mean (depending on geometry) that certain navigation actions are infeasible without first inspecting select objects. We therefore define the problem specific predicates: `(VisitedPlace ?p)` which indicates the current and past places the robot has visited, and `(Safe ?o)` or `(Suspicious ?o)` which identifies whether an object has been inspected or not. Goal specifications in this domain can include positive or negated facts based on these predicates. The agent's available actions are to `move` between poses in connected places, and to `inspect` objects from appropriate poses (for simplicity we do not separate the inspect and neutralize actions). Note that only the `move` action need be parameterized by a place symbol, which we present here:

```
:action move
    :parameters (?p1 ?p2 ?c1 ?c2 ?t)
    :precondition (and
              (Trajectory ?c1 ?t ?c2)
              (PoseInPlace ?c1 ?p1)
              (PoseInPlace ?c2 ?p2)
              (Connected ?p1 ?p2)
              (AtPose ?c1)
    :effect (and (AtPose ?c2)
                 (not (AtPose ?c1))
                 (VisitedPose ?c2))))
```

This instantiation of the `move` action allows the agent to navigate between its current place p_1 and any place p_2 that is connected to p_1 in the scene graph, and updates the state to note the agent's current pose and that p_2 has been visited. We refer to this problem representation as the *direct encoding* of the Inspection domain. We further define streams for sampling poses for inspecting and neutralizing objects, sampling poses in a specific place, and planning motion between two poses in order to find feasible continuous parameters for a given abstract plan. Any valid plan is composed of these actions, which at execution time are converted into a motion sequence of `FollowPath`(t_i) and `InspectObject`(o_i) primitives for paths t_i and objects o_i. Later, we will also consider additional `Pick` and `Place` actions, which are defined in Appendix A.3.

The direct encoding of the problem has one major drawback: moving between places that are not physically near each other necessitates chaining together potentially many `move` actions, which requires motion planning in many short segments between neighboring places. For simple navigation, planning in short segments can actually aid in the efficiency of long-distance motion planning. However, relying on a general task planner to construct long-horizon action sequences, any segment of which may be invalidated by the low-level geometry, leads to very inefficient planning in practice. Consider, for example, the case demonstrated in Fig. 5.3, where our agent is given a complex specification, one element of which requires picking up object `O105`. This necessitates that the robot navigate near the object, however any path which might do so is obstructed by object `O110`.

In many TAMP planning abstractions, there is no *a priori* logical connection in the PDDL problem between a trajectory and the objects which might interfere with it. This is a direct result of the separation between high-level task planning and low-level geometric planning. As a result, choosing the right object to inspect in order to unblock a path is difficult for the task planner; the longer the sequence of `move` actions needed to reach a goal state, the more potential plans the solver must consider. In the case of the problem in figure 5.3, the segment of the full plan the agent finds to reach `O105` consists of approximately thirty (30) actions. As far as the high-level planner is concerned, changing any of these

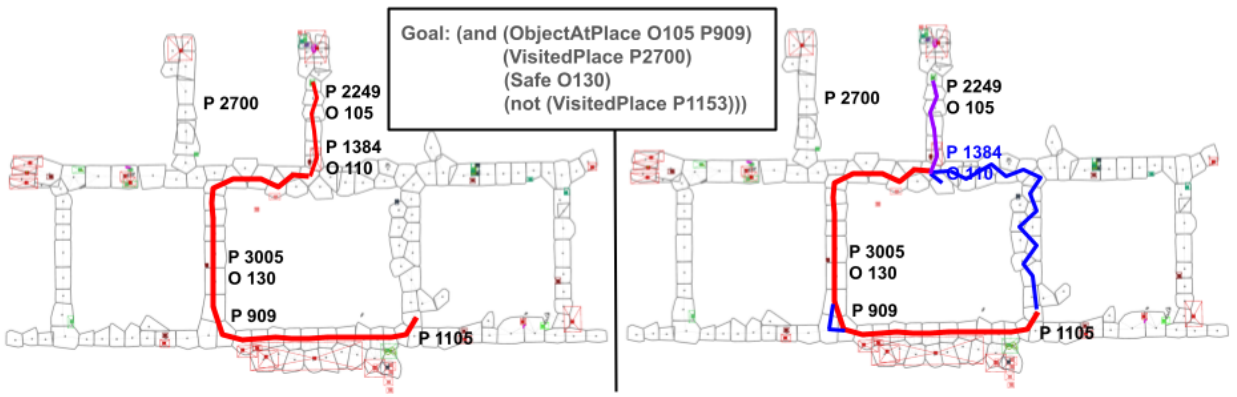


Figure 5.3: A visualization of a high-level plan derived from our `Direct` encoding, attempting to solve a subset of a task specification in the MIT building 45 environment. On the left we see the first plan proposed by the high-level PDDL planner. However, this plan is obstructed by `O110`, and so we must consider different plan skeletons. Because the task planner cannot be certain if a different sequence of `Move` actions might lead to success, we may have to consider many different abstract plans before attempting one which moves the obstruction out of the way. On the right, we highlight a few such superficially different plans.

(i.e., moving to a different place) constitutes an entirely new plan which might succeed. The planner does not know why the original plan failed, and therefore must check each new plan for feasibility, no matter how superficially similar. Given the sheer number of different ways an agent might navigate to the object of interest, it might take many planning iterations before we even consider moving obstructing objects out of the way (as shown in Fig. 5.3). This is especially problematic in large scene graphs, where any problem has a relatively long horizon, particularly as we consider increasingly complex goal specifications (Sec. 5.3). In the following section, we propose a new planning abstraction which reduced the depth of search required to find plans in these settings, as well as an approach to address its imperfections.

5.2 Scalable Scene Graph Planning

Our objective is to both enable and accelerate solving TAMP problems in large environments. One associated challenge is that the number of symbols created by a scene graph can quickly overwhelm the ability of the planner to reason efficiently due to increasing branching factor, and the depth of search needed to find plans. However, we notice that the vast majority of planning domains, and the world in general, tend to factor into sequence of navigation actions punctuated by periodic object-centric actions. This factorization allows us to identify a subset of symbols relevant for object interaction, and a subset needed to move from place to place, potentially simplifying search. We therefore propose a planning formulation that aligns with the scene graph hierarchy and naturally divides the planning problem into a high-level, task-relevant planning problem such as finding a sequence of manipulation actions, mid-level coarse navigation planning between locations, and low-level continuous trajectory planning between points.

Specifically, instead of requiring a PDDL planner to find paths through the places in the scene graph at the discrete symbolic level, we reduce the depth of the planning horizon at the highest level by reasoning only over places that are directly relevant for achieving the goal. Then, a coarse navigation planner plans through the 2D places layer to create an abstract motion plan composed of a sequence of subgoals. Finally, a fine-grained motion planner is guided by the navigation plan subgoals through the places layer, quickly finding motion plans over large distances. We discuss this in Sec. 5.2.1.

By focusing each layer of our tri-level planner hierarchy on specific types of actions, we can prune irrelevant symbols within each layer and simplify the corresponding problems by reducing the branching factor of search (Sec. 5.2.2). Critical to our approach is that the proposed factorization must not limit the types of problems that can be solved, nor produce plans which violate intended constraints. To that end, we show a sufficient condition for symbols to be removed from each planning problem while maintaining feasibility, then show how to reason about whether the resulting motion plans adhere to the original specification (Sec. 5.2.3). Finally, in Sec 5.2.4, we consider an additional heuristic that enables optimistically ignoring objects that are irrelevant due to scene geometry.

5.2.1 Hierarchical Planning

Here we describe our tri-level planning approach. At the highest level of abstraction, our planner must reason about which objects to interact with, which regions to avoid, and which destinations the robot should move to. As discussed previously in section 5.1.3, the most straightforward encoding of planning motion through a scene graph would closely mirror the connectivity of the graph, modeling the `move` action as a transition between two places that

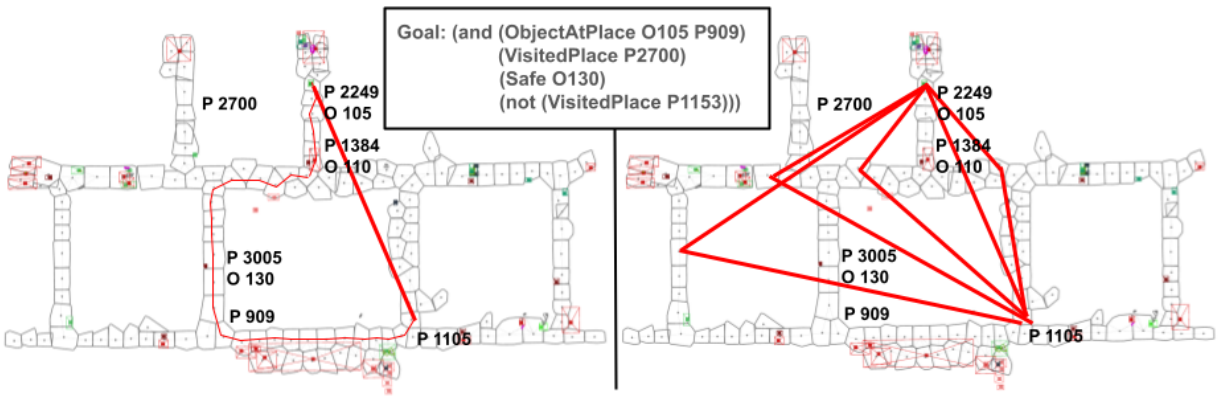


Figure 5.4: A visualization of a high-level plan derived from our `Relaxed` encoding, attempting to solve a subset of a task specification in the MIT building 45 environment. On the left we see the first plan proposed by the high-level PDDL planner, and the attempt by the lower levels to refine it to a motion plan. However, as before, this plan is obstructed by `O110`, and so we must consider different plan skeletons. In the `Direct` setting, our planner was burdoned by long planning horizons. We have solved that problem here, but now face a relality where the branching factor of our search has greatly increased. Specifically, any time we consider a move action, we could consider transitioning to any `Place` in the scene graph. Now, when the first attempt to reach the target object fails, the next set of plan skeletons will likely be to move to some other place, then a place near the object. This is because two `Move` actions are still potentially of lower cost then a sequence of `Move`, `Pick`, `Move`, `Place`, `Move`. On the right, we highlight a few such plans. In order to avoid considering these redundant options during search, we consider pruning the planning problem instance at the highest level of abstraction.

share an edge in the graph. However, as the scale of the environment increases, this direct encoding results in very long horizon plans that are expensive for the general-purpose PDDL planner to find.

Instead, we propose a more general move action: `moveRelaxed`. This action takes place p_1 and p_2 as parameters, in addition to initial and final poses c_1 and c_2 and a trajectory symbol t. The action's effect moves the robot's pose from c_1 to c_2, and marks p_1 and p_2 as visited. Notably however, there is no requirement the two places are "connected" in the scene graph. For example, consider the task in Fig. 5.4 where the robot begins in Place P_{1105}, and must eventually reach place P_{2249} to pick up object O_{105}. A motion sequence corresponding to a plan to move through dozens of specific places can be equivalent to a sequence generated from a plan to move from P_{1105} to P_{2249} directly, meaning the high-level planner need not explicitly plan to move through the full scene graph. Extrapolating this approach to larger scenes and more complex goals has the potential to vastly reduce planning horizons, although it imposes certain constraints on the lower-level planners that will be addressed at length in Section 5.2.3. The encoding of `moveRelaxed` is shown here:

```
:action moveRelaxed
    :parameters (?p1 ?p2 ?c1 ?t ?c2)
    :precondition (and (Trajectory ?c1 ?t ?c2)
                       (PoseInPlace ?c1 ?p1)
                       (PoseInPlace ?c2 ?p2)
                       (AtPose ?c1)
    :effect (and (AtPose ?c2)
                 (not (AtPose ?c1))
                 (VisitedPose ?c2))
```

Abstract plans produced by the high-level planner using this new action do not initially contain information about how the robot moves from the start to end poses. Instead, they optimistically contain trajectory symbols with continuous parameters that must be filled in by the lower-level planners. In order to do this efficiently, we rely on the 3D scene graph to accelerate motion planning. To find a motion plan between two configurations c_1 and c_2, we plan through the places layer of the scene graph, finding a sequence of places that leads from c_1 to c_2 and which respects the connectivity of the scene graph. Planning navigation at this level of abstraction enables us to take advantage of Euclidean distance heuristics to accelerate planning, while allowing for the constraints of the task (e.g., avoiding a particular place) to be encoded simply.

At the lowest level of abstraction, the planner generates a kinematically-feasible path for the robot to follow based on the reference path from the mid-level planner. This path can be generated efficiently by first considering an optimistic path that connects the waypoints on the reference path and ignores obstacles. Any segments of this path that are rendered infeasible by obstacles or violations of kinematic constraints can be re-solved by a planner that considers obstacles, such as RRT [23]. Better alignment between which edges are present in the scene graph and kinematic feasibility for the robot leads to better performance of this heuristic.

Remember that the primary challenge associated with the `Direct` encoding was the long-horizon nature of the induced planning problem. Our new `relaxed` encoding successfully reduces the planning depth significantly for the high-level planner, though unfortunately introduces a different problem. While in the direct case, the agent could only take `move` actions between adjacent places, now the high-level planner can consider moving between any two places as a valid action. Unsurprisingly, this greatly increases the branching factor of search, particularly as the number of viable Places increases. We highlight this scenario on the right side of figure 5.4. Due to the fact that our abstraction is imperfect (many high-level actions are infeasible at a low level), we must explicitly account for these imperfections to plan efficiently. In the following sections, we propose a method for pruning the planning domain at the highest level to address these issues. After describing our pruning approach, we visualize our tri-level planner in the real world in Fig. 5.7. The full Domain and Stream files for the *relaxed encoding* of the Inspection domain can be found in Appendix A.3.

5.2.2 Removing Redundant Symbols

To address the problems associated with a large branching factor, we consider reducing the size of the planning instance by pruning `Place` symbols that are not relevant to a given planning problem. Unfortunately, identifying symbols that do not impact the solution is in principle as hard as solving the problem itself, and naively removing places from a problem instance might render the problem infeasible. In this section, we characterize a set of places that we know can safely be removed from the problem before planning given the semantics of `moveRelaxed`. We begin by defining a set of symbols that are redundant for a particular goal specification.

