Planning to Optimize and Learn Reward in Navigation Tasks in Structured Environments with Time Constraints

Max Korein

CMU-RI-TR-21-53

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

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15217

July 2021

Keywords: Orienteering Problem, Structured Environments, Reinforcement Learning

To my parents

Abstract

Planning problems in which an agent must perform tasks for reward by navigating its environment while constrained by time and location have a wide variety of applications in robotics. Many real-world environments in which such planning problems apply, such as office buildings or city streets, are very structured. They consist of passages with notable locations along them such as streets or hallways, and notable locations might be grouped into clusters such as floors in a building divided by bottlenecks such as elevators.

In this thesis, we introduce algorithms designed for an agent performing tasks for reward while constrained by time and location in a structured environment. Our goal is specifically to exploit the structure of the environment when planning in order to improve the reward received. We present three algorithms that do so in different ways. First, we present the Task Graph algorithm, which groups actions the agent can perform into larger "tasks" and creates a plan from a sequence of tasks, rather than individual actions. We test the algorithm in a problem we call "Exploration Scheduling," in which an agent seeks to gather information in the free time between mandatory tasks in order to improve the reward received from the mandatory tasks, and find it yields strong results.

Second, we present the RegionPlan algorithm, which divides the environment into regions based on its structure and creates a separate plan for each region to choose from. We test the RegionPlan algorithm's ability to solve a variant of the orienteering problem in simulated structured environments, and demonstrate environment structures and reward distributions in which the RegionPlan approach allows the agent to avoid getting stuck at certain locally-optimal solutions.

Third, we present Route-Based Multi-Level Variable Neighborhood Search (Route-MLVNS), a variable neighborhood search algorithm that represents solutions to the problem as routes the agent takes through the environment, rather than sequences of actions. In this way, Route-MLVNS takes into account the locations that the agent passes by as it travels when evaluating the reward of a location, rather than only considering the reward of traveling to a single location at a time. We test Route-MLVNS in a simulation of the orienteering problem in structured environments, comparing it to an MLVNS algorithm from the literature that treats solutions as ordinary plans of tasks, and find Route-MLVNS to yield superior results. We also demonstrate a unique capability that Route-MLVNS has, the ability to solve the orienteering problem for a continuous range of time constraints simultaneously rather than only for a single time constraint or discrete set of time constraints.

Finally, we consider scenarios in which an agent solves repeated instances of the orienteering problem in a structured environment with stochastic rewards that are initially unknown. We present algorithms which treat locations in the environment as levers in a multi-armed bandit problem and assign them values based on both the expected reward and the potential value of the knowledge gained by visiting them, and use planning algorithms based on the environment structure to find a plan that maximizes the assigned value received. We perform tests demonstrating the algorithms' effectiveness at both receiving reward over

time and learning an effective model. Additionally, we analyze the relationship between the structure of the environment and the distribution of the data received by the agent.

Overall, our work demonstrates the benefit of an agent considering and exploiting the structure of its environment when planning tasks, and how the structure affects the data it can gather in environments with unknown stochastic rewards.

Acknowledgements

I would first like to thank my parents. Without their words of encouragement, their willingness to listen to me vent, their ability to help me calm down whenever I felt overwhelmed, and their continued belief that I was capable of getting my PhD even when I'd stopped believing it myself were all invaluable. I would also like to thank the rest of my family, especially my brother Dorian and my sister Emma, whose friendship has been a great source of comfort while their accomplishments have been a great source of pride.

I would like to thank my advisor, Manuela Veloso. I learned so much about research from Manuela, who always had good advice and encouragement when I needed it while also helping me get back on track whenever I found myself getting lost. I know that advising me took a lot of patience at times. Thank you, Manuela, for sticking with me and seeing this through to the end.

I would like to thank Reid Simmons. Reid was not just a member of my thesis committee, but part of the reason I came to the Robotics Institute in the first place. And although he never advised me in an official capacity, Reid was always willing to meet with me to provide advice and feedback when I needed it, and his insight was crucial in the development of my research and thesis.

I would also like to thank the other members of my thesis committee, Illah Nourbakhsh and Peter Stone, for their invaluable feedback and support throughout the process of proposing, writing, and defending my thesis.

I would like to thank my girlfriend, Jess Mrzlack, for providing me company and support throughout this process, and for enduring all of my stressing, venting, and late nights spent working.

I would like to thank the members of the CORAL Lab, for supporting me and giving feedback on my various presentations throughout the years, including Anahita Mohseni-Kabir for helping me to organize a last-minute practice talk for my defense.

I would like to thank my therapist, Jeff Beyer, for helping me learn to handle the stress and frustration of being a PhD student.

I would like to thank all the various friends who have supported me during this whole process. I would like to thank Ada Zhang: Commiserating over our struggles as PhD students helped me feel like I wasn't alone, and seeing you graduate helped me realized I could too. I would like to thank Jonathan Jaquette, who was happy to discuss my thesis when I needed to talk about it or discuss anything else when I needed a break, and who attended my defense to support me even though it was completely outside his field of study.

Finally, I would like to thank my cats, who provided company, comfort, and something soft to pet whenever I needed it most.

Contents

Chapter 1

Introduction

Planning algorithms are often designed to be general-purpose and independent of the structure of the environment in which the planning takes place. However, the environments in which robots operate are frequently very structured. Indoor service robots often operate in buildings that are structured into floors connected by elevators and stairs. Each floor is typically a network of connected hallways with rooms along them. Driving robots operate on streets, which like the hallways of many office buildings form networks of linear paths with points of interest located along them. The goal of this thesis is to study ways in which an agent can exploit the structure of its environment in order to plan more effectively.

Our specific focus is planning problems in which an agent seeks to gather reward by performing tasks at different locations in its environment while constrained by a time limit and fixed start and end locations. Performing a task requires traveling to a specific location in the environment and spending time there, and the agent cannot perform the same task repeatedly and must instead perform tasks at a variety of locations. We consider situations in which the expected reward of visiting a location is known and the agent must maximize the reward received over a single iteration of the planning problem, as well as situations in which the expected reward is initially unknown and the agent must maximize the reward received over many iterations.

This type of planning problem has a wide variety of applications. The one that served as the original motivation for our work is service robots with spare time in between user requests. For example, the CoBots are mobile autonomous service robots that operate in office buildings and perform services scheduled by users through an online interface[**?**, **?**]. When CoBots are not performing a service scheduled by a user, they have spare time during which they could perform other useful actions, such as gathering information about the building or offering food or other potentially useful objects to users in their offices. CoBots operate primarily in office buildings, which are typically environments with structural features that could be exploited when planning. An example of another application of our work is an autonomous delivery robot planning a set of deliveries to customers' houses for a day, which could take advantage of the structured nature of city streets.

In this thesis, we develop algorithms for an agent such as a service robot or autonomous delivery robot to maximize the reward received in a structured environment, both in a single planning scenario with known rewards or over repeated scenarios in the same environment with initially unknown rewards. Our algorithms focus on exploiting the structure of the environment in order to change the size and structure of the solution space to improve the reward received.

1.1 Thesis Question

This thesis seeks to answer the question:

> For an agent creating a plan of tasks to perform for reward while constrained by time and location, how can the structure of the agent's environment be exploited to increase the reward received, and how does it affect the agent's ability to learn unknown stochastic reward functions?

Our work focuses on scenarios in which an agent has a map of its environment and is capable of navigating that environment autonomously without any problems. We assume that the environment has certain structural features that appear in many real-world environments, such most notable locations being spread out along hallways or the environment being divided into multiple sections separated by a bottleneck (such as stories of a building separated by an elevator). We assume the agent can travel to different locations in the environment and perform tasks, which yield reward representing the usefulness of those tasks or the probability of those tasks succeeding. The goal is to maximize the reward the agent receives from the tasks it performs. The agent is constrained by a time limit and a given end location where it needs to be when the time limit ends. The rewards could be known, or they could be unknown and need to be learned by the agent based on the reward it received from performing tasks in the past.

In this planning problem, there are potential benefits to considering the structure of the environment. Consider, for example, a multi-story office building in which the only way to travel between floors is by a single elevator. In such a case, we would expect that the optimal solution will frequently be one that minimizes the number of elevator uses, and we can likely ignore plans that use the elevator more times than necessary, reducing the search space and allowing us to plan more efficiently.

For another example, consider a hallway containing a number of locations at which the agent can perform tasks for reward. If the agent travels to the far end of the hallway to perform a task there, there is an expected reward for that task. But in the process, the agent also passes by each other location on that hallway at which it could perform a task. When evaluating the expected reward of traveling to the far end of that hallway to perform a task, the agent could consider not just the expected reward of that task, but also the potential value of passing by the other locations where tasks could be performed in the process.

Our first goal in this thesis is to exploit the properties of the environment's structure when planning. By taking the structure of the environment into account, we can potentially create planning heuristics that yield more reward than ones designed without a structured environment in mind. Our second goal is to consider how the structure of the environment affects the agent in situations where the rewards of tasks the agent can perform are initially unknown. In those cases, the structure of the environment can potentially affect the distribution of data the agent is able to gather. Additionally, we would like the agent to address the need to learn a model of the reward functions in a way that does not interfere with its ability to exploit the structure of the environment when planning.

1.2 Approach

In this thesis, we contribute three different planning algorithms that are each designed to exploit the structure of the agent's environment in different ways. We also present two different learning algorithms for scenarios in which the rewards of tasks are initially unknown and must be learned over time. Our algorithms are intentionally not mutually exclusive – they are designed to be modular, so that they can be applied individually or together, or combined with other planning algorithms, in order to suit the needs of a particular scenario based on the properties of the environment.

Our first approach to exploiting the structure of the environment when planning is the Task Graph algorithm, an algorithm that uses the structure of the environment to group individual tasks into larger meta-tasks, and creates a "task graph" in which the nodes are larger tasks rather than individual locations. It then plans over the task graph, rather than the original map of the environment. The task graph algorithm is based around the intuition that certain actions are unlikely to be part of the optimal plan by themselves without other nearby actions. For example, it is unlikely that the optimal plan includes traveling partway down a hallway and then turning around, so by grouping the nodes in a hallway into a single task and only considering entire hallways rather than each individual location on each hallway, we can simplify the planning problem while being unlikely to eliminate the optimal plan from consideration. The modularity of the task graph algorithm comes from the fact that there is flexibility in both how tasks are chosen and what algorithm is used to find a plan on the task graph.

Our second approach to planning is the RegionPlan algorithm, which divides the environment into regions based on the structure of the environment and creates a separate plan for each region to choose from. The RegionPlan algorithm is designed for environments structured into distinct sections, such as floors in a multi-story environment, in which we would expect the optimal plan to be concentrated in one section of the environment rather than being spread throughout multiple sections. The modularity of the RegionPlan algorithm comes from the fact that any planning algorithm can be used to create the plans for each region, including other algorithms that exploit the environmental structure of the individual regions.

For environments that consist of a network of hallways, we contribute the concept of Route-Based Task Planning. Route-Based Task Planning is a framework for planning in hallway-based environments in which the agent represents solutions as routes it can travel through the environment, rather than sequences of actions it can perform. The purpose of Route-Based Task Planning is three-fold. First, when evaluating the value of performing an action at a location, it takes into account the other locations the agent passes on the way, as there will frequently be many such locations in a hallway-based environment. Second, it changes the structure of the solution space, which affects the way that local search algorithms can explore the solution space in a way we believe would be beneficial. Third, a single route through the environment can represent multiple different plans of different lengths.

Route-Based Task Planning is a flexible framework, and its route-based representation of plans could be applied to a wide variety of planning algorithms. We use it to create the Route-Based Multi-Level Variable Neighborhood Search (Route-MLVNS), an algorithm designed for finding solutions to the Orienteering Problem by taking advantage of the benefits of Route-Based Task Planning. Route-MLVNS works by creating a plan or set of plans and performing local searches of other plans in various neighborhood structures, and is capable of solving the Orienteering Problem for a continuous range of time constraints simultaneously.

We also consider scenarios in which the expected rewards of performing tasks are unknown, and must be learned through the execution of many plans. We do not directly exploit the structure of the environment to our advantage when approaching the need to learn the reward of tasks. Instead, our focus is on finding a method for solving the trade-off between exploration and exploitation that is completely independent from the planning algorithm being used. The goal is that the planning algorithm being used can be freely selected based on the structure of the environment without concern for whether or not it will be compatible with the learning algorithm.

In order to create a learning algorithm that is independent of the planning algorithm used, we treat the locations the agent can perform tasks as if they are levers in a multi-armed bandit problem. Some approaches to the multi-armed bandit problem consist of computing a value for each lever, then pulling the lever with the highest value. We create two algorithms — Planning UCB1 and Planning Thompson Sampling — inspired by multi-armed bandit algorithms, which assign values to each action the agent can perform, then treat those values as if they are known rewards and find a plan that maximizes the "reward" received. The result is a modular system where the learning algorithm used to assign values, and the planning algorithm used to maximize the value of the plan, can be chosen separately based on what is best-suited to the scenario and the agent's environment.