Definition 1 (Redundant Symbol)**.** *For a set of domain actions $\mathcal{A}$ and specific goal $\mathcal{G}$, a symbol x is redundant if both of the following hold:*

1. *For every valid plan π where x parameterizes an action, there is another valid plan π' with equivalent motion sequence, where x is not an action parameter.*

2. *No action precondition or goal, expressed in negative normal form, contains a universal quantifier that can be parameterized by x.*

The intuition behind this notion of redundancy is that 1) if any plan involving the symbol yields a motion sequence that can be rewritten without the symbol, the symbol is redundant, and 2) if we solve a planning instance where a redundant symbol has been removed, the plan must remain valid in the original problem. Note that this definition of redundancy is general for any planning domain, although we will use this definition specifically for place symbols that become redundant given the `moveRelaxed` action. Importantly, removing redundant symbols preserves the feasibility of a planning problem.

Proposition 1 (Removing Redundant Symbol Preserves Feasibility)**.** *Consider a feasible planning instance* $R = (\mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{O}, \mathcal{I}_0, \mathcal{G})$*. For a redundant symbol* $x \in \mathcal{O}$*, we define a related instance* $R' = (\mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{O}', \mathcal{I}_0', \mathcal{G}')$ *where* x *has been removed, i.e.,* $\mathcal{O}' = \mathcal{O} \setminus x$ *and* $\mathcal{I}_0'$ *contains all facts in* $\mathcal{I}_0$ *except those parameterized by* x*, and similarly for* $\mathcal{G}'$*. Let* Π_R *denote the set of valid plans for* R*. Then,* $\Pi_{R'} \subseteq \Pi_R$ *and* $\Pi_{R'} \neq \emptyset$*. (Proof deferred to Appendix B.1.)*

The requirements for a symbol to be redundant are quite strong (*every* plan that uses a symbol must have an alternate plan that does not use the symbol and still results in the same motion sequence), but many places in the Inspection Domain have this property given the semantics of `moveRelaxed`. Removing these places from our task planner's domain, assuming the motion planner is still aware of them, enables the solver to more efficiently find valid plans that are guaranteed to have also been valid in the un-pruned problem. Moreover, the ability to prune these elements does not restrict the type of goals we are able to specify to our agent, preserving expressivity, while enabling planning at a larger scale.

Proposition 2 (Redundant Places)**.** *Consider a problem instance in the Inspection Domain with no quantifiers that can be parameterized by a place in the goal. A place* p *is redundant if no facts parameterized by* p *appear in the initial or goal states, or if* `(not (VisitedPlace p))` *appears as a clause in the conjunctive normal form (CNF) of the goal specification. (Proof deferred to Appendix B.2)*

We have now identified a potentially large (depending on the sparsity of the goal specification) set of place symbols that can be ignored in the Inspection domain. Our explicit method of defining our problem's initial state is as follows:

Remark 1 (Problem Initialization)**.** *In light of Proposition 2, we only include the following places when instantiating a problem in the Inspection domain: 1) the initial place that the robot is in and 2) any place that appears in the goal. A place p that parameterizes a negated fact (`not` (`VisitedPlace` p)) that appears as a clause in the CNF of the goal specification can also be removed.*

We have shown that for the Inspection domain, redundant places are very easy to identify. We would like to apply the same idea to similar domains, without needing to reason from scratch about redundancy. We now characterize a sufficient condition on the planning domain structure for places to be redundant. Let $\mathcal{P}_{static}$ denote the set of predicates that do not appear as effects of any action (i.e., they can only be set in the initial state). Let $\mathcal{F}_{static}$ denote the set of facts that correspond to parameterizations of $\mathcal{P}_{static}$. Intuitively, if a domain is structured such that a place can only be parameterized by an action if certain facts hold in the initial state, then it is very easy to check whether a specific place can be used by any actions.

Proposition 3 (Sufficient Conditions for Ignoring Places)**.** *Consider a planning instance* $(\mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{O}, \mathcal{I}_0, \mathcal{G})$*, where for all actions* $a_j \in \mathcal{A}$ *except* $a_j =$ `moveRelaxed`*, satisfying* $Pre(a_j)$ *implies that any place parameterized by* a_j *is in* F_{static}*. In this case, all places that do not parameterize any facts in* $\mathcal{F}_{static}$ *or the goal are redundant.*

For example, if the Inspection domain is augmented with a "report home" action that can only be executed at a designated set of places, then these places (and no others) need to be added to the problem instance. Fig. 5.5 illustrates how we prune the planning domain in practice.

5.2.3 Execution Consistency

While our decision to use `moveRelaxed` to model motion between distant places enables faster planning, it creates a mismatch between the discrete and continuous parts of the problem. In the example in Fig. 5.6, consider that the robot was also instructed to avoid place P_{1153}. The ability to include a constraint on the goal states of the form `(not (VisitedPlace` P_{1153}`))` requires further constraints on the mid- and low-level planners. Executing `moveRelaxed`

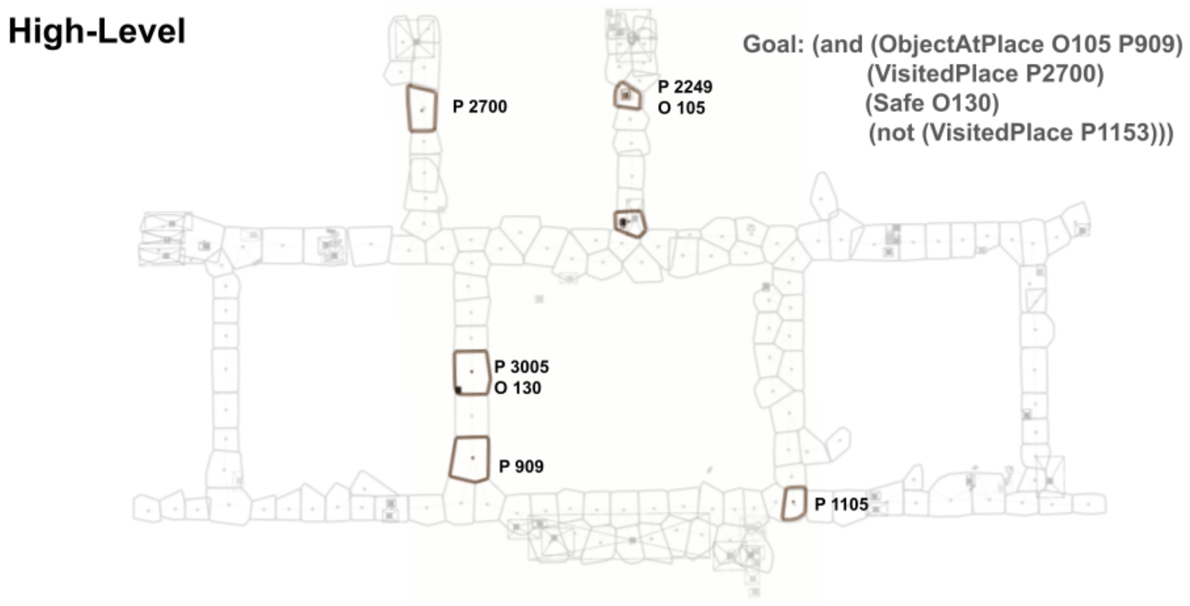


Figure 5.5: A visualization of the initial pruned high-level planning domain for the given goal specification. We overlay this domain over the original scene graph to highlight its reduction. Removing provably redundant places greatly simplifies the planning problem.

from the initial place may involve following a trajectory that takes the robot through place P_{1153}, even if the goal specifies that P_{1153} should not be visited. Technically this is still a valid solution to the pruned planning problem since place P_{1153} never appears as a parameter to the `moveRelaxed` action (and therefore `(VisitedPlace` P_{1153}`)` is not an effect), but clearly the domain with a relaxed movement action does not fully capture the intent of the original planning domain.

To formalize the discrepancy between what happens when the robot executes a motion sequence and the constraints that we expect a planning problem to impose, we introduce the concept of a *verifier function.* A verifier function maps motion sub-sequences to sets of PDDL domain facts, and "verifies" which additional domain facts would be implicitly true as a result of the agent executing a motion sequence, even if actually adding these facts to the problem instance during the solving process is undesirable computationally. Given a verifier V, the facts that hold at each step when executing a motion sequence may be different than expected in the original plan. We denote the facts that would be added by such a verifier applied to the motion sequence associated with a_i as $V(a_i)$, and term this sequence of expanded states the V-extended state plan.

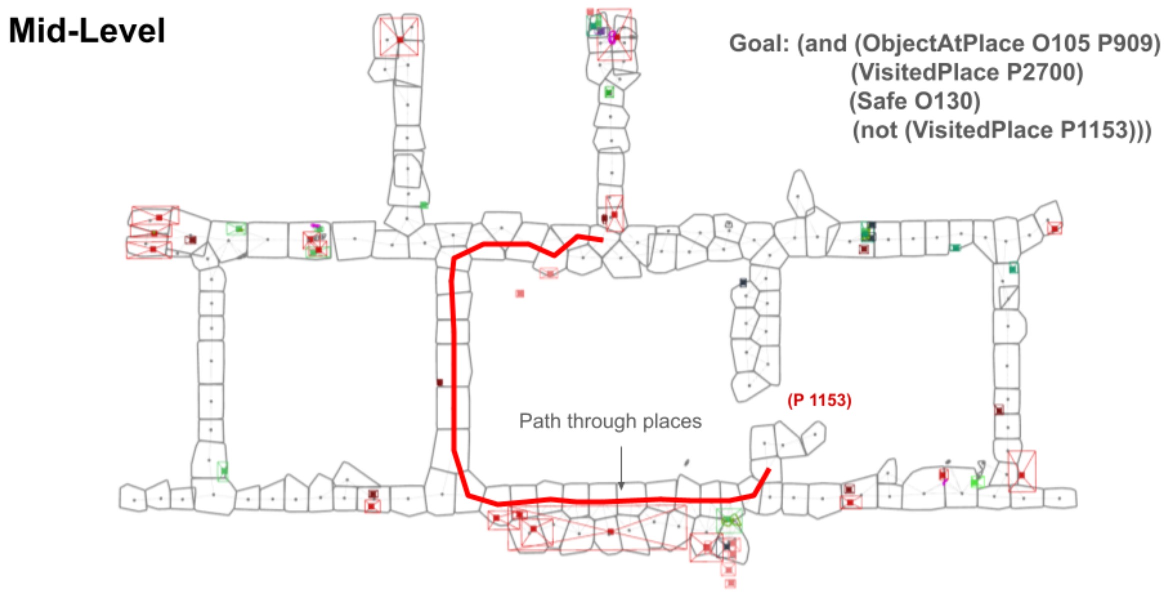


Figure 5.6: A visualization of the initial mid-level planning domain for the given goal specification. In this domain, there is little incentive to prune places, as they do not materially slow the search for solutions to goal directed navigation sub-problems. Notice however that we do remove one place: `P1153`. This place is referenced in the goal as one that we should avoid, and so we remove it as an option at this level to ensure any plans we find are execution consistent. This is how the concept of *verifier functions* are implemented in practice.

Definition 2 (V-Extended State Plan)**.** *For an action plan* $\pi = [a_1, ..., a_n]$*, its corresponding state plan* $\mathcal{I}_\pi = [\mathcal{I}_0, ..., \mathcal{I}_n]$*, and verifier function* V*, the V-extended state plan for state* $\mathcal{I}_k$ *is* $\mathcal{I}'_1 = \mathcal{I}_1 \cup V(a_1)$*, and* $\mathcal{I}'_k = \mathcal{I}_k \cup \left(\mathcal{I}'_{k-1} \setminus \mathit{Eff}^-(a_k)\right) \cup V(a_k)$*.*

The extended state $\mathcal{I}'_k$ is the state at step k as experienced by the verifier. $\mathcal{I}'_k$ is composed of the facts $\mathcal{I}_k$ in the initial plan, plus any extra facts that were present in the previous extended state $\mathcal{I}'_{k-1}$ other than those removed by action a_k, plus any facts that would be returned by a verifier applied to action a_k. As discussed in Sec. 5.1.1, any state plan found by a search algorithm is valid by construction. However, a state plan that is subsequently augmented with the extra facts that would be produced by a verifier might not be valid.

Consider a verifier V_{place} that takes a motion sub-sequence μ, and returns a VisitedPlace fact for each place that intersects with the agent's position while executing μ. For a place p and a trajectory t to be followed by the motion primitive `FollowPath`(t), we denote $p \cap t$

the section of t that intersects with p. We can then define a verifier as

$$V_{place}(\mu) = \{\texttt{(VisitedPlace p)} \mid p \cap t_i \neq \emptyset \text{ for } \texttt{FollowPath}(t_i) \in \mu\}. \qquad (5.1)$$

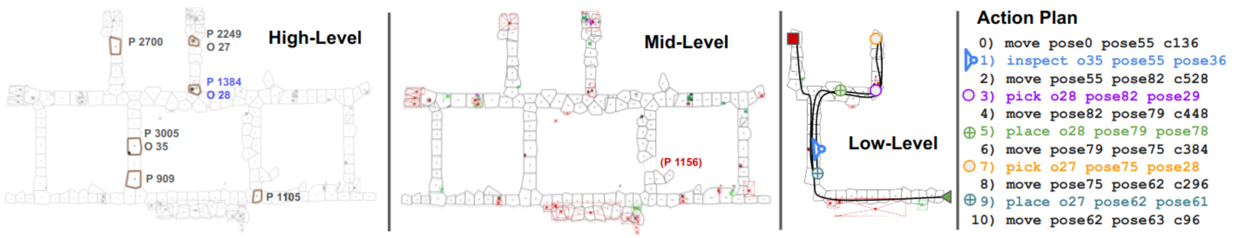
If the motion sequence associated with the action plan would result in the agent visiting a place that we do not expect, then the V_{place}-extended state plan would include a VisitedPlace fact that may conflict with the goal. If we care about the robot's motion respecting the problem's constraints on visiting certain places, then we need to prove that the V_{place}-extended state plan is a valid solution to the planning problem for any instance of the planning domain.

From this idea, we define the concept of execution consistency, which requires that solutions to the planning problem are still valid after considering the facts from a verifier.

Definition 3 (Execution consistent). *A domain is execution consistent with respect to verifier V if, for every valid plan π, the V-extended state plan is valid.*

A domain is trivially execution consistent for the empty verifier $V(\cdot) = \emptyset$, as the extended state plan is equal to the original plan. A domain is also execution consistent if the range of V applied to each motion subsequence μ_i corresponding to action a_i is limited to facts in $\text{Eff}(a_i)$. In other cases, a domain can still be execution consistent for a verifier that would introduce new facts if the planner is carefully crafted. In defining a planning domain for any task, we seek to have it execution consistent with respect to any defined verifiers. If a domain is not execution consistent, then any properties related to predicates in the verifier cannot be guaranteed to hold when executing a plan.

In our example, we want to prevent the agent from entering places that it should not, and so we should show that the Inspection domain is execution consistent with respect to the verifier V_{place}. Recall that V_{place} can only introduce new `VisitedPlace` facts. As `VisitedPlace` does not appear in any action preconditions, the only way for a `VisitedPlace` fact to render a valid state plan invalid is to conflict with the goal specification. Consider the set of places that must be avoided to satisfy some goal state: $\mathcal{P}_{avoid} = \{p \mid \texttt{(not (VisitedPlace p))} \in \mathcal{G}\}$. If a place in $\mathcal{P}_{avoid}$ can only be visited by an action that explicitly lists it in the action effects, then the domain will be execution consistent with respect to V_{place}. This can easily be guaranteed by preventing the mid and low level motion planners from generating plans



```
 0) move pose0 pose55 c136
 1) inspect o35 pose55 pose36
 2) move pose55 pose82 c528
 3) pick o28 pose82 pose29
 4) move pose82 pose79 c448
 5) place o28 pose79 pose78
 6) move pose79 pose75 c384
 7) pick o27 pose75 pose28
 8) move pose75 pose62 c296
 9) place o27 pose62 pose61
10) move pose62 pose63 c96
```

Figure 5.7: Our tri-level planner in a real world environment. A scene graph of the fifth floor of building 45 at MIT was built using the Spot robot, from which we extract our planning abstraction for the goal: `(and (ObjectAtPlace O27 P909) (VisitedPlace P2700) (Safe O35) (not (VisitedPlace P1153)))`, which instructs the agent to move `O27` to a `P909`, inspect `O35`, visit `P2700`, all while avoiding `P1153`. At the highest level, the task planner is given a very sparsified version of the scene, as highlighted above. The mid-level planner plans a path through the places guided by the abstract plan found at the highest level, avoiding place `P1153`. Feedback from this level leads to the addition of `O28` to the high-level domain, as `O27` would be otherwise unreachable. The low-level planner computes full trajectories, guided by the path found at the mid-level. The plan produced (and executed by the robot) is shown on the right.

that enter places $\mathcal{P}_{avoid} \setminus \mathcal{P}_{param}$, where $\mathcal{P}_{param}$ is the set of places that appear as parameters to the action. Our mid- and low-level motion planners are therefore constrained not to enter a place which we might want to avoid, unless that place is given as the parameter to the `moveRelaxed` action, ensuring execution consistency. Note that the verifier need not be actually implemented, but the concept can be used to prove execution consistency. We highlight how we ensure execution consistency in practice in Fig. 5.6.

5.2.4 Ignoring Irrelevant Objects

We have demonstrated the ability to identify and ignore elements of a planning instance that are redundant when searching for a plan. However, many symbols are not redundant according to our definition, but might still be safely ignored. Objects which might obstruct motion, for example, are not redundant because an "inspect" motion primitive will never be generated for an object that has been removed from the planning instance. Nevertheless, there are clearly cases when an agent can ignore objects when planning, such as an object that is not part of an agent's goal and is far from the agent's path to the goal. As with the places, ignoring objects can accelerate planning by reducing the branching factor at the highest level of search. In contrast to places however, the objects that should be ignored cannot be identified from the logical structure of the planning instance alone; the problem geometry must also be considered.

We propose an incremental approach to further identify relevant symbols. We begin by including some subset of all symbols $\mathcal{O}_S$ in the domain of the high-level planner, and then attempting to solve the planning problem. If this limited problem has a valid solution which is also a valid solution to the original problem, then we have found a plan. Otherwise, we incrementally add symbols to the planning problem, and repeat (Algorithm 2). The inner loop corresponds to solving a TAMP instance with symbols $\mathcal{O}_I$. Meanwhile, the outer loop corresponds to adding more symbols to $\mathcal{O}_I$ when we fail to find a solution.

The performance of this incremental planning approach depends on three key choices: which initial symbols are chosen in $\mathcal{O}_S$, when new symbols are added to the planning instance, and how the new symbols $\mathcal{O}_{new}$ are chosen. As long as all symbols are eventually added to the planning instance, this planning approach will maintain the completeness properties guaranteed by the chosen PDDLStream solution algorithm [36]. We outline those choices here:

- The initial set $\mathcal{O}_S$ should be as small as possible while still including the symbols necessary to find a plan. In particular, we can begin by including symbols based on the problem's logical structure. For the Inspection domain, we include the non-redundant places identified in Remark 1 and any objects that appear in the goal. In general, there is a large body of literature dedicated to identifying object relevance, such as by reachability analysis [144], [145] or learning to predict importance [49], which may identify more symbols to add to the initial set.

- The choice of symbols to add to the planning instance can be informed by feedback (*Check* in Algorithm 2) from failed solutions to sub-problems. We compute this feedback as follows. Every time a mid-level motion plan succeeds in finding a path through the places, we then attempt to solve for the low-level trajectory guided by this path. This low-level planner first solves for a path with only the objects which are known to the high-level planner. If such a path exists, we check if it intersects any objects in the scene that have not yet been added to the domain. If so, and we cannot find another path which does not intersect any objects, we save the offending object. Other approaches to identifying feedback might involve querying language models.

- Finally, at the end of each iteration of attempting to solve sub-problems, we return to the feedback objects that were saved. Each saved object, which was determined to be a potential cause of failure, is then added to the planning domain. Specifically, we add the same facts to the initial state that we would have added had we considered that object relevant initially. In this case, those are facts like `AtPose` and `Suspicious`. Now that the high-level planner is aware of these objects, it will suggest abstract plans which include them as parameters (e.g., inspecting the given object to neutralize it).

Finally, we must decide how many task plan skeletons will be checked by *Skeletons* and how much time will be spent attempting to solve the continuous subproblems before adding new symbols to the planning instance. In the Inspection domain, the full problem only has a solution if the pruned problem has a solution, so we choose to stop iterating through plan skeletons once we find a solution to the reduced problem. In general, a maximum time must be set for iterating through plan skeletons (we do so according to the *Adaptive* approach as defined in Garrett, Lozano-Pérez, and Kaelbling [36]). Below we present pseudo-code for the Incremental Object Solver approach.

Algorithm 2 Incremental Object Solver

```
Input: $A, S, \mathcal{O}, I, G$
Output: A valid plan $\pi$ or INFEASIBLE
$\mathcal{O}_S \leftarrow$ GetRelevantObjects($\mathcal{O}$) ;    // Objects relevant for non-collision reasons
$\mathcal{O}_I \leftarrow \mathcal{O}_S$
$\mathcal{O}_R \leftarrow \mathcal{O} \setminus \mathcal{O}_S$
while $|\mathcal{O}_I| < |\mathcal{O}|$ do
    $SkelInfo \leftarrow [\,]$
    foreach $k \in$ Skeletons($A, S, \mathcal{O}_I, G$) do
        $T \leftarrow$ SolveSubProblems($k$)
        $Feedback \leftarrow$ Check($T, \mathcal{O}_R$)
        $SkelInfo$.append($Feedback$)
    if $\pi \in$ SkelInfo is valid then
        return $\pi$
    $\mathcal{O}_{new} \leftarrow$ NewObj($SkelInfo$)
    $\mathcal{O}_I \leftarrow \mathcal{O}_I \cup \mathcal{O}_{new}$
    $\mathcal{O}_R \leftarrow \mathcal{O}_R \setminus \mathcal{O}_{new}$
return INFEASIBLE
```

5.3 Evaluation

There are three primary axes of complexity for planning problems in large scene graphs. One is the complexity of the goal specification, the second is correlated to the scale of the environment, and the third is related to the geometry of the scene (and potential obstructions therein). We aim to characterize which types of planning problems our planning approach is well suited for considering these sources of complexity. We compare our encoding of the Inspection domain to the dense, direct encoding in a variety of different settings. We conduct tests on four map archetypes – a synthetic small constrained alleyway, a synthetic 10x10 gridworld, a scene graph built from real data in an office environment comprising 557 Places and 28 Objects, and a much larger scene graph built from the KITTI dataset composed of 17861 Places and 1315 Objects (Figs. 5.10 and 5.8). For each environment, we test several different goal clauses across different variations in robot and object initial conditions. Finally, we also implement and test our planner on a Spot robot, which we use to build and plan in a Hydra scene graph in building 45 at MIT.

To randomize tasks across trials, we define a mechanism for sampling goal specifications according to an increasing number of clauses in Disjunctive Normal Form (DNF). A goal with complexity (N, K) is a formula in DNF with N clauses, where each clause has K conjunctions. For example, for complexity (2, 3), the goal has the form `(Or (`C_1`, `C_2`))`, where C_i is a clause consisting of three facts (e.g., `And ((Visited P1), (Safe O4), (Not (Visited P9))))`.

5.3.1 Scene Graph Size

First, we investigate the effect of scene graph size on our ability to plan. For this set of trials, the goal complexity is (N, K) = (3, 3), and we compare planning time for the direct encoding to the planning time for our planner across the *Alley*, *Hallway*, and *Office* environments. The results for this comparison is shown in Fig. 5.9-A. Each point on the scatter plot corresponds to a single trial and different colors correspond to different environment types. Any samples above the black line indicate that our planner outperforms the dense baseline.

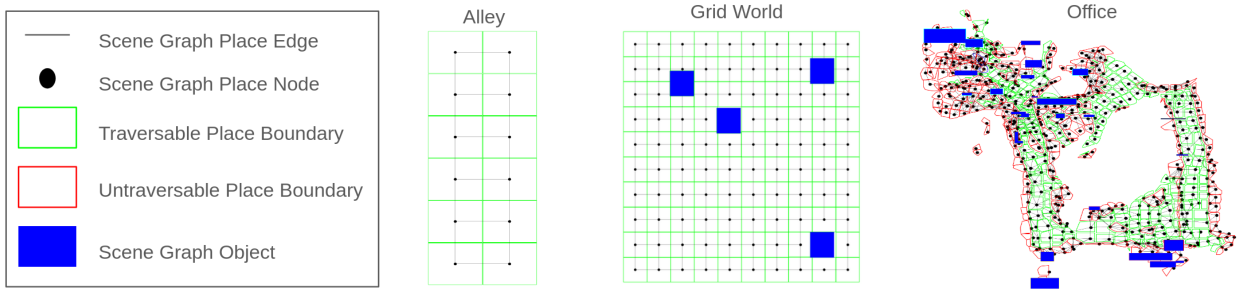


Figure 5.8: Three of the maps used for evaluation (not shown is the KITTI or building 45 environments). A narrow alley map, a simple 10x10 grid world map, and a scene graph built from real data collected by a robot in an office environment.

In the small Alley environment, our planner performs about as well as the dense encoding. This is expected, as there is not much advantage to sparsification in such a small environment. As we scale up environment size however, the relative performance of our planner improves. In the 10x10 grid, we see modest improvement as shown by the red points in Fig. 5.9-A. Scaling environment size even further, with the building-scale scene graph, we see the baseline planner taking hundreds of seconds to plan, whereas our planner averages in the tens of seconds. For the sake having a fair comparison, this experiment only considers goals that involve visiting or not visiting certain places in the scene graph. When we attempted to introduce object inspection, the dense baseline planner timed out before finding a plan in almost all instances. Similarly, when testing the baseline in the KITTI scene graphs, it was also unable to find solutions for goals of any complexity. Our proposed planner experienced only a modest increase in planning time as the size of the map scaled.

5.3.2 Goal Complexity

Next, we consider the effect of increasing goal complexity on planning time. To do this, we investigate a series of different goal constructions in the Grid World environment. Specifically, we run experiments with goal complexity (N, K), for K = 5 and K = 10, and N from 1 to 5. Goal facts are chosen to be either visiting or not visiting specific places. Fig. 5.9-B presents a plot comparing the complexity of the goal in terms of total unique symbol referenced vs planning time. For less complex goals in this environment, our planner outperforms the dense planner, up to a crossover point at around 20 unique objects. Advantages from

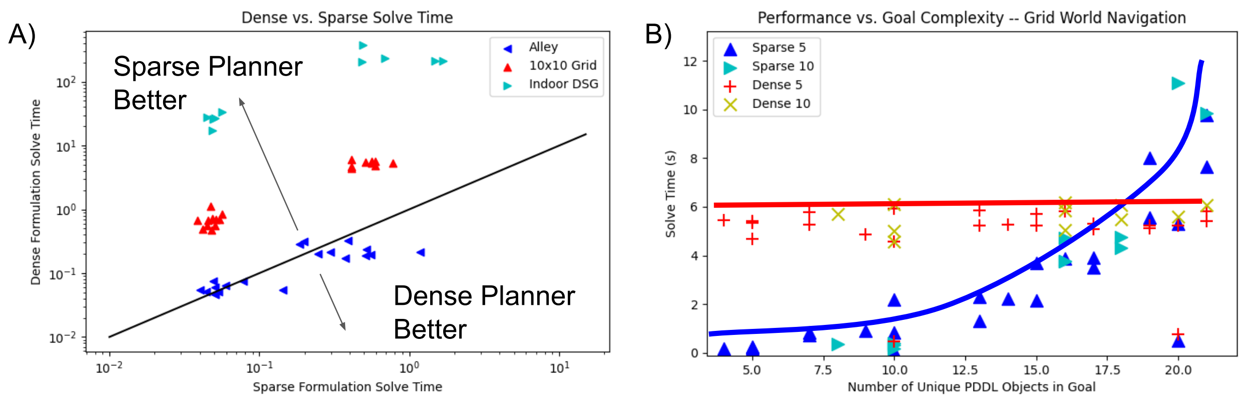


Figure 5.9: **A)** Comparison of the time to solve tasks of comparable complexity across different environmental scales for the dense formulation and the proposed sparse formulation. **B)** The scaling of our approach with the complexity of the goal specification in the simple 10x10 grid world. As we increase the number of unique PDDL objects in the goal specification, the problem is no longer sparse, and so it no longer benefits from our approach.

the additional structure in the dense formulation outweigh the gains of our sparser method when a large percentage of the place symbols are relevant to the goal. Once again, if we were to introduce obstructing objects, the dense direct encoding baseline is unable to solve goals of any meaningful complexity. Not included in either of these plots are results on the KITTI dataset, as the direct encoding never successfully completed a trial in this setting due to timing out.