1.3 Contributions

The key contributions of this thesis are as follows:

- A formalization of structural features that exist in many real-world environments, and a discussion of how those features can impact the effectiveness of existing planning approaches or be exploited when planning.
- The **Task-Graph** algorithm for planning in structured environments by grouping actions together into single larger tasks and planning over a graph of those tasks, rather than a map of the environment.

- The **RegionPlan** algorithm for planning in a structured environment by dividing it into regions and creating separate plans for each region.
- The **Route-Based Task Planning** framework for approaching planning problems in hallway-based environments by considering the route the agent travels through the environment rather than the individual actions it performs, and the **Route-MLVNS** algorithm that applies RBTP to plan in hallway-based environments.
- The **Planning UCB1** and **Planning Thompson Sampling** algorithms for solving the exploration versus exploitation tradeoff in scenarios where the agent begins without a model of reward functions that works independently of the planning algorithm being used by treating tasks as levers in a multi-armed bandit problem.
- A discussion of how the structure of an agent's environment affects its ability to learn unknown reward functions of performing tasks at different locations.

1.4 Thesis Outline

Chapter ?? describes the orienteering problem, a planning problem related to much of our work. It also contributes definitions of specific structural features that appear in real world environments that we seek to exploit, as well as a discussion of how those features affect an agent's ability to solve orienteering-like problems in a structured environment.

Chapter ?? contributes the Task Graph algorithm. We demonstrate the effectiveness of the Task Graph algorithm in a simulated scenario in which a service robot's spare time tasks consist of gathering information that will improve the robot's ability to schedule user requests at the optimal time.

Chapter ?? contributes the RegionPlan algorithm. We demonstrate scenarios in which the RegionPlan improves the reward of the solution found in a variant of the Orienteering Problem.

Chapter ?? contributes Route-Based Task Planning and Route-MLVNS. We compare Route-MLVNS in simulation to a similar algorithm algorithm from the literature that does not use Route-Based Task Planning to demonstrate the effectiveness of RBTP.

Chapter ?? contributes Planning UCB1 and Planning Thompson Sampling, our algorithms for addressing scenarios in which the agent must learn the expected reward of tasks from experience over time. We demonstrate the effectiveness of both algorithms in simulation and discuss the relationship between the structure of the environment and the distribution of data that the agent gathered in order to learn a model of the reward functions.

Chapter ?? discusses existing research relating to our work.

Chapter ?? concludes the thesis by summarizing our contributions and discussing possible directions for future work.

Chapter 2

Problem Description

In this chapter, we define the problems and terms that we focus on solving in this thesis. First, we describe a planning problem called the orienteering problem. Much of this thesis focuses on solving variants of the orienteering problem in structured environments.

Next, we describe what we mean by a "structured environment." We define two key environmental features that are the focus of this thesis: Hallways and bottlenecks. We then contribute a discussion of different ways in which the presence of hallways and bottlenecks in an agent's environment affects its ability to solve the orienteering problem and related planning problems.

2.1 The Orienteering Problem

The orienteering problem is a planning problem in which a agent seeks to maximize the reward received by visiting locations in its environment while constrained by a fixed start location, end location, and time limit[**?**]. The agent is given a graph G representing its environment. The graph has nodes V representing the locations in the environment at which the agent can travel to received reward. The edges E represent the paths along which the agent can travel to get from node to node, and their lengths represent the time to travel along the edges.

For each node the agent is given a reward function $R(v)$, which determines the reward it receives when it visits that node for the first time. The agent is also given a start location v_s, an end location v_e, and a time limit t_{max}. The goal is to find a path p from v_s to v_e with length less than t_{max} that maximizes the reward received.

The orienteering problem is NP-Hard and impractical to solve optimally, so research on the problem and its variants have focused on finding approximation and heuristic algorithms. In this thesis, we focus primarily on devising heuristics for solving orienteering-like problems in certain types of structured environments.

This chapter is focused on discussing how the structure of the environment can impact the orienteering problem, including how methods used by many existing heuristics may be rendered inefficient or ineffective by the environment's structural features. We present a more thorough discussion of existing research that has been done on the orienteering problem and its variants in Chapter **??**.

2.2 Structured Environments

When we talk about the structure of the environment, we are referring to the topology of the graph G that results from the way the nodes are connected by edges. Specifically, when we describe an environment as "structured," we are referring to environments in which most nodes have a small number of edges, and there are certain structural features present. The specific structures that we focus on in this thesis are "hallways" and "bottlenecks."

Hallways

The environment feature that this thesis focuses on most is hallways. We define a hallway as a sequence of nodes $[v_0, v_1, \ldots, v_{n-1}, v_n]$ such that there is an edge from v_0 to v_1, v_1 to v_2, and so on, v_0 and v_n each have either exactly one edge (a "dead end") or three or more edges (an "intersection"), and v_1 through v_{n-1} each have exactly two edges. In other words, the hallway forms a path from v_0 to v_n that visits each node in the hallway in order, and the hallway can only be existed from either end — if an agent starts traveling down the hallway from v_0, the only way to reach a node not in the hallway is to traverse the entire hallway and leave by passing v_n or to turn around and backtrack to v_0.

Figures **??**, **??**, **??**, and **??** show examples of simple environments comprised entirely of hallways. Some examples of hallway structures that appear in real-world environments are hallways in an office building or streets in a city. The environments we consider in this thesis are constructed primarily, and in some cases exclusively, of networks of hallways.

Bottlenecks

Another structural feature some real-world environments have that we consider is bottlenecks. We define a bottleneck as a node, edge, or hallway that connects multiple sections of an environment, serving as one of the few, or only, ways to travel between sections. In the case where a bottleneck is an edge, we can think of the graph as being partitioned and the edges between partitions as bottlenecks.

Figures **??**, **??**, **??**, and **??** show environments each containing a bottleneck. In Figure **??**, the node E serves as a bottleneck that must be passed through to get from one side of the environment to the other. In Figure **??**, the edge between E and F is a bottleneck (alternatively, E or F itself could be treated as a bottleneck). In Figure **??**, the hallway [E, F, G, H] is a bottleneck (or any node or edge along the hallway). Finally, in Figure **??**, the central node E is a bottleneck that must be passed through to travel between any of the six hallways extending out from it.

Some examples of bottlenecks in real-world environments are bridges in a city that is divided by a river or elevators connecting multiple floors in an office building.

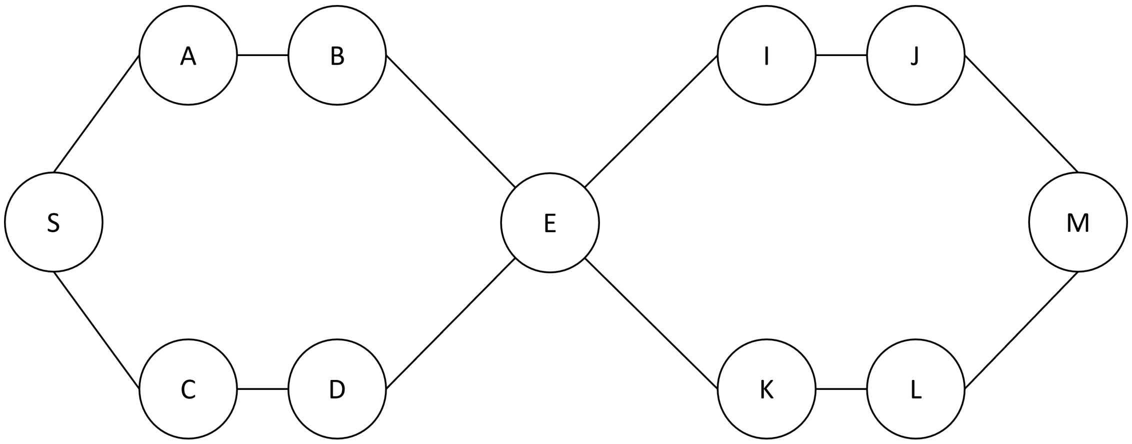


Figure 2.1: An example environment comprised of hallways in which the node E acts as a bottleneck dividing the left and right sections.

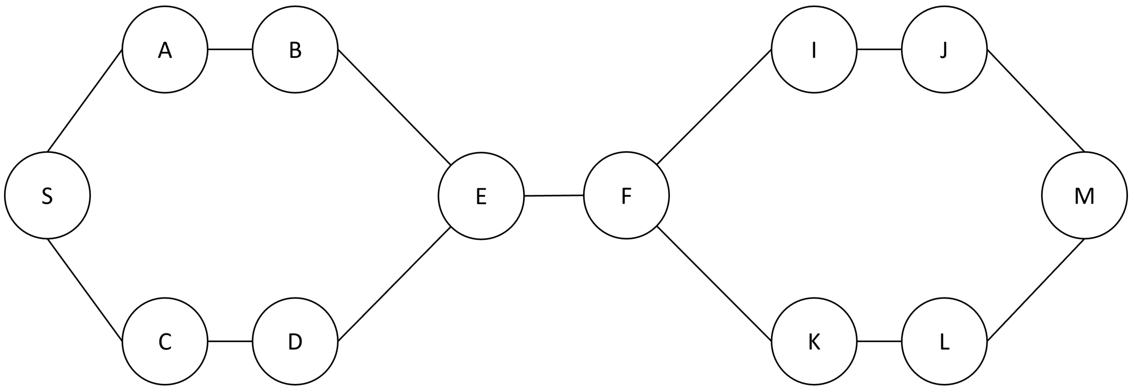


Figure 2.2: An example environment comprised of hallways in which the edge between E and F acts as a bottleneck dividing the left and right sections.

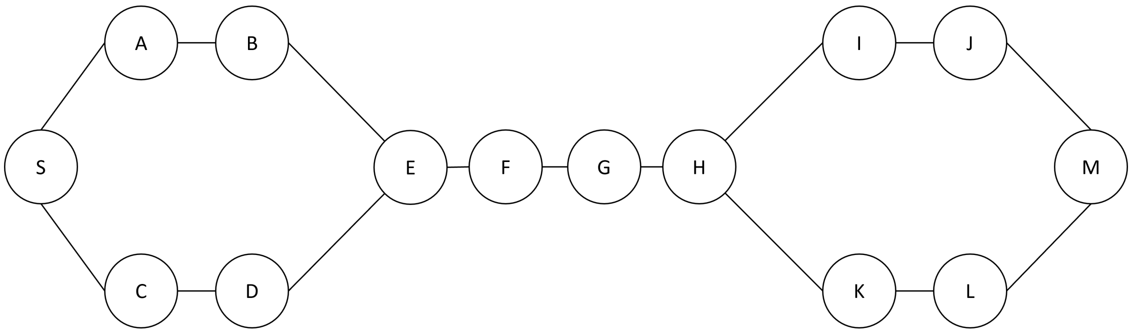


Figure 2.3: An example environment comprised of hallways in which the hallway [E, F, G, H] acts as a bottleneck dividing the left and right sections.

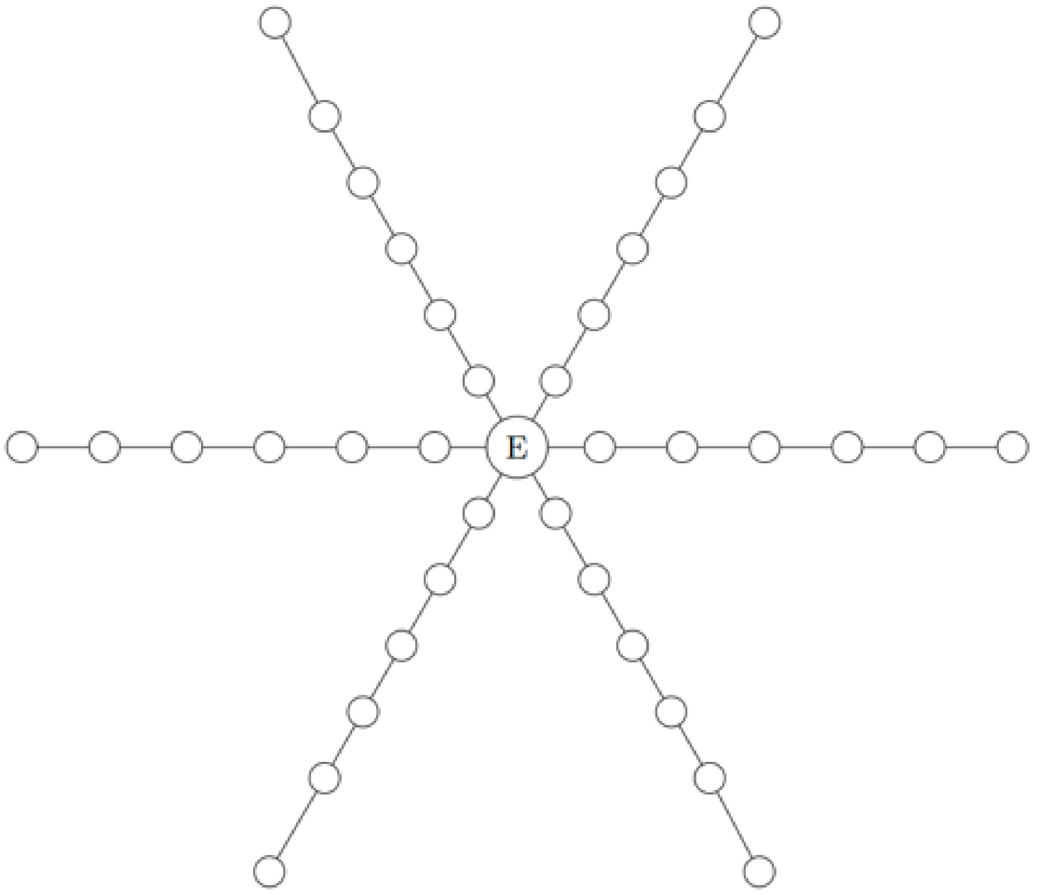


Figure 2.4: An example environment comprised of hallways in which the center node E acts as a bottleneck dividing the six hallways.