5.3.3 Object Obstruction

We now investigate the performance of the incremental object adding algorithm in task instances where objects not directly listed in the goal must be inspected in order to solve the task. In other words, the geometry of the scene requires that the planner inspect unspecified objects before reaching its goal. In this experiment, we give a robot one of two goal types in a scene graph built from the KITTI dataset (Figs. 5.1 and 5.10): either `(Visited` P_i`)` or `(Safe` O_j`)`. Given the size of the map, satisfying these goals may require the agent to traverse a large distance. Perhaps more importantly, the environment shown in Fig. 5.10 contains inflated *obstacles*, which may be labeled as `suspicious`. If a low-level path passes through any `suspicious` objects on its way, that plan would be invalid, and the robot would be forced to inspect and neutralize them to find a safe path to the goal. As a baseline, we

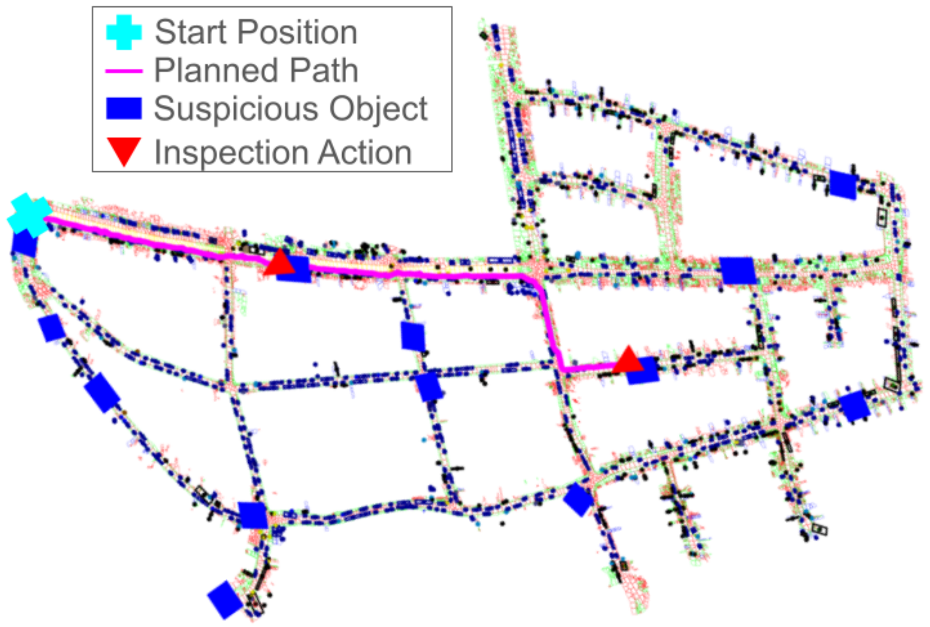


Figure 5.10: An example plan from the KITTI environment. The robot begins in the top left, and is tasked with inspecting one object (denoted by the red triangle at the end of the trajectory). Along the way, there are numerous objects potentially blocking the path, so we must add at least one to its planning domain. After inspecting and neutralizing this object, the robot can reach its goal.

sample 20 goals in the map shown in Fig. 5.10 using our planner without any of the objects being labeled as `suspicious`. In this case, they do not obstruct the agent's path, and we find plans in 19 of 20 trials.

Next, we "activate" 13 objects in the scene by labeling them as `suspicious`. A `suspicious` object has an inflated radius that is only safe for the robot to enter after it has been inspected and neutralized (in the KITTI scene, this radius is large enough to block an entire road as shown by the blue objects in Fig 5.10). For the agent to inspect the object, it has to find a pose that is traversable and within range of the object. Then, by taking the inspect action, the object becomes `safe`, and can be passed. To highlight the importance of object pruning, we attempt to solve these same tasks without using our incremental feedback approach for object pruning (Sec. 5.2.4). Instead, we add all 13 suspicious objects to the scene directly. Using this encoding, the planner only succeeds in finding a plan in 4 out of 20 trials. Inspecting these solutions further reveals that in all 4 of these successful cases, there was a direct path to the goal without inspecting any objects. This result makes sense, as the odds of sampling the correct object to inspect is low without the benefit of geometric information.

Finally, we test our proposed approach of incrementally adding objects to the planning instance (Sec. 5.2.4). Our planner solves 12 of the 20 trials, including 9 cases wherein the agent inspected one or more obstructing objects on the way to its goal. Failure to find plans resulted when PDDLStream was unable to find sequences of inspection poses on the correct side of obstructing objects when several such objects needs to be inspected to find a plan. These failures might be mitigated by integrating the approach proposed in Chapter 4, though that was not tested here. These experiments further demonstrate the importance of our proposed approach to sparsifying otherwise dense, long-horizon planning problems. An example plan, where the agent investigates two objects on the way to its goal is shown in Fig. 5.10.

5.3.4 Real-World Manipulation

Finally, we demonstrate our planner in a real-world setting (Fig. 5.7), using a Boston Dynamics Spot quadruped to build a 3D scene graph in real-time in a university building. In order to demonstrate that our approach is effective on domains different from “Inspect”, we implement a “Retrieval" domain, which adds additional `Pick` and `Place` actions to the Inspection domain, enabling the robot to move objects around the environment. The goal specifies which place an object should be in (e.g., `(ObjectAtPlace O1 P3)`, denoting a goal state where object `O1` is in place `P3`). For each trial, we structure the environment such that there is an obstruction preventing the robot from reaching its target object. To solve the task, the robot must move this obstruction out of the way, before retrieving the specified object. We encoded `pick`, `place`, `inspect`, and `move skills` for the robot. The full domain and stream files are presented in Appendix A.3.

Like the KITTI domain, we rely on our incremental object adding approach to only add the obstructing object to our high-level planner. Scattered throughout the environment are a number of different objects (Fig. 5.11), which would lead to an intractable problem if we were to consider them all. We ran ten trials, each time finding plans, and executing those plans on the robot. While the planner reliably finds feasible plans, execution often requires

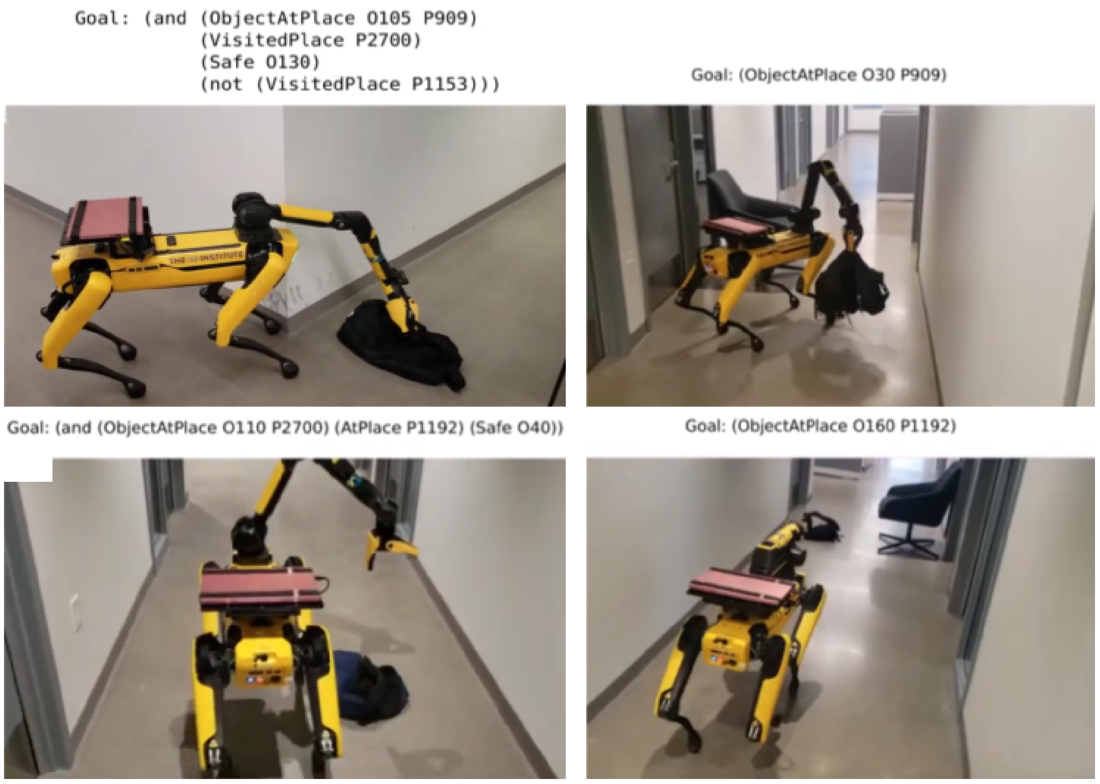

Goal: (and (ObjectAtPlace O105 P909)
(VisitedPlace P2700)
(Safe O130)
(not (VisitedPlace P1153)))
Goal: (ObjectAtPlace O30 P909)
Goal: (and (ObjectAtPlace O110 P2700) (AtPlace P1192) (Safe O40))
Goal: (ObjectAtPlace O160 P1192)

Figure 5.11: Here we highlight a selection of four real work experiments executed on the Spot robot in building 45 at MIT. The complexity of the trial was in part given in the goal specification (above each image), and in part by the environmental setup. In many cases, objects not specified in the goal had to be identified by the planner as obstructions and moved out of the way before Spot could accomplish its task. See Fig. 5.7 for the environmental layout, as well as the final trajectory for the plan in the top left.

several attempts due to failures when executing the `Pick` skill and Spot's local planner failing in certain constrained passages.

5.4 Discussion

In this chapter, we proposed an approach for enabling and accelerating TAMP in large environments. Until this point in the thesis, we had not yet addressed how our robots build planning abstractions from real perception, and so here we defined a method for constructing planning domains from real world Hydra scene graphs. The representations that naturally derive from this approach fit the requirements for hierarchical planning well. 3D scene graphs like Hydra are themselves hierarchical, separating higher level abstract elements like objects or rooms from lower level geometric information. This provides a natural decomposition for TAMP approaches, which initially consider the continuous and discrete elements of a plan separately. However, while the scene graph contains all the information we might need to solve a plan grounded in real perception, it also includes plenty of details superfluous to an efficient, accurate planner. There are many elements in a scene which may be irrelevant for a given goal specification, and so only serve to distract a planner in its search for a concrete plan.

The first contribution of this chapter is a method for composing a TAMP planning representation from a Hydra scene graph, and the second is our tri-level planner, which greatly reduces the planning horizon within that representation (Sec. 5.2.1). Our planner is organized as follows. At the top level, we consider interactions with objects and other higher-level abstract concepts, using the *Adaptive* approach in Garrett, Lozano-Pérez, and Kaelbling [36] to identify navigation sub-problems that require solutions to ground any abstract plans. The mid-level planner then uses the *Places* layer of the scene graph to accelerate the search for paths through space, while the low-level planner grounds these to continuous trajectories. If (as is assumed in many previous approaches to planning through 3D scene graphs) we could rely on the idea that all abstract plans could be refined to concrete trajectories, this planner would be sufficient. However, we know this is not the case, and given that Hydra scene graphs can be very rich in information, our planning abstraction will be bogged down by a tyranny of choice as demonstrated in Fig. 5.4.

To reduce the breadth of possible options for our high-level planner, we proposed a method for identifying provably redundant elements of a scene graph to remove from its planning instance. We also proved how the plans we produce from this reduced scene graph are valid and conform to the constraints of the full planning domain. These proofs hinge on the fact that while we remove certain elements from our high-level planner's domain, they remain usable for the mid and low-level planners, ensuring we do not limit the types of trajectories our planner can find. We identified that certain other high-level objects (which are not certain to be redundant), are also potentially too distracting to naively leave in our high-level planning domain. We further proposed a method for removing these elements from the planning domain initially, then adding them back upon receiving feedback from the mid and low-level planners that they were a cause of failure for an abstract plan. This incremental object-adding approach allows us to reduce planning complexity, while still guaranteeing a plan will eventually be found if a pruned symbol does turn out to be relevant to the planning problem.

Finally, we demonstrated our approach experimentally across a few different axes. Specifically, we conducted experiments that highlighted how our planner performs as we scale scene graph size, goal complexity, and geometric constraints in several environments. We tested our method in two simulated scene graphs, as well as three build from real perception, including a scene graph built from the KITTI dataset and real-world execution on a Spot quadraped. The real world experiments in particular highlight how our method is robust enough to produce executable plans in noisy, real world scene graphs. However, these experiments do reveal some limitations to our approach, as well as areas for potential improvement, which we will discuss now.

- **The Importance of Parameter Selection:** One thing that is (perhaps surprisingly) critical in TAMP solvers is the selection and tuning of certain search parameters. Particularly as it relates to the *Adaptive* algorithm [36], parameters such as the search/sample ratio (which determines how much time is proportionally spent finding high level plan skeletons vs. time spent attempting to solve the associated sub-problems to ground them) greatly impact the efficiency of the planner. For instance, we can set the maximum for the number of times the planner will attempt to solve a particular sub-problem. If we choose to only sample 3 different placement poses for example, this greatly reduces the number of symbols the high level planner can consider, affecting the problem's branching factor. However this of course may prevent us from finding solutions in particularly cluttered environments. These decisions, while impactful, approximately affect our planner and the baseline we compare to equally, meaning the experimental improvement we saw is still meaningful, though the actual planning times might have significant room for improvement.