2.3 Properties of Structured Environments

When an environment is composed of hallways and bottlenecks, it has various properties that can arise that could potentially hinder an agent's ability to plan if not taken into account, or improve its ability to plan if they are considered. In this section, we discuss some of the properties that we consider in this thesis.

2.3.1 Limitations on Local Search Operations

Many algorithms used to solve the orienteering problem include a local search step that searches for new plans similar to one that has already been found[**?**]. The local search typically consists of performing simple operations that transform a plan and using some criteria to determine if the new plan should be further searched or rejected. The operations generally fall into one of two categories: they are either designed to reduce the length of the

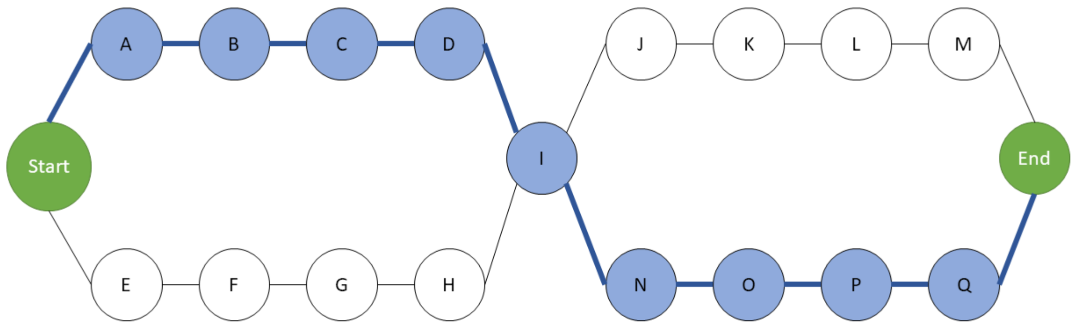


Figure 2.5: An example scenario that demonstrates the ineffectiveness of a "replace" operation for a local plan search in a hallway-based environment.

plan or increase the reward of the plan. Some examples of commonly-used operations are exchanging the position of two nodes within a plan or replacing a node in the plan with a different node not in the plan.

Those operations, however, may be inefficient in hallway-based environments. Consider the "Replace" operation that replaces a node in a plan with one not in the plan. Assume the algorithm is operating in the environment shown in Figure **??**. Assume every edge has a length of 1, v_s is the node labeled "Start," v_e is the node labeled "'End," and $t_{max} = 10$.

Assume the agent is currently considering the solution $p = [Start, A, B, C, D, I, N, O, P, Q, End]$ shown by the nodes in blue. It is exploring new, similar solutions by performing a Replace operation. In this case, there are no possible Replace operations. Removing any node in the plan and replacing it with a node not in the plan will result in a plan that exceeds the length limit. The structure of the environment, and the way it restricts the agent's ability to navigate, limits the effectiveness of the "replace" operation. An operation that can replace one hallway with another, such as removing A, B, C, and D from the plan and inserting E, F, G, and H, might be more useful.

We we address this challenge in this thesis with the Route-MVLNS in Chapter **??**. With Route-MLVNS, we represent solutions in such a way that simple search operations commonly used for algorithms that solve the orienteering problem can result in significant changes to a plan that allow them to be effective in hallway-based environments.

2.3.2 On the Way Nodes

In many cases in structured environments, there is no direct path between two given nodes. Instead, the shortest path between two nodes will involve traveling past other nodes that are "on the way." This can be very relevant when planning. In a scenario in which an agent receives reward automatically for visiting a node, this means that the process of traveling to any given node in the environment yields not just the reward of the destination node itself, but also each node on the way. Even in scenarios in which an agent must spend time at a location to receive reward, such as stopping at the location to perform a task, it can perform a task at those locations with no additional travel time.

In hallway-based environments, the shortest path between any two nodes in environment will typically have some, possibly many, nodes that are one the way. Meanwhile, in environments with bottlenecks, nodes at or near the bottleneck will often be on the way in plans that involve traveling through the bottleneck.

When evaluating the benefit of inserting a node into a plan in a hallway-based environment, it may be useful to consider not just the node itself, but other nodes that are on the way to or from that node. Consider, for example, an agent doing a local search in the environment shown in Figure **??**. The agent's current plan is simply [S, E] and it is performing a local search using an "insert" operation which inserts a node not in the plan into the plan. It inserts the node B into the plan and evaluates the result.

Without considering nodes "on the way," the plan [S, B, E] has a reward equal to the reward of B. However, in the process of traveling from S to B and from B to E, the agent will pass through nodes A and C in the process. In order to properly determine the value of inserting node B into the plan, the algorithm needs to consider not just B itself, but also the other nodes that the agent will pass by on the way to and from B.

We seek to address this with our Route-MLVNS algorithm in Chapter **??**. The Route-MLVNS algorithm represents plans in such a way that when it evaluates a plan, it considers the path the agent will take while traveling between nodes in the plan, and takes into account all nodes that it passes in the process.

The presence of on-the-way nodes in most plans also has an effect in situations in which the agent must learn the expected reward of tasks it can perform, rather than being given that information. In those cases, it is notable that some nodes will be on the way more often than others. For example, nodes at or near bottlenecks will be on the way significantly more often than nodes that are not near any bottlenecks. As a result, the distribution of data that the agent is able to acquire to learn the expected reward of tasks in the environment will not be uniform or independent in a structured environment with bottlenecks and hallways. We explore this dynamic and its impact on the agent's ability to learn an effective model of the reward of its tasks in Chapter **??**.

2.3.3 Backtracking and Choices

Another important property of structured environments is that backtracking — traveling past locations that the agent has already been earlier in the plan — is sometimes necessary,

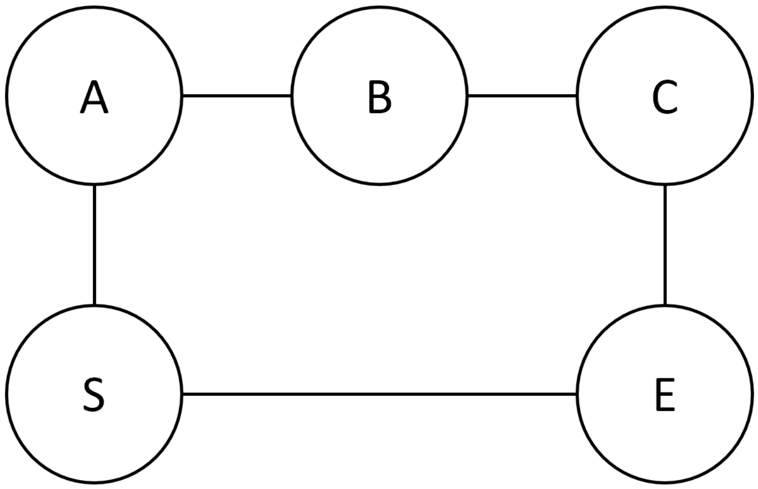


Figure 2.6: An example scenario demonstrating the importance of considering "on the way" nodes. If the agent begins with the plan [S, E] and wants to determine the value of inserting B into the plan, it will pass by A and C in the process and should consider the reward of all three nodes, not just B.

but frequently very costly. Consider, for example, the environments shown in Figures **??** and **??**. Assume the agent's start location is the node labeled S and the end location is the node labeled I. Consider the case where the planning algorithm begins by having the agent travel to A to received a reward.

In the more connected environment shown in Figure **??**, the agent can choose between continuing down the top path to node B, or it can travel across the environment to E, F, G, or H. If it does choose to travel to B, it can still then travel from B across to E, F, G, or H. The choice between the top path and bottom path is not a significant commitment because the agent can easily cross between paths.

Compare this to the more structured environment show in Figure **??**. Once the agent has gone to A, in order to reach the bottom path it has to backtrack past S. If it goes all the way to B, then switching to the bottom path becomes even more costly. As a result, the initial choice of which path to begin traveling along is much more significant due to the high cost of switching paths.

The Task Graph algorithm we present in Chapter **??** is designed partly to address this issue. The TaskGraph algorithm groups nodes in the environment into tasks, such as traversing a hallway. In this way, it treats entering a hallway as a commitment to traversing the entire hallway, without considering the possibility of traveling partway down a hallway and then backtracking out of it.

We address this issue in a different way with the RegionPlan algorithm we present in Chapter **??**. The RegionPlan algorithm is designed around breaking the environment into regions and creating plans limited to a single region. In the example environment in Figure **??**, it could use the top and bottom paths as different regions, treating the planning problem not as a choice between eight nodes, but as a choice between two hallways.

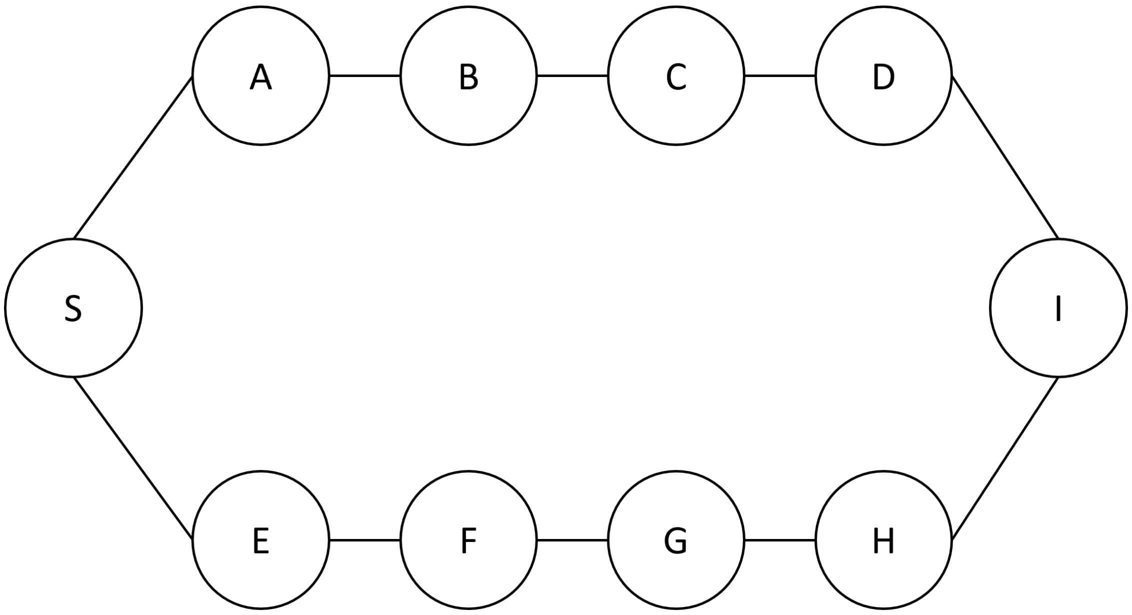


Figure 2.7: A simple structured environment. When an agent is traveling from S to I, its choice of which path to take is significant, as backtracking to reach the other path is very costly.

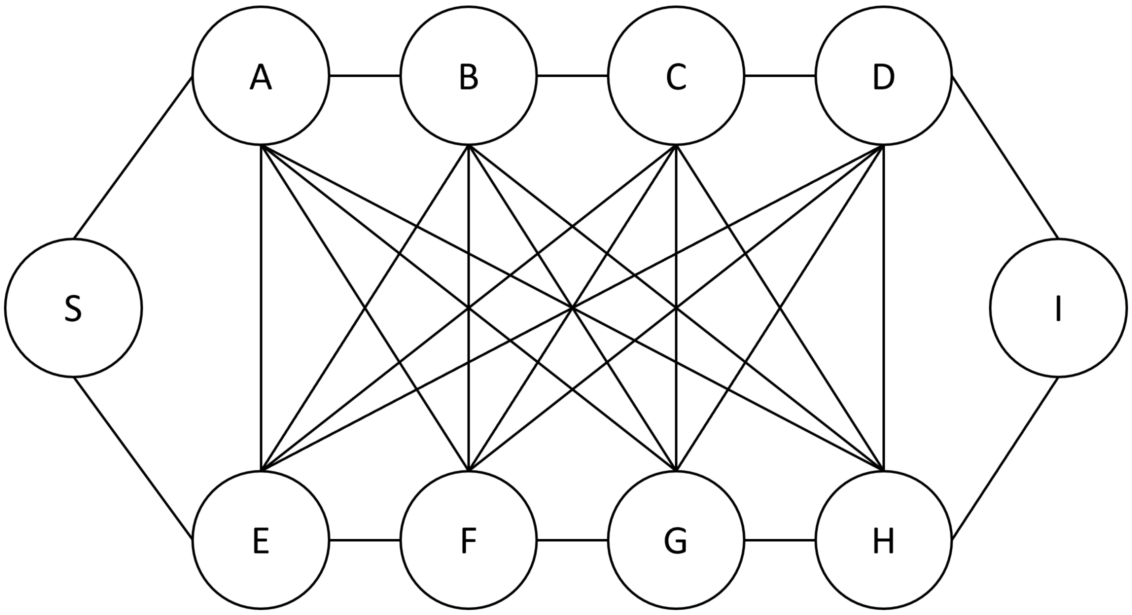


Figure 2.8: A less structured version of the environment shown in Figure **??**. The agent's initial choice of path when traveling from S to I is not as significant, due to the ability to travel directly from one path to the other without backtracking.

Chapter 3

The Task Graph Algorithm

Our first approach to exploiting the structure of the environment to our advantage is the TaskGraph algorithm[**?**, **?**]. The core concept of the Task Graph algorithm is to group actions or locations together into larger tasks based on the structure of the environment. We then plan over a graph of the available tasks, rather than over individual locations or actions.

We also present a new planning problem, which we call "Exploration Scheduling." Exploration Scheduling is motivated by the idea of a service robot performing scheduled user requests while also using spare time in between user requests to gather additional information. The information is then used to improve the scheduling of user requests in the future. The goal is to improve the reward received from user requests over time by gathering information without the robot's information gathering ever interfering with the its ability to perform user requests as optimally as possible.

In this chapter, we will first formalize the problem of Exploration Scheduling. We will then describe the Task Graph algorithm. Finally, we present experimental results demonstrating its effectiveness in a simulated Exploration Scheduling scenario.

3.1 Exploration Scheduling

The Exploration Scheduling problem is meant to represent a service robot gathering information in its spare time that it can use to improve its ability to schedule user requests.

3.1.1 Problem Description

We consider a robot that receives mandatory tasks, such as requests from users, that require the robot to travel to a specific location to perform them. The robot is given windows of time during which it must perform each task, and the robot's performance on the request depends on decisions it makes. The quality of the robot's performance can be predicted using features of the environment that the robot is capable of observing, and these features may be time-dependent. For example, the probability of an office robot's message deliveries succeeding depends on the probability of the recipient's door being open, which depends on the time of day. When the robot is not performing a user request, it is idle.

The goal of Exploration Scheduling is to have the robot make use of the idle time between user requests to gather information that will allow it to improve its services. The specific case we use is a robot learning users' schedules by observing the doors of their offices. Whenever the robot passes by an office, the robot can observe the status of their door, and infer whether or not they would be available to receive a user request at that time. Over time, the robot can use that information to build a model of the probability of each user being available at any given time, and use that model when scheduling user requests.

The goal is to maximize the reward received from user requests without the robot's information-gathering actions interfering with its ability to perform user requests as optimally as possible.

3.1.2 Formalization

We assume the robot operates in an environment represented by a graph G with nodes V representing relevant locations in the environment and edges E representing the paths that the robot can travel along between the nodes. The length of an edge represents the amount of time it takes the robot to travel from one node to the next. The robot has a map of the environment and is capable of determining its location, finding the shortest path to any other location, and traveling along that path without issue.

The Exploration Scheduling problem takes place in iterations. In each iteration, the robot is given an "Interval of Operation" T_{op} representing a continuous span of time, starting at time $t_s(T_{op})$ and ending at time $t_e(T_{op})$ over which the user requests and exploration take place.

The robot is also given a set of user requests Q. Each user request q_i has an interval of constraint $T_c(q_i)$ representing a span of during which it must be performed. To perform a user request q_i the robot must travel to the location v_i during the interval of constraint. For each location, there is a reward function $R_q(v_i, t)$ giving the probability of a request being successful when performed by the robot at location v_i a time time t. The robot receives a request reward of 1 if the request succeeds and 0 if it fails. The robot begins with no knowledge of R_q for any locations or times.

When the robot passes by a location v_i, it observes the location and receives information telling it the reward it word have received if it had performed a user request there. In other words, there is a probability of $R_q(v_i, t)$ that the robot observes a 1, otherwise it observes a 0. This observation does not directly give any reward — reward is only received when the robot performs an actual given user request during the given interval of constraint. However, the observation provides information that can help the robot learn the function $R_q(v_i, t)$ which it can use to better schedule user requests and increase the reward it receives in the future. We assume that repeated observations of the same room do not provide new information. To simulate this, we limit the robot to a single observation of each location per interval of operation.

The primary goal is to maximize the user request reward R_q received from user requests. During the intervals of time between user requests, the robot must choose where to explore to gather information that will most improve its ability to optimally schedule user requests. To this end, we assume that the robot is given an exploration reward function $R_e(v_i, t, S_i)$ representing the expected benefit of observing the location v_i at time t given S_i, the set of results and times of all past observations the robot has made at that location.

In each iteration, the goal of exploration scheduling is to produce a schedule consisting of the time at which it will perform each given user request, as well as the locations it will travel to in order to make observations in the spare time between user requests. There are two hard constraints that the schedule must meet:

1. The start time for each user request q_i must be within the interval of constraint $T_c(q_i)$.
2. There must be sufficient time to travel from the end location of each user request to the start location of the next one.

The first hard constraint is that the robot must attempt all user requests within the given interval of constraint. Creating a schedule that violates the interval of constraint or is missing a request is not an option. The second constraint ensures that completing the schedule is physically possible. We assume that meeting these hard constraints is always possible — the robot will not accept a user request if if it impossible to perform within the requested interval of constraint.

In addition to the two hard constraints, there are three soft constraints on the schedule the robot creates:

1. Maximize the total expected request reward R_q received for performing user requests.
2. Complete all user requests at the earliest time possible.
3. Maximize the total expected exploration reward R_e received for gathering information.

These soft constraints are in order of priority. For the purposes of this tnesis, we will treat these priorities as strict and not consider tradeoffs: maximizing the request reward is always more important than performing user requests early, which is always more important than any amount of exploration reward. The goal here is that the exploration the robot does in its spare time to make observations should never interfere with user requests. All user requests should always be performed as optimally as possible, and exploration should only be done if it does not prevent that.

Having the soft constraints be strictly in order of priority could result in situations where the robot's ability to explore is limited. In some scenarios, allowing the robot to schedule user requests suboptimally if it significantly improves the exploration that could be done could yield better long term user request reward, at the cost of the robot's exploration sometimes interfering with the optimal performance of user requests. In this chapter we choose to treat it as mandatory that user requests be scheduled as optimally as possible, but future work could explore the benefits of allowing the robot to schedule user requests suboptimally if the benefits to exploration are deemed high enough.

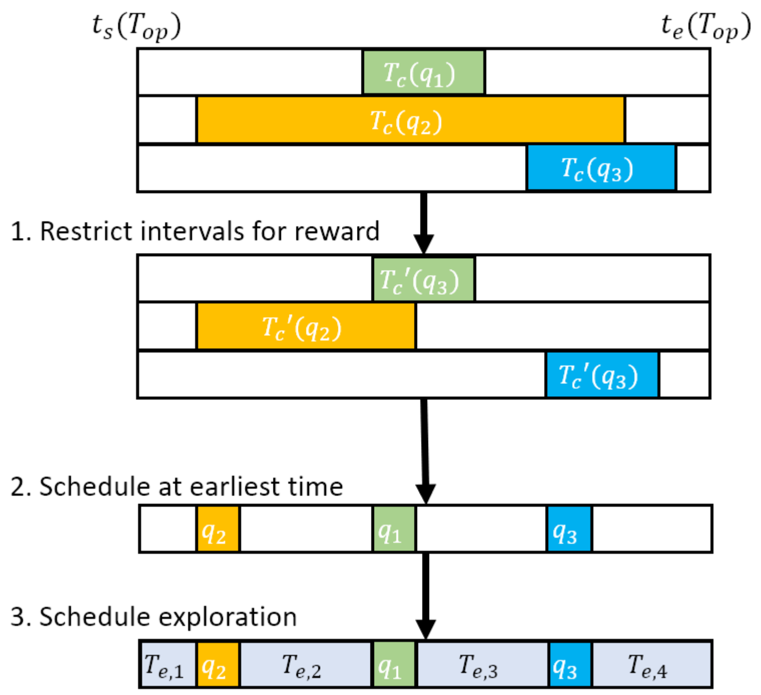


Figure 3.1: An illustration of the process of exploration scheduling. The robot begins with a set of user requests, each with an interval of constraint. It first restricts the intervals of constraint to maximize reward, then scheduled them at the earliest time possible within the restricted intervals. Finally, it schedules exploration in the spare time between the user requests.

3.2 The TaskGraph Algorithm

We now present the TaskGraph algorithm for finding solutions to the Exploration Scheduling problem. The core concept of the TaskGraph algorithm is to group locations in the environment into "exploration tasks" based on the environment's structure. The algorithm then creates a "Task Graph" in which the nodes are exploration tasks, rather than locations, and creates a schedule by finding a path through the task graph.

Overall, the process of solving the Exploration Scheduling problem has three steps:

1. For each user request q_i, find the sub-interval $T_c'(q,i)$ of the request's interval of constraint $T_c(q,i)$ during which the request will yield the maximum reward.
2. Schedule each user request at the earliest possible time in $T_c'(q_i)$.
3. Schedule exploration in between the user requests.

A diagram of this process is shown in Figure **??**. The first two steps are how we schedule user requests in order to meet the hard constraints and the first two soft constraints described in Section **??**. The third step is where we use the Task Graph algorithm to plan the exploration the robot will perform in its spare time.

3.2.1 Scheduling User Requests

The first step in the process is to restrict the intervals of constraint for each user request to the sub-interval that provides the optimal reward. For each user request q_i with interval of constraint $T_c(q_i)$, we choose a new restricted interval of constraint $T'_c(q_i)$:

$$T'_c(q_i) = \underset{T \subseteq T_c(q_i)}{\arg\max} \frac{1}{\text{Length}(T)} \int_{t_s(T)}^{t_e(T)} R_q(q_i, t) dt, \tag{3.1}$$

In some cases, there may be a single instantaneous time at which the reward is maximized. In these cases, we choose some minimal interval length minlength and use an interval of that size surrounding the optimal time. However, if the maximum in the request reward occurs as a plateau, rather than a single point, then the interval will be the entire plateau.

It is possible for a conflict to exist in which it is not possible to perform each request within the chosen optimal intervals in the same schedule. In these cases, we schedule the user requests according to the function:

$$T'_c(q_i) = \underset{T \subseteq T_c(q,i)}{\arg\max} \frac{1}{\text{length}(T)} (f * \frac{\text{Length}(T)}{\text{minlength}} + 1) \int_{t_s(T)}^{t_e(T)} R_q(q_i, t) dt, \tag{3.2}$$

where f is the flexibility parameter. The larger the flexibility parameter is, the more the function promotes choosing a larger interval of constraint over an optimal one. When $f = 0$, this is equivalent to using Equation **??**. In the case of a conflict, f is increased incrementally (we used $\delta = 0.1$ in our experiments) until a schedule can be found. The effect is that the restricted intervals of constraint are slowly widened until the conflict no longer exists.

The second step of the process is to schedule each user request as early as possible within the restricted intervals of constraint chosen in the first step. This is done using the existing algorithm described by Coltin et al., which solves a mixed integer program to determine the schedule that will accomplish all user requests as early as possible[**?**].

This process assumes that the maximum reward of each user request occurs over a single contiguous interval, which is the case in all of our experiments. If it is possible for user requests to have a reward function such that the reward is maximized over multiple disjoint intervals, the same process could be used by choosing only one interval, but it is possible a better method could be found.

3.2.2 Scheduling Exploration with the Task Graph Algorithm

Once the user requests are scheduled, we use the Task Graph algorithm to schedule the actions the robot will take to gather information in the spare time in between. With the user request times chosen, there are now fixed gaps in between the user requests, which we will call the Intervals of Exploration $\{T_{e,1}, \ldots, T_{e,n+1}\}$. We consider each Interval of Exploration independently, scheduling a set of exploration tasks for that interval and then merging the schedules for the individual intervals of exploration into the full schedule. This assumption simplifies the planning problem, but may result in the agent performing suboptimal exploration over the course of the whole interval of operation in some cases.