- **Planning Inefficiencies:** As mentioned above, we rely on the *Adaptive* method as presented by Garrett, Lozano-Pérez, and Kaelbling [36] as our TAMP planner. This planner is integrated within our overall tri-level planning approach, as well as the incremental-object-adding algorithms. However, there is a compelling argument to be made that the *Adaptive* method is ill-suited for the types of problems we address in this work. In particular, the explicit separation between the low-level geometry and high-level discrete decision making is why our feedback method is necessary in the first place. Consider instead the approach proposed by Kaelbling and Lozano-Pérez [44] in Hierarchical Task and Motion Planning in the Now (HPN). In this method, if a high-level plan fails, the next plan that is tried is one which specifically attempts to address the source of failure. For example, if a motion plan fails, the next plan will be one which attempts to move the obstruction out of the way. This behavior is what we are trying to encourage with our incremental object-adding approach. The HPN approach is strictly less general (there are implicit assumptions that local information is all that might be relevant for failure), but there is an opportunity for our feedback approach to "fix" this. We might consider replacing our use of *Adaptive* with an HPN planner (using the same encoding and tri-level-planning structure).

- **Execution Failures:** During our real world trials on the Spot robot, extensive tuning of our hand written *skills* was required before execution of any found plans was possible. This tuning involved properties such as an imaging stand-off distance to ensure the semantic segmentation could detect graspable objects in the camera frame, as well as different parameters in that grasp skill such as: 1) how far from the object the robot should stand before initiating, 2) how forcefully the gripper should be closed, 3) where the object should be held in relation to the robot's body, and 4) how the object should be released when placement occurs. This sort of fine-tuning is to be expected; believing that hand-written skills will work out-of-the-box in the real world would be naive at best.

  However, a major reason adaptive skills are required in the first place is due to the fact that we executed our plans in an otherwise open-loop manor. The planner returns a grasp position based on the scene graph presented to it, potentially for objects on the other side of a building, and the robot navigates to that point before attempting the skill. This approach requires a very accurate estimate of robot odometry (which the Spot robot does have), though even then we are required to have the robot potentially look all around it for the object it is meant to grab in case it is not immediately in its hand camera frame.

In theory, if execution of our plans were perfect, and if the scene graphs we planned in were noiseless, there would be no need to adapt the trajectories our planner produced. Of course, we know this is never the case on a real robot. This brittle modeling assumption raise an interesting question however. If we are correcting for slight errors in position and relying on a low-level controller to solve for grasps and trajectories in real-time during execution, should we be spending computation at planning time to produce these trajectories in the first place? On the one hand, we want to confirm that a trajectory exists before beginning execution, however on the other hand, perhaps we do not need to be as precise with this initial solution, considering it is likely to be overwritten. Learning when it is necessary to confirm the feasibility of sub-problems is one approach which might further reduce the computational burden of our approach.

These factors, while limiting, are not insurmountable, and in fact present interesting opportunities for future work. In the next chapter, we will consider a few different possible future directions, not only related to the work in this chapter, but also concerning the research presented in Chapters 3 and 4.

Chapter 6

Conclusion

The goal of this thesis is to make progress toward the aim of building robots which are capable of acting intelligently within a complex world. Enabling this kind of behavior on real world robots requires that an agent make decisions hierarchically within a representation built from real world sensor input. Unfortunately, the types of hierarchical abstractions in robotics problems are unavoidably imperfect. One way this manifests is that solutions at one level of a hierarchy do not always transfer to lower levels. This fact introduces challenges to planning, leading to backtracking and wasted computational effort. Over the previous few chapters, we have shown how we can improve hierarchical planning efficiency by explicitly reasoning about the imperfections inherent to the abstractions available to robots. Specifically, we have highlighted three domains commonly faced by robots in the real world, which we briefly summarize below.

In Chapter 3 we addressed the problem of planning in the presence of uncertainty, specifically considering complex goals with temporal constraints, wherein a robot must solve its task in a previously unexplored environment. We defined a hierarchical representation with high-level actions derived from the environment and the given task itself, forming an abstraction which reduces the horizon of the search problem, but which we know does not satisfy the property of downward refinement. To address this challenge, we proposed an approach which learns from visual input to predict the feasibility and cost of executing a given action. We then used these predictions to guide a stochastic search algorithm, biasing exploration to regions which are more likely to lead to a low-cost, feasible solution. Notably, our learned model is structured to generalize across environments and task specifications without requiring retraining, and is trained with relatively few training examples. Over several different environments, varying in scale from a handful of rooms, up to the size of a building on MIT's campus, we demonstrated improvement in total cost in both simulated and real-world experiments compared to a heuristic-driven baseline.

While the contributions in Chapter 3 were useful in sequential navigation problems, we made some assumptions which make planning in higher-dimensional problems difficult. Notably, many robotics tasks (particularly in the domain of manipulation), require carefully thinking about planning through known space in order to interact with the world in cluttered environments. In Chapter 4, we built upon the work in Chapter 3 to address the challenges of TAMP. Specifically, instead of learning the success or failure of executing actions in a top-down manner, we trained models to predict the feasibility and cost of solving sub-problems in an abstract plan. We used these predictions within a novel approach for search, allowing us to once again reason about the imperfections in our TAMP abstraction. We were able to demonstrate improvement across a variety of simulated and real world tasks when compared to state of the art TAMP solvers.

Finally, we next addressed the challenges presented by environmental scale in TAMP. In Chapter 5, we proposed an approach to extract planning domains from hierarchical scene graphs, producing a representation which is overloaded with objects and symbols irrelevant to the overall task. To alleviate the burden on our planner, we propose a method for pruning our planning domain of superfluous elements, provably identifying symbols which can be safely ignored. We also proposed removing objects which may not be redundant depending on scene geometry, however we account for this possibility by explicitly reasoning about the the possibility that our new abstraction is imperfect after they are removed. If an object is determined (via planner feedback) to be needed to solve a task, we add it back to our domain, and continue planning. We showed our approach to be an improvement over existing approaches, demonstrating successful planning across a variety of environments at tasks. Notably, we demonstrated mobile manipulation executed on a Spot robot, executed at whole building scale.

The contributions made in these areas are a step toward efficient robotic decision making, however there are still many opportunities to make progress in this area. The rest of this chapter is a discussion of potential future (and some ongoing) research, and how we can build off the contributions presented in this thesis.

6.1 Incorporating Foundation Models into Planning

In Chapters 3 and 4, we presented planning approaches which relied upon learning to make predictions about the feasibility and cost of refining high-level plans. The problem settings were different, and so how these predictions were used within a planner varied as well, however one core principle was consistent between approaches. Regardless of the task, we were careful to design systems which did not require large amounts of training data in order to learn the signals we needed to plan.

The motivation behind this is clear, and we discussed it somewhat in Chapter 1. Finding useful data in robotics can be challenging depending on the setting, though we can collect some valuable data in simulation of course for certain tasks. In particular, recent work in Reinforcement Learning (RL) has taken advantage of improved simulation to produce short-horizon controllers which translate well to the real world. However, regardless of the task, some real world experience is needed to fine-tune what we have learned in simulation to match reality. For example, in Chapter 3, our agent was able to reuse its learned models for different environmental orientations in both simulation and the real world. However, if we were to have moved our robot outdoors instead of within an MIT academic building, its models would have been useless. Requiring data across every class of environment is expensive, and is not always possible.

In the past few years, an increasing subset of the research in robotics has been centered around the development of foundation models [146]. These networks are of significant scale—on the order of hundreds of billions of parameters—and require massive amounts of data. Large Language Models (LLMs) for example are trained on large corpora of textual information, and are able to be incorporated into robotics planning stacks with some compelling results [147], [148]. Interestingly, while the data requirements to train these models are massive, little to no real robot data may actually be required, meaning they may be a useful tool when robotic data is difficult to acquire.

The early results in this space have primarily been focused on solving problems in a top-down manner, and have been tested on settings where most actions are feasible. There have been some efforts toward incorporating foundation models into TAMP frameworks [149], though less so in the domain of mobile manipulation. One approach which attempts to solve a similar problem to the one we address in Chapter 3 is presented in Shah, Equi, Osiński, *et al.* [150]. There, the authors navigate partially revealed environments by querying a language model for its best guess of where it should explore given a set of frontiers. Notably however, the time to find a solution in this approach is very slow (on the order of 30 minutes) for home sized environments.

Missing in these approaches is the insight we have gained in this thesis, that we can improve planning efficiency by explicitly reasoning about the imperfections in hierarchical planning abstractions. Taking advantage of the generalization of foundation models, and incorporating them within the frameworks presented here, is a promising potential direction for future research.

6.2 Ongoing Work Augmenting Hierarchical Planning with Language

In the last section we discussed the possibility of guiding planning within our algorithms with predictions informed in part by large-scale foundation models. Some steps have already been taken to this end, though in a somewhat orthogonal direction, building upon the work in Chapter 5. One downside of our approach in Chapter 5 is its complexity of use. In order to use our planner, a human operator inspects a Hydra scene graph visually, and writes a PDDL style goal specification which captures their intent. Consider the scene graph highlighted in Fig. 6.1. If, for example, we wanted a robot to 1) inspect a particular object, 2) move a different object to a chosen location, 3) go to a specified place, all while 4) avoiding a different place, the appropriate encoding is as follows: `(and (ObjectAtPlace O0 P0) (VisitedPlace P1) (Safe O1) (not (VisitedPlace P2)))`, where the assigned

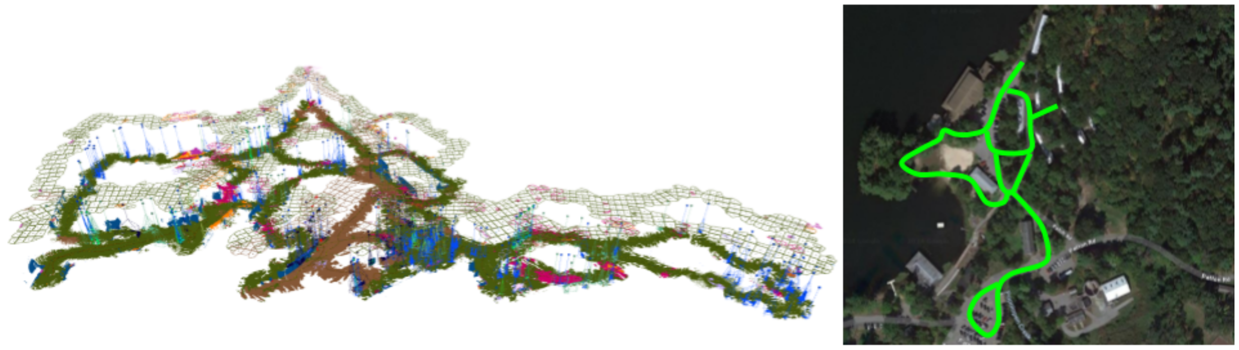


Figure 6.1: Hydra scene graph built in large scale outdoor environment. On the right, we see a top down satellite view of the same area, with the traversed regions highlighted in green. This map was produced by multiple agents fusing their sensor measurements to produce one large graph, and so we would like to be able to generate multi-agent plans within it.

numbers must match the true values in the graph. Identifying this specification requires an intimate knowledge of the system. In particular, the operator must pick out the specific symbol in the scene (of which there may be thousands), associate it with the proper predicate which can be interpreted by the planner, and format them in a way which conveys the user's true intent. Moreover, if the operator wants to specify relational goals such as *inspect the nearest door*, or allow for ambiguity *bring me any bag*, they must refine that desire into a PDDL encoding themselves.

We can build upon approaches for interpreting hierarchical scene graphs with natural language to address these challenges. The semantic information in a Hydra scene graph, along with the structure encoded in edges between different symbols in the graph, can be used as input to the context of a Large Language Model. If we augment that context with the planning domain of our robot (defined in PDDL), then the model has access to a representation of both its environment, and what actions it can take within that environment. This allows us to design a system where a human operator can give a command in natural language, and the language model has the information needed to produce a goal specified in PDDL that the planner can use to produce an executable plan.

In Figure 6.1, we see a Hydra scene graph built from data collected at Camp Buckner, West Point. The scale of this environment is nearly on the order of a kilometer, and the graph was build with data fused from multiple different robots. The planner we discussed in Chapter 5 is not designed for multi-robot planning, nor is it exposed to higher levels of the abstraction like regions (e.g., which places exist in roads, forest, etc.). However, with our LLM encoding, we can query a model (e.g. GPT4-o) to produce PDDL task specifications for multiple robots for a task dictated in natural language.

Consider the command: *“The bag in the shelter and the trash are both suspicious. I need you two to secure the area. Then rendezvous at the tree in the road.”* Given this task, our system is able to produce goals in PDDL, specific to each robot, which allow to them to plan to solve and execute this task. Specifically, this command is translated to `(and (Safe O54) (Safe O85) (AtPlace P1034))` for the Spot robot, and `(AtPlace P1034)` for a Husky. We are able to extend the capabilities of our planner with this approach, where we essentially have a foundation model sitting at the highest level of our hierarchical planning system. The potential for such a system is exciting, and is a compelling area of future research.