The core concept of the task graph algorithm is to create "exploration tasks" based on the structure of the environment. Each task consists of visiting multiple locations and making observations. The exploration tasks are scheduled by constructing what we call a "task graph," a directed graph with nodes representing the exploration tasks the robot can perform. The robot then finds a path through the task graph using a greedy hill-climbing algorithm, and schedules the tasks on the path.

Exploration Tasks

The Task Graph algorithm takes as input a set of exploration tasks. The goal in choosing exploration tasks is for each task to contain locations that we would generally expect to be part of the same optimal plan. In other words, we choose locations such that we believe it is unlikely the optimal plan contains some locations in an exploration task but not all of them.

The experiments we perform take place in environments based on real office buildings and consist of networks of hallways, as defined in Section **??**. To take advantage of the hallway-based nature of the environments, we chose exploration tasks that consist of traversing a single hallway. While there may be cases where the optimal places for the robot to visit during an interval of exploration include traveling only part way down a hallway and then backtracking, we expect that most often if the robot travels part way down a hallway, it is worth traversing the entire hallway. Each exploration task s_i consists of a start location $v_s(s_i)$ and end location $v_e(s_i)$ and represents traveling from the start location to the end location, observing each location along the way.

For each hallway, there are two exploration tasks: one that starts at one end of the hallway and travels to the other end, and another that goes in the opposite direction. Because the robot can only observe each location once per interval of operation, the two tasks consisting of traversing the same hallway are considered mutually exclusive: the robot cannot include both tasks in the same plan, because it cannot make observations and receive exploration reward from both tasks. Not that this does not mean the robot cannot retrace its steps — it can backtrack down a hallway that it has already traversed, it just does not consider that to be performing a task because it yields no reward.

Task Graph Construction

Once the exploration tasks are chosen, we construct the task graph. The task graph is a directed graph featuring one node for each exploration task the robot can perform. There is an edge from each task node s_i to each other task node s_j with length equal to $D(v_e(s_i), v_s(s_j))$, where $D(v_i, v_j)$ is the distance of the shortest path from v_i to v_j in the robot's environment (representing the time it takes for the robot to travel between two locations). The exception is nodes for tasks that are mutually exclusive with each other, which are not connected by an edge.

Each task node also has a duration, equal to the expected duration of performing the task. In the case of our hallway tasks, the duration of a task $L(s_i)$ is equal to $D(v_s(s_i), v_e(s_i))$, the distance from the start location of the task to the end location of the task (i.e. the time it takes for the robot to traverse the hallway). Additionally, each exploration task has a reward, equal to the total exploration reward $R_e(v_i, t, S_i)$ of all locations v_i in the task, averaged over all times within the interval of exploration.

In addition to the task nodes, the task graph has two nodes representing locations: one for the start location of the interval of exploration, and the other for the end location. There is an edge from the start location node v_s to each task node s_i with a length of $D(v_s, v_s(s_i))$. Similarly, there is an edge from each task node s_i to the end location node v_e with length equal to $D(v_e(s_i), v_e)$.

An example of a simple environment and a corresponding task graph is shown in Figures **??** and **??**, respectively.

Task Graph Hill Climbing

A path through the task graph represents a sequence of exploration tasks. The total cost of a path is the sum of all edge costs and task node durations along the path, and represents the expected time it would take to perform all of the tasks in order, including traveling from task to task. The total reward of a path is the sum of the rewards of all task nodes in the path. Any path from the start location node to the end location node with length less than the length of the interval of exploration that does not include mutually exclusive tasks is a valid schedule of exploration tasks for the interval of exploration. Thus, our goal in scheduling exploration tasks is to find the path that meets these requirements with the highest reward.

Our approach to this problem is a greedy hill-climbing algorithm. The task graph hill-climbing algorithm takes as input a task graph G_s (which includes a function $D(s_1, s_2)$ that gives the edge length from node s_1 to node s_2, and a duration function $L(s)$ that gives the duration of a task node), a start location v_s, an end location v_e, an exploration reward function $R_e(s_i)$ which gives a task's average reward over the interval of exploration, a maximum cost for the path t_{max}, and a dictionary of mutually exclusive nodes *mutexes*. It returns a path P through the graph from v_s to v_e with cost less than or equal to t_{max} that does not violate any of the constraints in *mutexes*. Pseudocode for the algorithm is shown in Algorithm **??**.

The algorithm works by iteratively adding tasks to the path until no more tasks can be added, at which point v_e is added to finish the path. Each iteration, each task in the graph is assigned a marginal reward (lines **??**-**??**). If the task is already in the path, mutually exclusive with a node already in the path, or there is not enough time left to perform the task and then travel to the end location, the task is assigned a marginal reward of $-\infty$ (lines **??**-**??**). Otherwise, the task is assigned a marginal reward equal the ratio of the task's reward to its cost, defined as the sum of the edge to the task node and the task's duration (line **??**). If no tasks have a positive marginal reward, then it is not possible to perform any more useful tasks in the time remaining, so the end location is added to the path and the path is returned (lines **??**-**??**). Otherwise, the task with the highest marginal reward is added to the end of the path, and the process is repeated (lines **??**-**??**).

Algorithm 1 The hill-climbing algorithm for finding a path of a given cost through the task graph.

```
 1: procedure TaskGraphHillClimb($G, v_s, v_e, R, t_{max}, mutexes$)
 2:     $p \leftarrow v_s$
 3:     last $\leftarrow v_s$
 4:     loop
 5:         for $s \in G$ do
 6:             if $D(\text{last}, s) + D(s, v_e) + L(s) > t_{max}$
                        $\vee\ s \in p$
                        $\vee$ mutexes$[s] \cap p \neq \emptyset$ then
 7:                 marginal($s$) $\leftarrow -\infty$
 8:             else
 9:                 marginal($s$) $\leftarrow \frac{R_e(s)}{D(\text{last},s)+L(s)}$
10:         if marginal($s$) $\leq 0\ \forall$ s $\in G$ then
11:             Append $v_e$ to $p$
12:             return $p$
13:         else
14:             last $\leftarrow \underset{s\in G}{\text{argmax}}(\text{marginal}[\text{task}])$
15:             Append last to $p$
```

Hill Climbing Example Consider the environment shown in Figure **??**. This environment has three intersections, L1, L2, and L3. There are six tasks, labeled as "T12", "T21", etc. in the map. Each task consists of the robot traversing from one end of a hallway to the other.

Now, for an example of the hill-climbing algorithm, consider our robot attempting to schedule a set of exploration tasks for an interval of exploration that begins at $t_{start}(T_e) = 0$ and ends at $t_{end}(T_e) = 25$, starting from L1 and ending at L3. Let each task have a reward equal to the number of rooms along the hallway, and let the two tasks for crossing each hallway be mutually exclusive. The task graph for this search is shown in Figure **??**.

The search begins at the L1 node. The algorithm computes the marginal reward of each task. Task 31 has a marginal reward of $-\infty$, because the total time to start from the current location, perform Task 31, and then travel to L3 would be 36, while the total time available for planning is 25. There is similarly not enough time to perform task 32. Of the remaining tasks, Task 12 has a marginal reward of 0.4, Task 21 has a marginal reward of 0.2, Task 13 has a marginal reward of 0.25, and Task 23 has a marginal reward of 0.22. So Task 12 is added to the path, and the search continues with Task 21 as the starting location.

Now, Task 21 is assigned a marginal reward of $-\infty$, because it is exclusive with Task 12. There is still not enough remaining time to perform Task 31 or 32, so those are out as well. Meanwhile, Task 13 gets a marginal reward of 0.18, while Task 23 gets a marginal reward of 0.31, so Task 23 is added to the path. Next the search continues from Task 23, but we find that all remaining tasks now have a marginal reward of $-\infty$: either there is insufficient time to complete them and still reach L3 (Tasks 13 and 31), or they are mutually exclusive

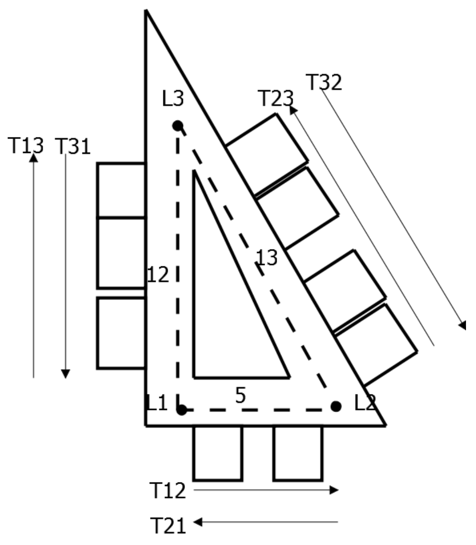


Figure 3.2: A simple environment consisting of three locations, L1, L2, and L3, and six tasks, T12, T21, T13, T31, T23, and T32. Each task corresponds to traversing a hallway in a certain direction.

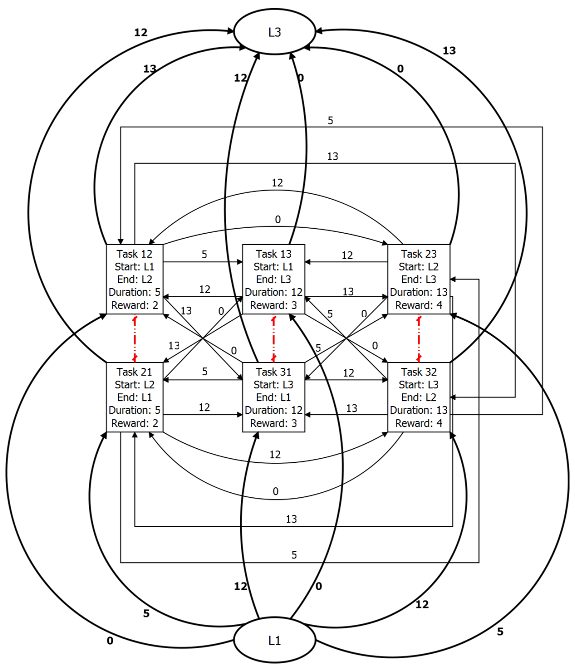


Figure 3.3: A task graph for the environment shown in Figure **??**, with a start location of L1 and end location of L3.

with a node that is already in the path (Tasks 21 and 32). So L3 is added to the path and the search is complete.

3.3 Evaluation

We tested the TaskGraph algorithm's ability to solve the Exploration Scheduling problem in simulation. Our simulation used an environment based on several floors of an actual office building in which service robots operate. In the simulation, the robot repeatedly received sets of user requests to perform services within a subset of a 30-minute interval of operation. The probability of requests being successful varied with time. The robot scheduled exploration tasks in between the user requests, and learned a model of the probabilities of requests being successful over time. It used the model to determine how to schedule future requests to maximize the odds of success.

Experimental Setup

Each simulated interval of operation was 30 minutes long. During that interval, the robot received four user requests, each to visit a randomly-chosen location within a random interval of constraint between 5 and 30 minutes long. Requests were performed (and succeeded or failed) instantaneously upon the robot's arrival at the given location. The requests were always tested to ensure that it was possible to perform all four within the given intervals of constraint. If not, a new set of requests was generated until a possible set was created.

The environment the robot operated in was based on the seventh through ninth floors of the Gates-Hillman Center at Carnegie Mellon University, with a total of 203 locations. A map of the seventh floor of the environment is shown in Figure **??**, and the same map can be seen in the form of a graph later in Figure **??**. The probability of a request at a location v_i being being successful at a time t was given by

$$R_q(v_i, t) = ae^{-\frac{(t-\mu)^2}{2\sigma^2}}, \tag{3.3}$$

where a, μ, and σ are randomly chosen parameters for each door: a was between 0.75 and 1.0, μ was between 0.0 and 30.0, and σ was between 2.0 and 17.0. Whenever the robot passed by a room, it observed either a success or a failure according to this probability. These observations were made whether the robot was deliberately observing a location as part of an exploration task or merely passing by while traveling from one location to another.

The robot stored the time and result of all observations it made. It did not know the form of the ground truth probability distributions for the requests, and inferred a probability at each minute by taking a weighted sample of observations at nearby times:

$$R_q(v_i, t) = \frac{0.5 + \displaystyle\sum_{o \in O_1(v_i)} e^{-\frac{(t(o)-t)^2}{50}}}{1.0 + \displaystyle\sum_{o \in O(v_i)} e^{-\frac{(t(o)-t)^2}{50}}}, \tag{3.4}$$

where t is the time, $O(v_i)$ is the set of all observations the robot has made of the location v_i, $O_1(v_i)$ is the set of observations the robot made of the location that gave a result of 1, $t(o)$ is the time at which the observation o was made, and only samples for which $|t(o) - t| \leq 10$ are

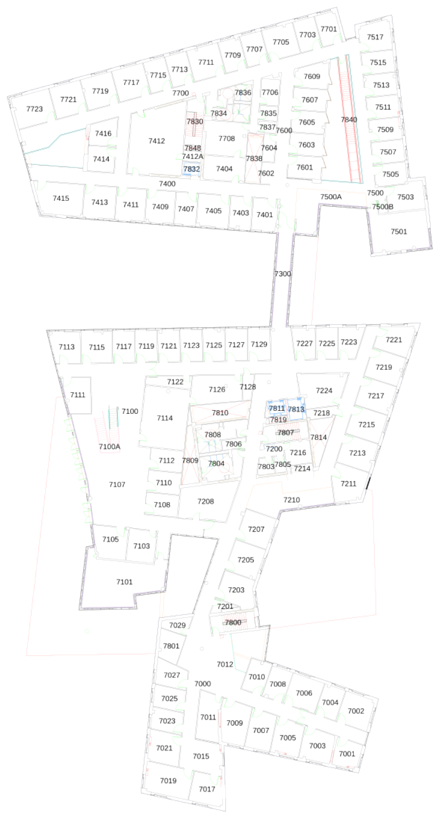


Figure 3.4: A map of the eighth floor of the Gates Hillman Center, one floor of the simulated environment used in the experiments. The robot travels through the hallways, and can travel between the floors using three elevators, labeled 7811, 7813, and 7832 in this map. The remaining numbers correspond to rooms at which the robot can perform user requests.

considered to speed up computations. The 0.5 in the numerator and 1.0 in the denominator are to reduce overfitting from small numbers of samples, biasing the probability very slightly towards 0.5.

As described in Section **??**, the exploration tasks available to the robot each consisted of traversing a single hallway in the environment. It would begin at one end of the hallway and travel to the other end of the hallway, observing all doors it passed along the way. There were two exploration tasks available per hallway, one starting at each end. Since the two tasks for each hallway yielded the same observations, they were considered mutually exclusive when performing the hill-climbing algorithm with the task graph.

Algorithms Tested

We compared three different algorithms for scheduling exploration tasks. Two were variations of task graph hill-climbing, one was a random exploration algorithm for comparison, and two were control models that did not do any exploration.

Uninformed Task Graph Hill-Climbing. The uninformed hill-climbing algorithm scheduled user requests, constructed task graphs for each interval of exploration, and carried out the hill-climbing algorithm as described in Section ??. It used a uniform reward function, assigning all tasks a constant reward of 1.0. This reward promoted gathering as many observations as possible, regardless of what they were or what data the robot already had.

Informed Task Graph Hill-Climbing. Like the uninformed hill-climbing algorithm, the informed hill-climbing algorithm scheduled requests and exploration tasks using the Task Graph algorithm described in Section ??. However, this algorithm used an informed exploration reward for observing a location based on the density of the observations the robot had of that location at that time, given by

$$R_e(v_i, t) = \frac{1}{\left(1 + \sum_{o \in O(v_i)} e^{-\frac{(t(o)-t)^2}{50}}\right)^2}. \tag{3.5}$$

The exploration reward for observing all locations on a hallway was the sum of the rewards for observing each of the locations. This reward function gave a higher reward to observations of hallways and times during which the robot had less data. As the number of observations the robot had of a hallway at a given time increased, the reward of exploring that hallway decreased dramatically. Thus, the informed hill-climbing algorithm sought to achieve an even distribution of observations over all locations and times.

Random Exploration. The random exploration algorithm began by restricting the intervals of constraint for the user requests as described in Section ??. Once the restricted windows were chosen, it randomly chose between four and seven exploration tasks. It then used the same mixed integer program that was used when choosing the starting times of tasks in Section ?? to create a schedule featuring both the user requests (with the restricted intervals of constraint chosen) and the chosen exploration tasks (with intervals of constraint spanning the entire interval of operation).

If a schedule was found, it would execute that schedule. If no schedule could be created, the chosen exploration tasks were rejected and only the user requests were executed using the mixed integer program.

No Exploration. With this algorithm, the robot did not schedule any exploration tasks at all. Instead, it simply ran the schedule of user requests created as described in Section ??. It still learned a model, but only from observations of the doors it passed by while traveling between user requests.

No Learning. The final model tested in each trial was one that learned nothing at all. In this case, the robot always assumed that all requests had a probability of 0.5 of succeeding at all times. This resulted in all user requests being scheduled as early as possible.

Results

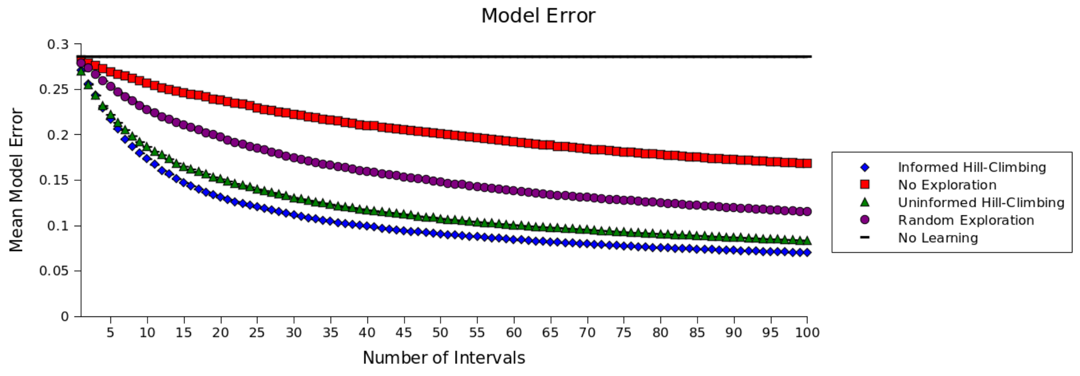


Figure 3.5: The average error of the model learned by the different algorithms over time in a simulation of the Exploration Scheduling problem.

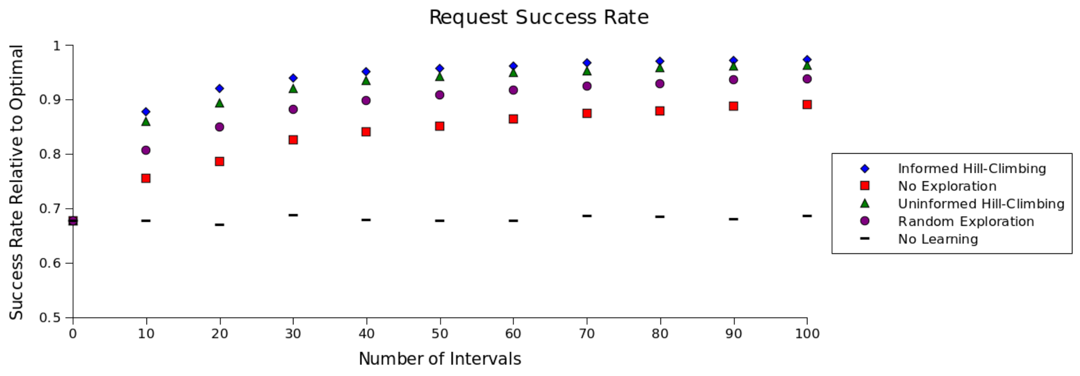


Figure 3.6: The average expected success rate of the model when delivering messages, relative to the optimal expected success rate, for different algorithms in a simulation of the Exploration Scheduling problem.

We ran 30 trials, each consisting of 100 30-minute intervals, with the robot accumulating more knowledge of the environment with each interval. We evaluated the algorithms using three different metrics.

Model Error The model error was computed by taking the mean difference of each model's learned probability and the true simulated probability of a user request being successful at each location at each minute. It represents how accurate a model each algorithm was able to create from the observations it gathered.

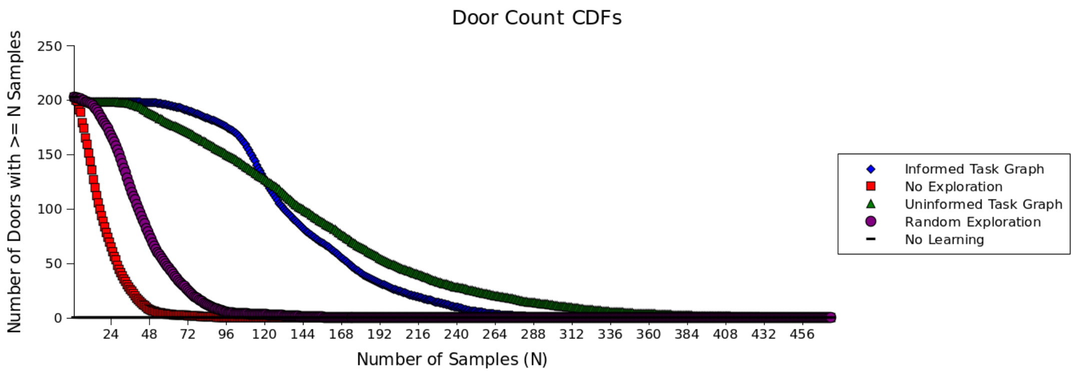


Figure 3.7: The number of locations at which each model had made at least a given number of observations after 100 intervals for different algorithms in a simulation of the Exploration Scheduling problem. The area under each curve is the total number of observations obtained by the algorithm.

Figure **??** shows the average error of the model learned by each of algorithm after each interval. The robot learned a much more accurate model of the environment when using exploration tasks than without them. The best model was learned when using the Task Graph algorithm with an informed reward, which learned a model that was more accurate than the uninformed version of the algorithm by more than one standard deviation within fifteen iterations. The gap between the model errors continued to grow with further exploration. The uninformed version of the algorithm, meanwhile, had an average model error less than that of the random exploration algorithm by more than one standard deviation within five iterations. This demonstrates that hill-climbing through a task graph allows the robot to learn a strong model of the environment, particularly when using an informed reward function that encourages the robot to explore the areas from which it has the least data.

User Request Success Rate Every ten iterations, each model was tested to obtain a measure of how well the model translated into an effective policy when carrying out user requests. For this test, the model was given 500 requests, one at a time, to visit randomly-chosen rooms in randomly-chosen constraint intervals between 5 and 30 minutes long. It chose the time in the interval at which it believed the delivery to have the highest chance of success, and received a reward equal to the actual probability of the delivery succeeding at that time. These rewards were normalized relative to the optimal reward the robot could have received if it chose the best possible time for every request.

Figure **??** shows the results of this test. Although the differences between the rewards the algorithms earned are less significant than the differences in their model errors, we can still see that, on average, a better model translated to a higher reward, with algorithms ranking in the same order in terms of performance. Thus, a better model of the environment translates to a better policy when carrying out user requests. The informed task graph hill-climbing algorithm was able to achieve over 97% of the optimal reward after 100 iterations, showing that the model learned was extremely effective for choosing a strong policy.

Cumulative Distribution of Observations Figure **??** gives some insight into why the algorithms ranked the way they did. It shows the number of locations for which the robot had made at least a given number of observations after 100 intervals. We can see that both Task Graph algorithms consistently acquired considerably more data than the other methods. Curiously, the uninformed task graph algorithm appears to have acquired a more even distribution of data than the informed algorithm, even though the informed algorithm was designed for acquiring a more even distribution of data.

3.4 Conclusion

In this chapter we presented the Task Graph algorithm for planning for a robot in a structured environment. The Task Graph algorithm is based around the idea of grouping locations in the environment together into tasks based on the structure of the environment, and planning over the tasks instead of individual locations.

We also presented the Exploration Scheduling problem, in which a service robot uses its spare time in between user requests to gather information that can be used to improve the services it provides. We tested the Task Graph algorithm in a simulated version of the Exploration Scheduling problem and found that an informed Task Graph algorithm was an effective way for the robot to gather information and learn a model of the environment, improving the reward received from user requests over time.

One notable feature of the Exploration Scheduling problem that we did not discuss in depth in this chapter is the fact that the robot could observe any location it passed, whether or not that location was explicitly included in the plan the robot created or was just a location it passed by traveling between two tasks in its schedule. We will explore this concept further, and a method for a planning algorithm can take the locations the robot passes into account when planning, in Chapter **??**.

Chapter 4

The RegionPlan Algorithm

In this chapter we contribute RegionPlan, our second approach to exploiting the structure of an agent's environment to improve its ability to plan. The concept of RegionPlan is to divide the environment into regions based on its structure, and create plans only for individual regions, rather than for the entire environment.

4.1 Problem Description

The problem we focus on in this chapter is a variant of the Orienteering Problem (OP) described in Section **??**. We call this variant the Orienteering Problem with Task Times (OP-TT). An agent operates in an environment defined by a graph G, with nodes V representing locations where it can perform tasks and edges E representing paths connecting them with lengths representing the time it takes to traveling along those paths. Each node v_i has a reward $R(v_i)$ representing the value of performing a task there. To actually perform a task and receive a reward, the agent must stop at a node for a time of t_{task}, and the agent cannot perform a task more than once at any given location.