Appendix A

PDDL Domain Encodings

A.1 Unpack Domain

A.1.1 Unpack Domain File

```
(define (domain unpack)
 (:requirements :strips :equality :action-costs)
  (:types obj grasp_dir grasp config pose trajectory)
  (:predicates

    (fixed ?r - obj)
    (graspable ?o - obj)

    (IsGraspDir ?o - obj ?p - pose ?dg - grasp_dir)
    (isgrasp ?o - obj ?g - grasp)
    (GraspAtPose ?g - grasp ?p - pose)

    (ispose ?o - obj ?p - pose)
    (isconf ?q - config)
    (istraj ?t - trajectory)
    (iskin ?o - obj ?p - pose ?g - grasp ?q - config ?t - trajectory)
    (isfreemotion ?q1 - config ?t - trajectory ?q2 - config)
    (isholdingmotion ?q1 - config ?t - trajectory ?q2 - config ?o -
       obj ?g - grasp)
    (trajcollision ?t - trajectory ?o2 - obj ?p2 - pose)

    (issupport ?o - obj ?p - pose ?r - obj)

    (atpose ?o - obj ?p - pose)
    (atgrasp ?o - obj ?g - grasp)
    (handempty)
    (atconf ?q - config)
    (canmove)

    (on ?o - obj ?r - obj)
    (holding ?o - obj)

    (usedgrasp ?g)
    (canpick)
    (canplace)
  )

  (:functions
      (total-cost) - number
  )

  (:action move_free
    :parameters (?q1 - config ?q2 - config ?t - trajectory)
    :precondition (and (isfreemotion ?q1 ?t ?q2)
                       (atconf ?q1) (handempty) (canmove))
    :effect (and (atconf ?q2)
                 (not (atconf ?q1))
                 (not (canmove))
                 (canpick)
```

```
                (increase (total-cost) 100)
                )
)
(:action move_holding
  :parameters (?q1 - config ?q2 - config ?o - obj ?g - grasp ?t -
     trajectory)
  :precondition (and (isholdingmotion ?q1 ?t ?q2 ?o ?g)
                      (atconf ?q1) (atgrasp ?o ?g) (canmove))
  :effect (and (atconf ?q2)
                (not (atconf ?q1))
                (not (canmove))
                (canpick)
                (canplace)
                (increase (total-cost) 100)
                )
)
(:action pick
  :parameters (?o - obj ?p - pose ?g - grasp ?q - config ?t -
     trajectory)
  :precondition (and (iskin ?o ?p ?g ?q ?t)
                      (atpose ?o ?p) (handempty) (atconf ?q)
                      (canpick)
                      (not (usedGrasp ?o ?p ?g))
                      (GraspAtPose ?g ?p)
                      )
  :effect (and (atgrasp ?o ?g)
                (canmove)
                (not (atpose ?o ?p))
                (not (handempty))
                (increase (total-cost) 100)
                )
)
```

```
  (:action place
    :parameters (?o - obj ?p - pose ?r - obj ?g - grasp ?q - config ?
       t - trajectory)
    :precondition (and (iskin ?o ?p ?g ?q ?t)
                       (issupport ?o ?p ?r)
                       (atgrasp ?o ?g)
                       (atconf ?q)
                       (graspable ?o)
                       (fixed ?r)
                       (canplace)
                       )
    :effect (and (atpose ?o ?p)
                 (handempty)
                 (canmove)
                 (not (atgrasp ?o ?g))
                 (not (canpick))
                 (not (canplace))
                 (increase (total-cost) 100)
                 )
  )

  (:derived (holding ?o - obj)
    (exists (?g) (and (isgrasp ?o ?g)
                        (atgrasp ?o ?g)))
  )
)
```

A.1.2 Unpack Stream File

```
(define (stream unpack)
  (:stream sample-place
    :inputs (?o - obj ?r - obj)
    :domain (Stackable ?o ?r)
    :outputs (?p - pose)
    :certified (and (IsPose ?o ?p) (IsSupport ?o ?p ?r))
  )
  (:stream sample-grasp
    :inputs (?o - obj ?p - pose)
    :domain (and (Graspable ?o) (isPose ?o ?p))
    :outputs (?g - grasp)
    :certified (and (GraspAtPose ?g ?p) (IsGrasp ?o ?g))
  )
  (:stream inverse-kinematics
    :inputs (?o - obj ?p - pose ?g - grasp)
    :domain (and (IsPose ?o ?p) (IsGrasp ?o ?g))
    :fluents (AtPose)
    :outputs (?q - config ?t - trajectory)
    :certified (and (IsConf ?q) (IsTraj ?t) (IsKin ?o ?p ?g ?q ?t))
  )
  (:stream plan-free-motion
    :inputs (?q1 - config ?q2 - config)
    :domain (and (IsConf ?q1) (IsConf ?q2))
    :fluents (AtPose)
    :outputs (?t - trajectory)
    :certified (IsFreeMotion ?q1 ?t ?q2)
  )
  (:stream plan-holding-motion
    :inputs (?q1 - config ?q2 - config ?o - obj ?g - grasp)
    :domain (and (IsConf ?q1) (IsConf ?q2) (IsGrasp ?o ?g))
    :fluents (AtPose)
    :outputs (?t - trajectory)
    :certified (IsHoldingMotion ?q1 ?t ?q2 ?o ?g)
  )
)
```

A.1.3 Kitchen Domain File

A.2 Kitchen Domain

```
(define (domain kitchen)
  (:requirements :strips :equality :action-costs :typing)
  (:types arm obj grasp config pose trajectory)
  (:predicates
    (IsArm ?a - arm)
    (Controllable ?a - arm)
    (Stackable ?o - obj ?r - obj)
    (Sink ?r - obj)
    (Stove ?r - obj)

    (IsPose ?o - obj ?p - pose)
    (IsGrasp ?o - obj ?g - grasp)
    (Kin ?a - arm ?o - obj ?p - pose ?g - grasp ?q - config ?t -
       trajectory)
    (BaseMotion ?q1 - config ?t - trajectory ?q2 - config)
    (Supported ?o - obj ?p - pose ?r - obj)

    (AtPose ?o - obj ?p - pose)
    (AtGrasp ?a - arm ?o - obj ?g - grasp)
    (Graspable ?o - obj)
    (HandEmpty ?a - arm)
    (AtBConf ?q - config)
    (IsBConf ?q - config)

    (CanMove)
    (Cleaned ?o - obj)
    (Cooked ?o - obj)
    (CanOperate)

    (On ?o - obj ?r - obj)
    (Holding ?a - arm ?o - obj)
  )

  (:functions
      (total-cost) - number
  )

  (:action move_base
    :parameters (?q1 - config ?q2 - config ?t - trajectory)
    :precondition (and (BaseMotion ?q1 ?t ?q2)
                       (AtBConf ?q1) (CanMove)
                       )
    :effect (and (AtBConf ?q2)
                 (not (AtBConf ?q1)) (not (CanMove))
                 (increase (total-cost) 200)
                 (not (CanOperate))
                 )
  )
  (:action pick
    :parameters (?a - arm ?o - obj ?p - pose ?g - grasp ?q - config ?
       t - trajectory)
    :precondition (and (Kin ?a ?o ?p ?g ?q ?t)
                       (AtPose ?o ?p) (HandEmpty ?a) (AtBConf ?q)
                       )
    :effect (and (AtGrasp ?a ?o ?g) (CanMove)
                 (not (AtPose ?o ?p)) (not (HandEmpty ?a))
                 (increase (total-cost) 100)
                 (not (CanOperate))
                 )
```

```
)
(:action place
  :parameters (?a - arm ?o - obj ?p - pose ?g - grasp ?q - config ?
     t - trajectory)
  :precondition (and (Kin ?a ?o ?p ?g ?q ?t)
                     (AtGrasp ?a ?o ?g) (AtBConf ?q)
                     )
  :effect (and (AtPose ?o ?p) (HandEmpty ?a) (CanMove)
               (not (AtGrasp ?a ?o ?g))
               (increase (total-cost) 100)
               (CanOperate)
               )
)
(:action clean
  :parameters (?o - obj ?r - obj)
  :precondition (and (Stackable ?o ?r) (Sink ?r)
                     (On ?o ?r)
                     (CanOperate)
                     )
  :effect (and (Cleaned ?o)
               (increase (total-cost) 50)
               (not (CanOperate))
               )
)
(:action cook
  :parameters (?o - obj ?r - obj)
  :precondition (and (Stackable ?o ?r) (Stove ?r)
                     (On ?o ?r) (Cleaned ?o)
                     (CanOperate)
                     )
  :effect (and (Cooked ?o)
               (not (Cleaned ?o))
```

```
                (increase (total-cost) 50)
                (not (CanOperate))
                )
  )

  (:derived (On ?o - obj ?r - obj)
    (exists (?p) (and (Supported ?o ?p ?r)
                     (AtPose ?o ?p))
                     )
  )
  (:derived (Holding ?a - arm ?o - obj)
    (exists (?g) (and (IsArm ?a) (IsGrasp ?o ?g)
                     (AtGrasp ?a ?o ?g))
                     )
  )
)
```

A.2.1 Kitchen Stream File

```
(define (stream kitchen)
  (:stream sample-place
    :inputs (?o - obj ?r - obj)
    :domain (Stackable ?o ?r)
    :outputs (?p - pose)
    :certified (and (IsPose ?o ?p) (Supported ?o ?p ?r))
  )
  (:stream sample-grasp
    :inputs (?o - obj)
    :domain (Graspable ?o)
    :outputs (?g - grasp)
    :certified (IsGrasp ?o ?g)
  )
  (:stream inverse-kinematics
    :inputs (?a - arm ?o - obj ?p - pose ?g - grasp)
    :domain (and (Controllable ?a) (IsPose ?o ?p) (IsGrasp ?o ?g))
    :fluents (AtPose)
    :outputs (?q - config ?t - trajectory)
    :certified (and (IsBConf ?q) (Kin ?a ?o ?p ?g ?q ?t))
  )
  (:stream plan-base-motion
    :inputs (?q1 - config ?q2 - config)
    :domain (and (IsBConf ?q1) (IsBConf ?q2))
    :fluents (AtPose)
    :outputs (?t - trajectory)
    :certified (BaseMotion ?q1 ?t ?q2)
  )
)
```

A.3 Relaxed Encoding of Inspection Domain

A.3.1 Relaxed Domain File

```
(define (domain manipulation)
  (:requirements :strips :equality)
  (:predicates

    (Place ?p)
    (CTraj ?ct)
    (CTrajPl ?ct)
    (Object ?o)
    (Pose2d ?p)
    (AtPose ?p)
    (AtPlace ?p)
    (AtObject ?o)
    (VisitedPose ?p)
    (VisitedPlace ?p)
    (VisitedObject ?o)

    (ObjectPose ?o ?p)
    (ObjectAtPose ?o ?p)
    (ObjectAtPlace ?o ?p)
    (ObjectPlacementPose ?o ?p0 ?pr)
    (LocalPlacementPose ?o ?p0 ?pr)
    (ObjectPoseInPlace ?o ?po ?pl)

    (HandFull)

    (Suspicious ?o)
    (Safe ?o)
    (Holding ?o)
    (Moved ?o)
    (CannotMove)

    (PutativePlaceMotion ?p1 ?p2 ?tr)
    (ConfigurationMotion ?c1 ?c2 ?ct)
    (ConfigurationMotionPassLast ?c1 ?c2 ?ct)
    (ObjectInspectionPose ?o ?po ?pr)
    (ObjectGraspPose ?o ?po ?pr)
    (PoseInPlace ?o ?p)
    (PutativePoseInPlace ?o ?p)
    (NeedPlaceCertificate ?pose)
    (NearbyObjectPose ?o ?p)

)
```

```
(:functions
  (MoveCost ?t)
  (MovePlaceCost ?t)
)

(:derived (AtPlace ?p)
  (exists (?pose) (and (AtPose ?pose) (PoseInPlace ?pose ?p))))

(:derived (ObjectAtPlace ?x ?p)
  (exists (?pose) (and (ObjectAtPose ?x ?pose) (ObjectPoseInPlace ?
     x ?pose ?p))))

(:derived (VisitedPlace ?p)
  (exists (?pose) (and (VisitedPose ?pose) (PoseInPlace ?pose ?p)))
     )

(:derived (AtObject ?o)
  (exists (?p) (and (AtPose ?p) (NearbyObjectPose ?o ?p))))

(:derived (VisitedObject ?o)
  (exists (?p) (and (VisitedPose ?p) (NearbyObjectPose ?o ?p))))

(:action inspect
  :parameters (?o ?pr ?po)
  :precondition (and (AtPose ?pr)
                     (ObjectPose ?o ?po)
                     (ObjectAtPose ?o ?po)
                     (ObjectInspectionPose ?o ?po ?pr)
                     (not (HandFull))
                     (Suspicious ?o))
  :effect (and (not (Suspicious ?o))
               (not (CannotMove))
               (Safe ?o)
               (increase (total-cost) 5))
)

(:action pick
  :parameters (?o ?pr ?po)
  :precondition (and (AtPose ?pr)
                     (ObjectPose ?o ?po)
                     (ObjectAtPose ?o ?po)
                     (ObjectGraspPose ?o ?po ?pr)
                     (not (HandFull))
                     (not (Holding ?o)))
  :effect (and (not (ObjectAtPose ?o ?po))
               (Holding ?o)
               (not (CannotMove))
               (HandFull))
)

(:action place
  :parameters (?o ?pr ?po)
  :precondition (and (AtPose ?pr)
                     (Holding ?o)
                     (HandFull)
                     (ObjectPose ?o ?po)
                     (ObjectPlacementPose ?o ?po ?pr))
  :effect (and
               (not (Holding ?o))
               (not (HandFull))
               (Moved ?o)
               (ObjectAtPose ?o ?po)
               (not (CannotMove))
               (increase (total-cost) 1))
```

```
)

(:action place_nearby
  :parameters (?o ?pr ?po)
  :precondition (and (AtPose ?pr)
                     (Holding ?o)
                     (HandFull)
                     (ObjectPose ?o ?po)
                     (LocalPlacementPose ?o ?po ?pr))
  :effect (and
               (not (Holding ?o))
               (Moved ?o)
               (not (CannotMove))
               (not (HandFull))
               (ObjectAtPose ?o ?po))
)

(:action movePlace
  :parameters (?p1 ?p2 ?c1 ?c2 ?t ?tr)
  :precondition (and (PoseInPlace ?c1 ?p1)
                     (PoseInPlace ?c2 ?p2)
                     (PutativePlaceMotion ?p1 ?p2 ?tr)
                     (ConfigurationMotionPassLast ?c1 ?c2 ?t)
                     (AtPose ?c1)
                     (not (= ?c1 ?c2)))
  :effect (and (AtPose ?c2)
               (not (AtPose ?c1))
               (VisitedPose ?c2)
               (increase (total-cost) (MovePlaceCost ?c1 ?c2)))
)

(:action move
    :parameters (?c1 ?c2 ?t)
    :precondition (and (ConfigurationMotion ?c1 ?c2 ?t)
                       (or (NeedPlaceCertificate ?c1) (
                          NeedPlaceCertificate ?c2))
                       (AtPose ?c1)
                       (not (CannotMove))
                       (not (= ?c1 ?c2)))
    :effect (and (AtPose ?c2)
                 (not (AtPose ?c1))
                 (CannotMove)
                 (VisitedPose ?c2)
                 (increase (total-cost) (MoveCost ?c1 ?c2)))
  )

  (:action VerifyPoseInPlace
    :parameters (?pose ?place)
    :precondition (PutativePoseInPlace ?pose ?place)
    :effect (PoseInPlace ?pose ?place))
)
```