The agent is given a start location v_s, and end location v_e, and a length limit t_{max}. The goal is to find a plan $p = \{v_s, v_0, \ldots, v_n, v_e\}$ consisting of a sequence of nodes, starting at the start location and ending at the end location, that maximizes the reward of performing a task at each location in the plan without the total length of the plan (including both travel time and time spent performing the task) exceeding t_{max}.

The OP-TT differs from the original OP because of the task times. In the original OP, the agent automatically receives reward from any node it visits without having to spend any time there. The OP-TT can be turned into an equivalent OP by turning the environment into a fully connected one in which the length of each edge is the shortest path between two nodes plus t_{task}. In existing research the OP-TT would be solved by changing the environment in this manner, and as such the OP-TT itself has not been studied as its own variant of the orienteering problem to our knowledge, although some heuristics used to solve the orienteering problem could be applied to the it with few or no modifications. However, because converting an instance of the OP-TT into an instance of the ordinary OP requires changing the environment into a fully connected one, the structure of the environment is lost in the process.

Our goal is to create heuristics designed for structured environments with hallways or bottlenecks. In order to do this, we must treat the OP-TT as its own variant of the OP, rather than converting it to an instance of the OP. Our goal is to show that doing so allows us to create heuristics that exploit the structure of the environment in order to yield more reward than heuristics designed without structured environments in mind.

4.2 The RegionPlan Algorithm

The RegionPlan algorithm is a planning algorithm we created that exploits the structure of the environment by dividing it into regions and only planning over each region, rather than over the environment as a whole. It consists of three main steps:

1. Identify subsets of nodes in the environment ("regions") that may contain the optimal plan based on the environment's structure.

2. Create a plan for each region.

3. Choose a plan to execute from the plans created in step two.

4.2.1 The Single-Plan RegionPlan Algorithm

We first present a version of RegionPlan called the Single-Plan RegionPlan algorithm (SP-RegionPlan). This algorithm only chooses a single region's plan to execute and does not consider the possibility of the optimal plan including multiple regions.

Pseudocode for the Single-Plan RegionPlan algorithm is shown in Algorithm **??**. The algorithm takes as input an environment G, a set of regions in the environment g, a start and end locations v_s and v_e, a time limit t_{max}, a reward function $R(v)$ for each node $v_i \in G$, a task duration t_{task}, and a region evaluation function *RgnEval*. It returns a plan p that starts at location v_s, ends at location v_e, and can be performed in time t_{max}.

RgnEval itself is a function that takes as input a subset of the nodes in the environment g, start and end locations v_s and v_e, a time limit t_{max}, and a reward function $R(v)$ for each node $v_i \in g$, and a task duration t_{task}. The goal of RgnEval is to find and return the plan p_g that starts at location v_s, ends at location v_e, can be performed in time t_{max}, and only receives reward from locations in g that maximizes the reward received $R(p_g)$.

The core idea of the algorithm is that it uses *RgnEval* to find a plan for each region, then chooses the best plan. It begins by initializing the best plan p_{best} to be the plan $[v_s, v_e]$ — that is, the plan consisting of traveling directly to the end location while performing no services — and sets the best reward r_{best} to 0 accordingly (lines **??** through **??**). It then iterates over all of the regions in g. For each region, it generates a plan for that region user *RgnEval* (line **??**). If the generated plan has higher reward than the previous best reward r_{best}, then p_{best} and r_{best} are updated to be the new plan and its reward, respectively (lines

?? through **??**). Finally, after it has iterated through all of the regions, it returns the best plan found (line **??**).

Algorithm 2 The Single-Plan RegionPlan Algorithm

```
1: procedure SP-RegionPlan($G, g, v_s, v_e, t_{max}, t_{task}, R, RgnEval$)
2:     $p_{best} \leftarrow [v_s, v_e]$
3:     $r_{best} \leftarrow 0$
4:     for $g_i \in g$ do
5:         $p_g \leftarrow RgnEval(g_i, v_s, v_e, t_{max}, t_{task}, R)$
6:         if $R(p_g) > r_{best}$ then
7:             $r_{best} \leftarrow R(p_g)$
8:             $P_{best} \leftarrow P$
9:     return $P_{best}$
```

4.2.2 Selecting Regions

The primary challenge of region-based planning is choosing the set of regions to use (the input g in the RegionPlan algorithm). The goal in selecting regions is to identify subsets of the environment that may contain the optimal plan, based on the structure of the environment.

The two environment structural features that we believe RegionPlan is best suited for are environments divided into sections by a bottleneck (as described in Section **??**) or sections in which there is a choice between non-overlapping routes between points in the environment (as described in Section **??**). In the case of environments with bottlenecks, having one region for each section of the environment divided by a bottleneck would be effective. In a case where the agent has a number of discrete, non-overlapping paths to choose from, each path would be a region.

For example, consider a robot operating in a multi-story office building. The robot can travel throughout each floor of the environment relatively effectively, while the elevator acts as a bottleneck between floors. In this case, we might expect the optimal plan to minimize the number of elevator trips and focus primarily on gathering reward from one floor of the building. Thus, having each floor of the building be a separate region would be an intuitive choice.

There is no requirement for what subsets of nodes can or can't be part of a region. It is possible for a single node to be part of multiple regions, or part of no region at all.

4.2.3 Region Evaluation

Another important element of RegionPlan is the region evaluation function *RgnEval*. As discussed in Section **??**, the region evaluation function takes as input a set of nodes g, a set of services S, start and end locations v_s and v_e, a time limit t_{max}, a task time t_{task}, and a reward function R. It returns a plan p that starts at location v_e, ends at location v_e, and can be performed in time t_{max}, and consists only of actions on locations in g. In other words, the region evaluation function is a function that solves the Orienteering Problem with Task Times described in Section **??**. It could even be another RegionPlan algorithm.

This is a very notable property of the RegionPlan algorithm: any algorithm that solves the OP-TT can be used as the *RgnEval* input. This allows the *RgnEval* algorithm to be chosen according to the properties of the individual regions, such as their structure. The same could also easily be true for any other planning algorithm for any other planning problem: The RgnPlan algorithm isn't necessarily limited to the OP-TT, and could apply to other variants of the OP as well.

4.2.4 The Greedy Algorithm

In the experiments performed in this chapter, each of the regions used consists of a single hallway. In these environments, we choose to use a simple greedy algorithm as our region evaluation function *RgnEval*.

Before we describe the greedy algorithm used in our experiments, we must first define two terms. The marginal reward of a location is the ratio of the reward that can be received from performing a task at that location to the cost of traveling there and collecting reward. That is,

$$R_{marginal}(v) = \frac{1}{D(v_c, v) + t_{task}} R(v), \tag{4.1}$$

where v_c is the agent's current location and $D(v_c, v)$ is the distance from v_c to v. An action is "reachable in time" if there is sufficient time for the agent to travel to that location, receive reward, and then travel to the end location in time. In other words,

$$v \text{ is reachable in time from } v_c \text{ in } t \text{ if } D(v_c, v) + t_{task} + D(v, v_e) \leq t.$$

The set $V_{reach}(v, t)$ is the set of all actions reachable in time from v in t.

With those terms defined, we can present the greedy algorithm used in our experiments. Pseudocode for the greedy algorithm is shown in Algorithm **??**. It initializes by creating a plan that consists only of the start location, setting the current location to the start location, and the current time to 0 (lines **??** and **??**). It then chooses the reachable location with the highest marginal reward that is not already in the plan and appends it to the plan (lines **??** and **??**). The current time and location are updated to match the time and location after traveling to that location and performing a task (lines **??** and **??**). This process is repeated until there are no more reachable actions (line **??**). Finally, it adds the end location to the plan and returns the plan (lines **??** and **??**).

This algorithm is simple and fast, but it does not plan ahead in any way, instead making the assumption that the optimal path is the one that begins with the highest marginal reward. The goal of our experiments is to show that by applying the concept of region-based planning to this algorithm, we can improve the quality of the plan created and avoid certain pitfalls of the algorithm, demonstrating the benefits of the RegionPlan algorithm.

Algorithm 3 The Greedy Planning Algorithm

```
1: procedure GreedyPlanner($g, v_s, v_e, t_{max}, R, t_{task}$)
2:     $p \leftarrow [v_s]$
3:     $v_c \leftarrow v_s$, $t \leftarrow 0$
4:     while $V_{reach}(v_c, t_{max} - t) - p$ is not empty do
5:         $v_{next} = \underset{v \in V_{reach}(v_c, t_{max}-t)-p}{\text{argmax}} R_{marginal}(v)$
6:         Append $v_{next}$ to $P$
7:         $t \leftarrow t + D(v_c, v_{next}) + t_{task}$
8:         $v_c \leftarrow v_{next})$
9:     Append $v_e$ to $p$
10:    Return $p$
```

4.2.5 The Region-Switching RegionPlan Algorithm

While the goal of the Single-Plan RegionPlan algorithm is that we select regions such that the optimal plan is contained within a single region, naturally that will not always happen. To address this, we propose that the algorithm use a heuristic to determine whether it should switch from executing the chosen plan to executing a different plan, rather than always carrying out the best plan found in any region to completion.

We now contribute the Region-Switching RegionPlan algorithm (RS-RegionPlan), a version of the RegionPlan algorithm that is not limited to only executing a single plan from a single region like the Single-Plan RegionPlan algorithm, but rather uses a heuristic to determine if it should switch regions partway through executing one of the computed region plans.

Pseudocode for RS-RegionPlan is shown in Algorithm **??**. It takes the same inputs as SP-RegionPlan: an environment G, a set of regions in the environment g, a start and end locations v_s and v_e, a time limit t_{max}, a reward function $R(v)$ for each node $v_i \in G$, a task duration t_{task} representing the time the agent must spend at a location to receive reward, and a region evaluation function *RgnEval*. Like SP-RegionPlan, it begins by computing a plan for each region and selecting the plan with the highest reward (Lines **??** through **??**). Rather than only storing the best plan found, however, it stores all plans in a list P_{all} (line **??**).

Once the plan has been found for all regions, RS-RegionPlan does not simply return the best plan like SP-RegionPlan. Instead, it begins stepping through that plan. After each step in the plan, it uses a heuristic to compare the remainder of the plan with a portion of each of the other plans.

Let $p_{current}$ be the plan the robot is currently following. Let p_{remain} be the remaining locations in $p_{current}$ that haven't been travelled to yet, and t_{remain} be the time required to complete plan p_{remain} starting from the agent's current position (lines **??** and **??**). For each other computed plan p in P_{all}, determine the portion of that plan $p_{partial}$ that can be completed from the robot's current location in time t_{remain}, including traveling to v_e (line

??). If there is any partial plan that has a better reward than p_{remain}, then the algorithm switches from continuing to executed the current plan to executing the new best partial plan instead (lines **??** through **??**). The next location in the chosen plan, whether it is the same one that was previously being followed or a different plan, is then added to the final plan (line **??**). This process repeats until there is not enough time to add the first node of any remaining plan to the final plan.

Algorithm 4 The Region-Switching RegionPlan Algorithm

```
 1: procedure RS-RegionPlan($G, g, v_s, v_e, t_{max}, t_{task}, R, RgnEval$)
 2:     $p_{best} \leftarrow [v_s, v_e]$
 3:     $P_{all} \leftarrow []$
 4:     $r_{best} \leftarrow 0$
 5:     for $g_i \in g$ do
 6:         $p_g \leftarrow RegEval(g_i, v_s, v_e, t_{max}, t_{task}, R)$
 7:         Append $p_g$ to $P_{all}$
 8:         if $R(p_g) > r_{best}$ then
 9:             $r_{best} \leftarrow R(p_g)$
10:             $p_{best} \leftarrow p_g$
11:     $p_{final} \leftarrow [v_s]$
12:     Remove $p_{best}$ from $P_{all}$
13:     $p_{current} \leftarrow p_{best}$
14:     repeat
15:         $p_{remain} \leftarrow$ remaining steps in $p_{current}$
16:         $r_{remain} \leftarrow R(p_{remain})$
17:         $t_{remain} \leftarrow L(p_{remain})$
18:         $r_{best} \leftarrow r_{remain}$
19:         $p_{best} \leftarrow p_{remain}$
20:         for $p \in P_{all}$ do
21:             $p_{partial} \leftarrow$ portion of $p$ that can be completed in $t_{remain}$
22:             if $R(p_{partial}) > r_{best}$ then
23:                 $r_{best} \leftarrow R(p_{partial})$
24:                 $p_{best} \leftarrow p_{partial}$
25:         $p_{current} \leftarrow p_{best}$
26:         Append first node of $p_{next}$ to $p_{final}$
27:         if $p_{current}$ was changed then
28:             Remove $p_{current}$ from $P_{all}$
29:     until No node can be added in time $t_{remain}$
30:     Append $v_e$ to $p_{final}$
31:     return $p_{final}$
```

4.2.6 Example

Consider the environment shown in figure **??**. Let the center node labeled S be both the start location and end location. Assume $t_{task} = 0$, all edges have length 1, $t_{max} = 6$, and the nodes have the rewards shown. We will compare the greedy algorithm described in Section **??** with the RS-RegionPlan algorithm.