A.3.2 Relaxed Stream File

```
(define (stream manipulation)

  (:stream plan-motion-passlast
    :inputs (?c1 ?c2)
    :domain (and (Pose2d ?c1) (Pose2d ?c2))
    :fluents (Suspicious Safe ObjectAtPose Holding)
    :outputs (?tr)
    :certified (and (CTrajPl ?tr) (ConfigurationMotionPassLast ?c1 ?
      c2 ?tr)))
```

```
(:stream plan-motion
  :inputs (?c1 ?c2)
  :domain (and (Pose2d ?c1) (Pose2d ?c2))
  :fluents (Suspicious Safe ObjectAtPose Holding)
  :outputs (?tr)
  :certified (and (CTraj ?tr) (ConfigurationMotion ?c1 ?c2 ?tr)))

(:stream plan-place-motion
  :inputs (?p1 ?p2)
  :domain (and (Place ?p1) (Place ?p2))
  :outputs (?tr)
  :certified (and (PutativePlaceMotion ?p1 ?p2 ?tr)))

(:stream sample-inspect-pose
  :inputs (?o ?po)
  :domain (and (Object ?o) (ObjectPose ?o ?po))
  :outputs (?pose)
  :certified (and (NeedPlaceCertificate ?pose) (Pose2d ?pose) (
     ObjectInspectionPose ?o ?po ?pose) (NearbyObjectPose ?o ?pose)
     ))

(:stream sample-grasp-pose
  :inputs (?o ?po)
  :domain (and (Object ?o) (ObjectPose ?o ?po))
  :outputs (?pose)
  :certified (and (NeedPlaceCertificate ?pose) (Pose2d ?pose) (
     ObjectGraspPose ?o ?po ?pose)))

(:stream sample-placement-pose
  :inputs (?o)
  :domain (and (Object ?o))
  :outputs (?po ?pr)
```

```
    :certified (and (NeedPlaceCertificate ?pr) (Pose2d ?pr) (
       ObjectPose ?o ?po) (LocalPlacementPose ?o ?po ?pr)))

  (:stream sample-placement-pose-in-place
    :inputs (?o ?p)
    :domain (and (Object ?o) (Place ?p))
    :outputs (?po ?pr)
    :certified (and (NeedPlaceCertificate ?pr) (Pose2d ?pr) (
       ObjectPose ?o ?po) (ObjectPoseInPlace ?o ?po ?p) (
       ObjectPlacementPose ?o ?po ?pr)))

  (:stream sample-pose-in-place
    :inputs (?p)
    :domain (Place ?p)
    :outputs (?pose)
    :certified (and (Pose2d ?pose) (PoseInPlace ?pose ?p)))

  (:stream certify-pose-in-place
    :inputs (?pose ?place)
    :domain (and (NeedPlaceCertificate ?pose) (Pose2d ?pose) (Place ?
       place))
    :outputs ()
    :certified (PutativePoseInPlace ?pose ?place))

  (:function (MoveCost ?c1 ?c2)
    (and (Pose2d ?c1) (Pose2d ?c2))
  )

  (:function (MovePlaceCost ?c1 ?c2)
    (and (Pose2d ?c1) (Pose2d ?c2))
  )
)
```

Appendix B

Deferred Proofs

B.1 Proof of Prop 1

Consider $\pi \in \Pi_R$. If π does not contain any actions parameterized by x, then the same plan π is also a valid solution for R'. Consider the alternative, where π does contain an action parameterized by x. By Def. 1, there is another plan π' with equivalent motion sequence not parameterized by x, which is a valid solution for R'.

Now that we have shown that $\Pi_{R'}$ is not empty, we need to show that any valid plan for R' is valid for R. Consider plan $\pi = [a_1, ..., a_N] \in \Pi_{R'}$ with corresponding state plan $\mathcal{I}_\pi = [\mathcal{I}_0, ..., \mathcal{I}_N]$. If the addition of facts $\mathcal{F}$ parameterized by x make π invalid, then there must exist a state $\mathcal{I}_k$ such that $\mathcal{I}_k \cup \mathcal{F} \notin \text{Pre}(a_{k+1})$, which means that a_{k+1} is parameterized by a symbol that did not exist in R'. Only a universal or existential quantifier in $\text{Pre}(a_{k+1})$ can cause a_{k+1} to be parameterized by an additional symbol. Adding additional facts cannot turn an existentially quantified formula from true to false. By Definition 1, a_{k+1} does not have any universal quantifiers that can be parameterized by x in its precondition. Thus π must be valid for R. $\square$

B.2 Proof of Prop 2

First note that no actions in this domain have universal quantifiers, so we only need to check Definition 1.1 to show that a symbol is redundant. Consider a place p such that `(not`

`(VisitedPlace` p`))` appears in the CNF of the goal. If p parameterizes `moveRelaxed`, then `(VisitedPlace` p`)` is in the effects, violating the goal. Since `moveRelaxed` is the only action that can be parameterized by a place, no plan can parameterize p and Definition 1.1 is trivially satisfied.

Next, consider a place p that does not parameterize any initial or goal facts. For any plan π with an action parameterized by p, let $a_k^{\mathcal{P}}$ denote an action parameterized by places $\mathcal{P}$, including p. Plan π' where $a_k^{\mathcal{P}}$ is replaced by $a_k^{\mathcal{P}\setminus p}$ is also valid, since state plans $\mathcal{I}_\pi$ and $\mathcal{I}_{\pi'}$ only differ by a `(VisitedPlace` p`)` fact, and no action preconditions or goals involve this fact. As a result the command sub-sequence corresponding to $a_k^{\mathcal{P}}$ is also valid for $a_k^{\mathcal{P}\setminus p}$. So, for any π parameterized by p, we can construct π' that has an equivalent motion sequence but does not parameterize p; thus p is redundant. □

References

[1] G. Konidaris, "On the necessity of abstraction," *Current Opinion in Behavioral Sciences*, vol. 29, pp. 1–7, 2019, Artificial Intelligence, ISSN: 2352-1546.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] S. Schaal, "Learning from demonstration," *Advances in neural information processing systems*, vol. 9, 1996.

[7] A. Block, A. Jadbabaie, D. Pfrommer, M. Simchowitz, and R. Tedrake, "Provable guarantees for generative behavior cloning: Bridging low-level stability and high-level behavior," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[8] T. Chen, J. Xu, and P. Agrawal, "A system for general in-hand object re-orientation," in *Conference on Robot Learning*, PMLR, 2022, pp. 297–307.

[9] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, "Learning to walk in minutes using massively parallel deep reinforcement learning," in *Conference on Robot Learning*, PMLR, 2022, pp. 91–100.

[10] V. Makoviychuk, L. Wawrzyniak, Y. Guo, *et al.*, *Isaac gym: High performance gpu-based physics simulation for robot learning*, 2021. arXiv: 2108.10470 [cs.RO].

[11] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[12] F. Bacchus and Q. Yang, "The downward refinement property.," in *IJCAI International Joint Conference on Artificial Intelligence*, 1991, pp. 286–293.

[13] G. J. Stein, C. Bradley, and N. Roy, "Learning over subgoals for efficient navigation of structured, unknown environments," in *Conference on Robot Learning (CoRL)*, PMLR, 2018.

[14] C. Bradley, A. Pacheck, G. J. Stein, S. Castro, H. Kress-Gazit, and N. Roy, "Learning and planning for temporally extended tasks in unknown environments," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021.

[15] C. Bradley and N. Roy, "Learning feasibility and cost to guide tamp," in *Experimental Robotics*, M. H. Ang Jr and O. Khatib, Eds., Cham: Springer Nature Switzerland, 2024, pp. 203–216, ISBN: 978-3-031-63596-0.

[16] N. Hughes, Y. Chang, and L. Carlone, "Hydra: A real-time spatial perception engine for 3d scene graph construction and optimization," *CoRR*, vol. abs/2201.13360, 2022. arXiv: 2201.13360.

[17] A. Ray, C. Bradley, L. Carlone, and N. Roy, "Task and motion planning in hierarchical 3d scene graphs," *arXiv preprint arXiv:2403.08094*, 2024.

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

[19] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual review of control, robotics, and autonomous systems*, 2021.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: his life, work, and legacy*, 2022, pp. 287–290.

[21] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

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

[23] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[24] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *2009 IEEE international conference on robotics and automation*, IEEE, 2009, pp. 489–494.

[25] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.

[26] N. J. Nilsson *et al.*, *Shakey the robot.* Sri International Menlo Park, California, 1984, vol. 323.

[27] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," 1998.

[28] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[29] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[30] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2024, pp. 8749–8756.

[31] M. Ghallab, D. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.

[32] E. Karpas and D. Magazzeni, "Automated planning for robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, 2020.

[33] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[34] M. A. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," 2018.

[35] E. Fernandez-Gonzalez, E. Karpas, and B. Williams, "Mixed discrete-continuous planning with convex optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[36] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, 2020.

[37] T. Ren, G. Chalvatzaki, and J. Peters, "Extended tree search for robot task and motion planning," *arXiv preprint arXiv:2103.05456*, 2021.

[38] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Sampling-based methods for factored task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 13-14, pp. 1796–1825, 2018.

[39] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 639–646.

[40] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach.," in *Robotics: Science and systems*, Ann Arbor, MI, USA, vol. 12, 2016, p. 00 052.

[41] W. Vega-Brown and N. Roy, "Asymptotically optimal planning under piecewise-analytic constraints," in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, Springer, 2020, pp. 528–543.

[42] J. Ferrer-Mestres, G. Frances, and H. Geffner, "Combined task and motion planning as classical ai planning," *arXiv preprint arXiv:1706.06927*, 2017.

[43] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Ffrob: Leveraging symbolic planning for efficient task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.

[44] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*, IEEE, 2011, pp. 1470–1477.

[45] W. Vega-Brown and N. Roy, "Task and motion planning is PSPACE-complete," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 10 385–10 392.

[46] L. Xu, T. Ren, G. Chalvatzaki, and J. Peters, "Accelerating integrated task and motion planning with neural feasibility checking," *arXiv preprint arXiv:2203.10568*, 2022.

[47] D. Driess, J.-S. Ha, R. Tedrake, and M. Toussaint, "Learning geometric reasoning and control for long-horizon tasks from visual input," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021.

[48] B. Kim and L. Shimanuki, "Learning value functions with relational state representations for guiding task-and-motion planning," in *Conference on Robot Learning (CoRL)*, PMLR, 2020.

[49] T. Silver, R. Chitnis, A. Curtis, J. B. Tenenbaum, T. Lozano-Perez, and L. P. Kaelbling, "Planning with learned object importance in large problem instances using graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, 2021.

[50] C. Agia, K. M. Jatavallabhula, M. Khodeir, O. Miksik, V. Vineet, M. Mukadam, L. Paull, and F. Shkurti, "Taskography: Evaluating robot task planning over large 3d scene graphs," in *Conference on Robot Learning (CoRL)*, PMLR, 2022.

[51] B. Kim, L. Kaelbling, and T. Lozano-Pérez, "Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[52] T. Pan, A. M. Wells, R. Shome, and L. E. Kavraki, "Failure is an option: Task and motion planning with failing executions," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, 2022.

[53] M. Khodeir, B. Agro, and F. Shkurti, "Learning to search in task and motion planning with streams," *IEEE Robotics and Automation Letters*, 2023.

[54] S. Thrun, "Exploring artificial intelligence in the new millennium," in G. Lakemeyer and B. Nebel, Eds., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ch. Robotic Mapping: A Survey, pp. 1–35, ISBN: 1-55860-811-7.

[55] J. Lobo, L. Marques, J. Dias, U. Nunes, and A. T. de Almeida, "Sensors for mobile robot navigation," in *Autonomous Robotic Systems*, A. T. de Almeida and O. Khatib, Eds., London: Springer London, 1998, pp. 50–81, ISBN: 978-1-84628-530-1.

[56] J. Redmon, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[57] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, *et al.*, "Segment anything," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4015–4026.

[58] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, no. 6, pp. 46–57, Jun. 1989, ISSN: 0018-9162.

[59] A. Elfes, "Occupancy grids: A probabilistic framework for robot perception and navigation," AAI9006205, Ph.D. dissertation, Pittsburgh, PA, USA, 1989.

[60] H. P. Moravec, "Sensor fusion in certainty grids for mobile robots," *AI Magazine*, vol. 9, no. 2, p. 61, Jun. 1988.

[61] M. Grinvald, F. Furrer, T. Novkovic, J. J. Chung, C. Cadena, R. Siegwart, and J. Nieto, "Volumetric instance-aware semantic mapping and 3d object discovery," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 3037–3044, 2019.

[62] K. Tateno, F. Tombari, and N. Navab, "Real-time and scalable incremental segmentation on dense slam," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, pp. 4465–4472.

[63] W. N. Greene and N. Roy, "Flame: Fast lightweight mesh estimation using variational smoothing on delaunay graphs," in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2017, pp. 4696–4704.

[64] G. Narita, T. Seno, T. Ishikawa, and Y. Kaji, "Panopticfusion: Online volumetric semantic mapping at the level of stuff and things," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2019, pp. 4205–4212.

[65] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: An open-source library for real-time metric-semantic localization and mapping," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 1689–1696.

[66] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment — a modern synthesis," in *Vision Algorithms: Theory and Practice*, B. Triggs, A. Zisserman, and R. Szeliski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 298–372, ISBN: 978-3-540-44480-0.

[67] J. McCormac, R. Clark, M. Bloesch, A. Davison, and S. Leutenegger, "Fusion++: Volumetric object-level slam," in *2018 international conference on 3D vision (3DV)*, IEEE, 2018, pp. 32–41.

[68] B. Xu, W. Li, D. Tzoumanikas, M. Bloesch, A. Davison, and S. Leutenegger, "Mid-fusion: Octree-based object-level multi-instance dynamic slam," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 5231–5237.

[69] C. Galindo, A. Saffiotti, S. Coradeschi, P. Buschka, J.-A. Fernandez-Madrigal, and J. González, "Multi-hierarchical semantic maps for mobile robotics," in *2005 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2005, pp. 2278–2283.

[70] I. Armeni, Z.-Y. He, J. Gwak, A. R. Zamir, M. Fischer, J. Malik, and S. Savarese, "3d scene graph: A structure for unified semantics, 3d space, and camera," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 5664–5673.

[71] A. Rosinol, A. Violette, M. Abate, N. Hughes, Y. Chang, J. Shi, A. Gupta, and L. Carlone, "Kimera: From slam to spatial perception with 3D dynamic scene graphs," *The International Journal of Robotics Research*, vol. 40, no. 12-14, pp. 1510–1546, 2021.

[72] N. Hughes, Y. Chang, S. Hu, R. Talak, R. Abdulhai, J. Strader, and L. Carlone, "Foundations of spatial perception for robotics: Hierarchical representations and real-time systems," *International Journal of Robotics Research*, 2024.

[73] H. Bavle, J. L. Sanchez-Lopez, M. Shaheer, J. Civera, and H. Voos, "S-graphs+: Real-time localization and mapping leveraging hierarchical representations," *IEEE Robotics and Automation Letters*, vol. 8, no. 8, pp. 4927–4934, 2023.

[74] S.-C. Wu, J. Wald, K. Tateno, N. Navab, and F. Tombari, "Scenegraphfusion: Incremental 3D scene graph prediction from RGB-D sequences," in *Proc. of the IEEE/CVF Conf. on Comp. Vision and Pattern Recog.*, 2021, pp. 7515–7525.

[75] J. Strader, N. Hughes, W. Chen, A. Speranzon, and L. Carlone, "Indoor and outdoor 3D scene graph generation via language-enabled spatial ontologies," *IEEE Robotics and Automation Letters*, vol. 9, no. 6, pp. 4886–4893, 2024.

[76] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, "Learning to summarize with human feedback," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3008–3021, 2020.

[77] Q. Gu, A. Kuwajerwala, S. Morin, K. M. Jatavallabhula, B. Sen, A. Agarwal, C. Rivera, W. Paul, K. Ellis, R. Chellappa, *et al.*, "Conceptgraphs: Open-vocabulary 3D scene graphs for perception and planning," *arXiv preprint arXiv:2309.16650*, 2023.

[78] D. Maggio, Y. Chang, N. Hughes, M. Trang, D. Griffith, C. Dougherty, E. Cristofalo, L. Schmid, and L. Carlone, "Clio: Real-time task-driven open-set 3D scene graphs," *arXiv preprint arXiv:2404.13696*, 2024.

[79] A. Werby, C. Huang, M. Büchner, A. Valada, and W. Burgard, "Hierarchical open-vocabulary 3D scene graphs for language-grounded robot navigation," in *First Workshop on Vision-Language Models for Nav. and Manip. at ICRA*, 2024.

[80] Z. Dai, A. Asgharivaskasi, T. Duong, S. Lin, M.-E. Tzes, G. Pappas, and N. Atanasov, "Optimal scene graph planning with large language model guidance," *arXiv preprint arXiv:2309.09182*, 2023.

[81] M. Khodeir, A. Sonwane, R. Hari, and F. Shkurti, "Policy-guided lazy search with feedback for task and motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 3743–3749.

[82] B. Vu, T. Migimatsu, and J. Bohg, "COAST: Constraints and streams for task and motion planning," *arXiv preprint arXiv:2405.08572*, 2024.

[83] R. S. Sutton, "Between mdps and semi-mdps: Learning, planning, and representing knowledge at multiple temporal scales," 1998.

[84] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, "From skills to symbols: Learning symbolic representations for abstract high-level planning," *Journal of Artificial Intelligence Research*, vol. 61, pp. 215–289, 2018.

[85] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[86] K. Aström, "Optimal control of markov decision processes with incomplete state estimation," *J. Math*, pp. 174–205,

[87] R. Bellman, "On the theory of dynamic programming," *Proceedings of the National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952, ISSN: 0027-8424.

[88] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, "Learning policies for partially observable environments: Scaling up," in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds., 1995, pp. 362–370.

[89] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT press, 2005.

[90] H.-T. Cheng, "Algorithms for partially observable markov decision processes," Ph.D. dissertation, University of British Columbia, 1988.

[91] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman, "Acting optimally in partially observable stochastic domains," in *Aaai*, vol. 94, 1994, pp. 1023–1028.

[92] H. Kurniawati, D. Hsu, and W. S. Lee, "Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces," 2009.

[93] N. Ye, A. Somani, D. Hsu, and W. S. Lee, "Despot: Online pomdp planning with regularization," *J. Artif. Int. Res.*, vol. 58, no. 1, pp. 231–266, Jan. 2017, ISSN: 1076-9757.

[94] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," in *Advances in Neural Information Processing Systems*, 2010.

[95] Z. Sunberg and M. Kochenderfer, "Online algorithms for pomdps with continuous state, action, and observation spaces," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018, pp. 259–263.

[96] S. Karaman and E. Frazzoli, "High-speed flight in an ergodic forest," *International Conference on Robotics and Automation (ICRA)*, 2012.

[97] N. Roy and C. Earnest, "Dynamic action spaces for information gain maximization in search and exploration," in *American Control Conference (ACC)*, Minneapolis, MN, 2006.

[98] I. Arvanitakis, K. Giannousakis, and A. Tzes, "Mobile robot navigation in unknown environment based on exploration principles," in *Conference on Control Applications (CCA)*, Sep. 2016.

[99] R. Nasir and A. Elnagar, "Gap navigation trees for discovering unknown environments," *Intelligent Control and Automation*, vol. 6, no. 04, p. 229, 2015.

[100] C. Richter, "Autonomous navigation in unknown environments using machine learning," Ph.D. dissertation, Massachusetts Institute of Technology, 2017.

[101] C. Richter, J. Ware, and N. Roy, “High-speed autonomous navigation of unknown environments using learned probabilities of collision,” in *International Conference on Robotics and Automation (ICRA)*, 2014.

[102] C. Richter and N. Roy, “Safe visual navigation via deep learning and novelty detection,” Jul. 2017.

[103] A. Pnueli, “The temporal logic of programs,” in *18th annual symposium on foundations of computer science (sfcs 1977)*, ieee, 1977, pp. 46–57.

[104] H. Kress-Gazit, M. Lahijanian, and V. Raman, “Synthesis for Robots: Guarantees and Feedback for Robot Behavior,” *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[105] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, “Temporal logic motion planning for dynamic robots,” *Automatica*, 2009.

[106] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, in *International Conference on Robotics and Automation (ICRA)*.

[107] B. Lacerda, D. Parker, and N. Hawes, “Optimal and dynamic planning for markov decision processes with co-safe ltl specifications,” in *International Conference on Intelligent Robots and Systems (IROS)*, 2014.

[108] S. L. Smith, J. Tûmová, C. Belta, and D. Rus, “Optimal path planning for surveillance with temporal-logic constraints,” *International Journal of Robotics Research (IJRR)*, vol. 30, no. 14, 2011.

[109] M. Bouton, J. Tumova, and M. J. Kochenderfer, “Point-based methods for model checking in partially observable markov decision processes.,” in *Association for the Advancement of Artificial Intelligence (AAAI)*, 2020.

[110] M. Ahmadi, R. Sharan, and J. W. Burdick, “Stochastic finite state control of pomdps with ltl specifications,” *arXiv*, 2020.

[111] A. I. M. Ayala, S. B. Andersson, and C. Belta, “Temporal logic motion planning in unknown environments,” in *International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[112] S. Sarid, B. Xu, and H. Kress-Gazit, "Guaranteeing High-Level Behaviors While Exploring Partially Known Maps," in *Robotics: Science and Systems (RSS)*, 2012.

[113] M. Lahijanian, M. R. Maly, D. Fried, L. E. Kavraki, H. Kress-Gazit, and M. Y. Vardi, "Iterative Temporal Planning in Uncertain Environments With Partial Satisfaction Guarantees," *IEEE Transactions on Robotics (TRO)*, 2016.

[114] M. Guo and D. V. Dimarogonas, "Multi-agent plan reconfiguration under local ltl specifications," *International Journal of Robotics Research (IJRR)*, vol. 34, no. 2, 2015.

[115] M. Guo, K. H. Johansson, and D. V. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *International Conference on Robotics and Automation (ICRA)*, 2013.

[116] C.-I. Vasile, K. Leahy, E. Cristofalo, A. Jones, M. Schwager, and C. Belta, "Control in belief space with temporal logic specifications," in *Conference on Decision and Control (CDC)*, 2016.

[117] Y. Kantaros, M. Malencia, V. Kumar, and G. J. Pappas, "Reactive temporal logic planning for multiple robots in unknown environments," in *International Conference on Robotics and Automation (ICRA)*.

[118] Y. Kantaros and G. Pappas, "Optimal temporal logic planning for multi-robot systems in uncertain semantic maps," in *International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[119] M. Svoreňová, M. Chmelík, K. Leahy, H. F. Eniser, K. Chatterjee, I. Černá, and C. Belta, "Temporal logic motion planning using pomdps with parity objectives: Case study paper," in *International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2015.

[120] M. L. Littman, U. Topcu, J. Fu, C. Isbell, M. Wen, and J. MacGlashan, "Environment-independent task specifications via gltl," *arXiv*, 2017.

[121] J. Fu and U. Topcu, "Probably Approximately Correct MDP Learning and Control With Temporal Logic Constraints," in *Robotics: Sciene and Systems (RSS)*, 2015.

[122] X. Li, Z. Serlin, G. Yang, and C. Belta, "A formal methods approach to interpretable reinforcement learning for robotic planning," *Science Robotics*, 2019.

[123] R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, "Teaching multiple tasks to an rl agent using ltl," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018.

[124] X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," in *International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[125] D. Sadigh, E. S. Kim, S. Coogan, S. S. Sastry, and S. A. Seshia, "A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications," in *Conference on Decision and Control (CDC)*, 2014.

[126] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, "Combining neural networks and tree search for task and motion planning in challenging environments," in *International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[127] S. Carr, N. Jansen, and U. Topcu, "Verifiable rnn-based policies for pomdps under temporal logic constraints," in *IJCAI International Joint Conference on Artificial Intelligence*, 2020.

[128] M. Merlin, N. Parikh, E. Rosen, and G. Konidaris, "Locally observable markov decision processes," in *ICRA 2020 Workshop on Perception, Action, Learning*, 2020.

[129] O. Madani, "On the computability of infinite-horizon partially observable markov decision processes," *Association for the Advancement of Artificial Intelligence (AAAI)*, 1999.

[130] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," in *Formal Methods in System Design*, 2001.

[131] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for LTL and ω-automata manipulation," in *Automated Technology for Verification and Analysis (ATVA)*, Springer, 2016.

[132] J. Pineau and S. Thrun, "An integrated approach to hierarchy and abstraction for POMDPs," CMU, Tech. Rep., 2002.

[133] J. Li, G. Pu, L. Zhang, Z. Wang, J. He, and K. Guldstrand Larsen, "On the relationship between ltl normal forms and büchi automata," in *Theories of Programming and Formal Methods*. 2013.

[134] Unity Technologies, *Unity game engine*, unity3d.com, 2019.

[135] T. Yamamoto, K. Terada, A. Ochiai, F. Saito, Y. Asahara, and K. Murase, "Development of HSR as the research platform of a domestic mobile manipulator," *ROBOMECH*, 2019.

[136] *Ricoh Theta S Panoramic Camera*, 2019.

[137] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Nov. 2011.

[138] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[139] G. M. J. Chaslot, M. H. Winands, H. J. v. d. Herik, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," *New Mathematics and Natural Computation*, 2008.

[140] R. Coulom, "Computing "elo ratings" of move patterns in the game of go," *ICGA journal*, 2007.

[141] D. Speck, R. Mattmüller, and B. Nebel, "Symbolic top-k planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[142] D. Xu, A. Mandlekar, R. Martín-Martín, Y. Zhu, S. Savarese, and L. Fei-Fei, "Deep affordance foresight: Planning through what can be done in the future," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021.

[143] H. Oleynikova, Z. Taylor, R. Siegwart, and J. Nieto, "Sparse 3D topological graphs for micro-aerial vehicle planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[144] A. L. Blum and M. L. Furst, “Fast planning through planning graph analysis,” *Artificial intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.

[145] M. Fishman, N. Kumar, C. Allen, N. Danas, M. Littman, S. Tellex, and G. Konidaris, “Task scoping: Generating task-specific simplifications of open-scope planning problems,” in *PRL Workshop Series –Bridging the Gap Between AI Planning and Reinforcement Learning*, 2023.

[146] M. Ahn, D. Dwibedi, C. Finn, M. G. Arenas, K. Gopalakrishnan, K. Hausman, B. Ichter, A. Irpan, N. Joshi, R. Julian, *et al.*, “Autort: Embodied foundation models for large scale orchestration of robotic agents,” *arXiv preprint arXiv:2401.12963*, 2024.

[147] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, *et al.*, “Do as i can, not as i say: Grounding language in robotic affordances,” *arXiv preprint arXiv:2204.01691*, 2022.

[148] K. Rana, J. Haviland, S. Garg, J. Abou-Chakra, I. Reid, and N. Suenderhauf, “Sayplan: Grounding large language models using 3d scene graphs for scalable robot task planning,” in *7th Annual Conference on Robot Learning*, 2023.

[149] Y. Chen, J. Arkin, C. Dawson, Y. Zhang, N. Roy, and C. Fan, “Autotamp: Autoregressive task and motion planning with llms as translators and checkers,” in *2024 IEEE International conference on robotics and automation (ICRA)*, IEEE, 2024, pp. 6695–6702.

[150] D. Shah, M. R. Equi, B. Osiński, F. Xia, B. Ichter, and S. Levine, “Navigation with large language models: Semantic guesswork as a heuristic for planning,” in *Conference on Robot Learning*, PMLR, 2023, pp. 2683–2699.