The greedy algorithm begins at S and computes the marginal reward of each node in the environment, equal to the reward of that node divided by its distance from the center. It finds that the node with the highest marginal reward is $R_{marginal}(J) = 9$, so it appends J to the plan, leaving a remaining time of 5. It now computes the marginal reward of each remaining node in the environment based on its distance from J, except for nodes I, F, and C, because it it cannot travel to those and return to S within a time limit of 5. The node with the highest marginal reward is K, with a marginal reward of 3, so it is appended to the plan, leaving a remaining time of 4. The only remaining nodes it has time to travel to are L, G, D, and A. Out of those, L has the highest marginal reward of 3, so it is appended to the plan. There is no time to visit any more nodes, so S is appended to the plan. The final time is $[S, J, K, L, S]$, with a reward of 15.

Now consider the RS-RegionPlan algorithm, using the four hallways extending out from S as the regions and the same greedy algorithm as the *RgnEval* function. The algorithm starts by computing a plan for each region. In clockwise order starting from the hallway on the right, those plans are $[S, A, B, C, S]$ with a reward of 19, $[S, F, E, D, S]$ with a reward of 12, $[S, G, H, I, S]$ with a reward of 8, and $[S, J, K, L, S]$ with a reward of 15.

The algorithm selects the plan $p = [S, A, B, C, S]$ because it has the highest marginal reward, and appends the first node, A, to the plan. The remaining time to finish p is 5, so it compares the reward of finishing p to the reward of completing as much of a different plan as possible in a time of 5. The reward of finishing p is 14. For the left hallway it can receive a reward of 6 and for the top hallway it can receive a reward of 12, while performing any steps of the plan for the bottom hallway is impossible because it starts with F and there isn't enough time to travel to F and return to S within a time of 5. The best option is continuing with p, so the algorithm adds B to the plan and compares the plans again. Finishing plan p now yields a reward of 6 in a time of 4, while spending a time of 4 executing the start of a plan in the left or top hallway yields a reward of 5 or 9, respectively. Since switching to the top plan gives more reward than finishing plan p, the algorithm switches plans and adds J to the plan instead of C. At this point, there is no time left to include any more nodes in the plan, so it adds the end location S. The final result is the plan $[S, A, B, J, S]$ with a reward of 22.

4.2.7 Efficiency of Region-Based Planning

One of the benefits of the RegionPlan algorithm is that it is potentially very fast. In fact, if no region switching is used and the total size of all regions is less than or equal to the size of the environment (e.g. there are no overlapping regions), then for any algorithm A that solves the OP-TT in worse than linear time, the SP-RegionPlan algorithm using A as the

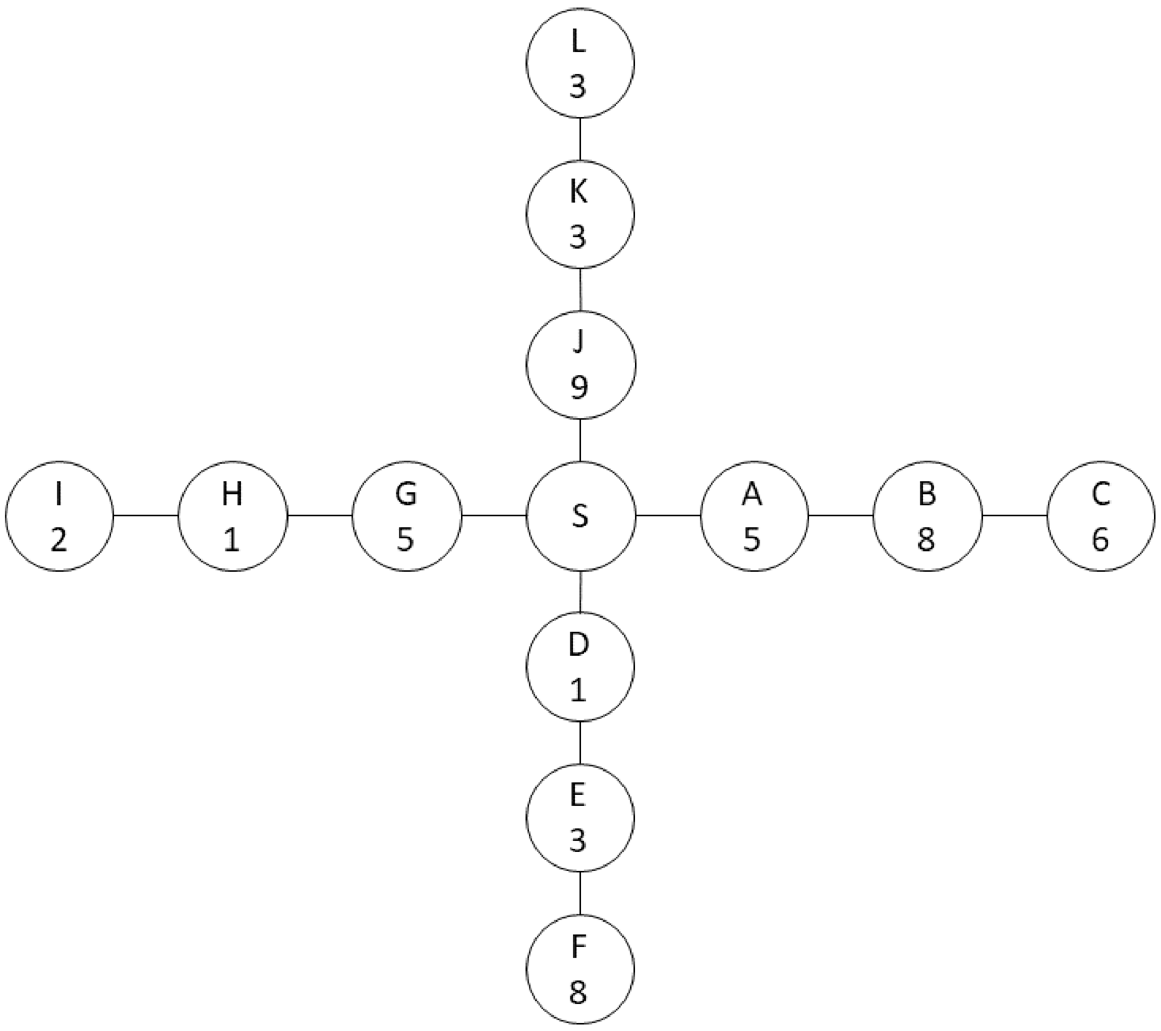


Figure 4.1: An environment used for an example of the RS-RegionPlan algorithm. The nodes are labeled with their names and rewards.

.

region evaluation function will find a solution faster.

To show this, let A be an algorithm that solves the OP-TT and is $O(f(n))$, where n is the number of nodes in the environment and $f(n)$ is slower than linear. Let A_r be an implementation of SP-RegionPlan that uses A as the region evaluation function $RgnEval$. Let there be c regions, each containing m nodes. In that case, A_r is $O(c * f(m))$. If $f(m)$ is slower than linear, and $m \leq \frac{n}{c}$ (which will be the case if the regions do not overlap), we can clearly see that $c * f(m) < f(n)$, so the RegionPlan variant of the algorithm will perform faster.

This math specifically applies to SP-RegionPlan, not necessarily RS-Region plan, which must spend additional time carrying out the region-switching heuristic. However, the region-switching heuristic can potentially scale better with the environment size than the original planning algorithm, particularly because it does not need to recompute new plans for any region. It only needs to recompute the rewards of partial plans, which can be done quickly.

4.2.8 Flexibility of the RegionPlan Algorithm

An important feature of the RegionPlan algorithms is the flexibility created by the *RgnEval* parameter. *RgnEval* can, effectively, be any algorithm Orienteering Problem with Task Times.

One of the core goals of this thesis as a whole is to demonstrate the benefits of exploiting the structure of the environment when planning. Depending on the structure of the environment and other parameters of the planning problem being solved, the best algorithm to solve it could vary.

While the RegionPlan algorithms are designed around taking advantage of the larger structure of the environment that lends itself well to being divided into regions, the individual regions could have their own structures. The *RgnEval* function can be chosen to suit those structures. As we have discussed, one possible application of the RegionPlan algorithms is a multi-story office building environment, using each floor as a region. But the individual floors of office buildings are also typically structured, usually consisting of networks of hallways. In such a scenario, *RgnEval* could be an algorithm that takes advantage of that hallway-based structure, such as the TaskGraph algorithm used in Chapter **??** or the Route-MLVNS algorithm that we will present in Chapter **??**.

RegionPlan is not even solely limited to the OP-TT. It could be used for other variants of the orienteering problem, and likely other planning problems, as well. The three-step process described in Section **??** could be applied to a wide variety of planning problems. In many cases, the SP-RegionPlan and RS-RegionPlan algorithms could be applied too, with minimal modifications to the the algorithms, simply by choosing an appropriate *RgnEval* function.

In this particular chapter, we focus on demonstrating the value of RS-RegionPlan for the OP-TT in specific environment structures, using a simple greedy *RgnEval* function. But we believe it has the potential to be a flexible framework with a variety of applications.

4.3 Evaluation of RegionPlan

In order to demonstrate the benefits of RS-RegionPlan, we compared the performance of the greedy algorithm described in Section **??** with an implementation of RS-RegionPlan using the same greedy algorithm as the region evaluation function. Our tests were performed in a simulation we created, using two different environment structures and two different types of reward distributions. The time to perform a task t_{task} was 0.

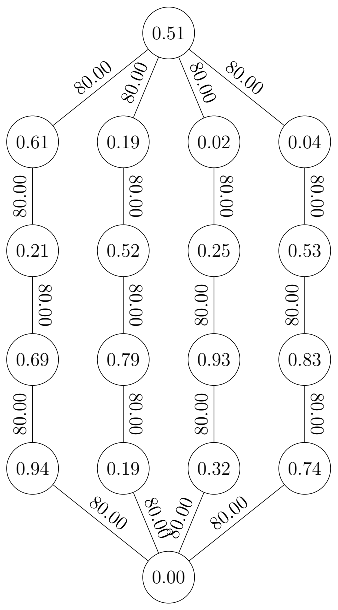


Figure 4.2: An example of a split path environment with four hallways and four rooms per hallway and randomly assigned rewards.

4.3.1 Environment Structures

We tested the algorithms in two environment structures: The Split Path and the Looped Asterisk.

The Split Path

The split path represents a situation in which an agent must travel from one location to another and has a variety of paths it can take, and must choose which path to travel in order to maximize the reward it acquires along the way. It features "start" and "end" locations, connected by a series of hallways. The hallways are all equal in length, and each hallway features a number of locations evenly spaced along it. An example of a split path environment is shown in Figure **??**. For the RS-RegionPlan algorithm, each hallway is a region. In all split path scenarios tested, the robot began at the start location and had to finish its plan at the end location.

The Looped Asterisk

The looped asterisk represents an environment that features a number of sections connected by a central bottleneck node. An example of this structure in real life is a multi-story building where a robot must use an elevator to travel between floors. The looped asterisk features a "center" location with a set of hallways radiating out from it. Each hallway loops back so that the first and last locations along the hallway are connected to the center. The distance between each pair of adjacent locations in the hallway, as well as from the first and

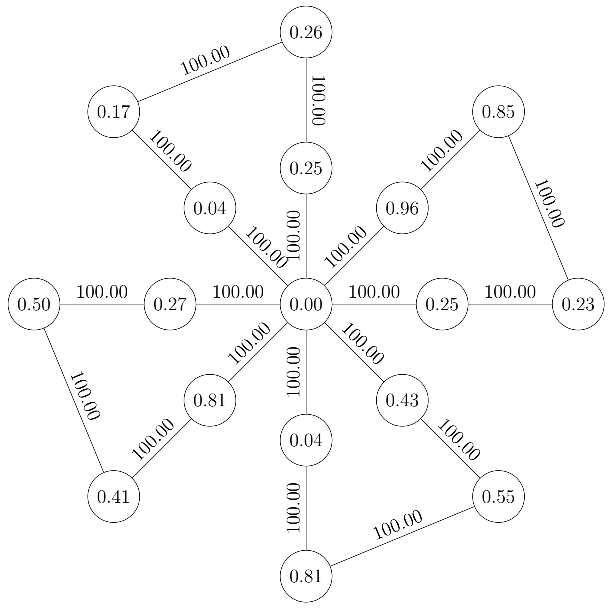


Figure 4.3: An example of a looped asterisk environment with four hallways and four rooms per hallway and randomly assigned rewards.

last locations to the center, is equal. Each hallway has the same total length. Figure **??** shows an example of such a map. As with the split path environment, for RS-RegionPlan each hallway was a separate region. In all looped asterisk scenarios tested, the start and end location were both the center.

4.3.2 Reward Distributions

In addition to the two environment structures, we tested two different ways of assigning rewards to the locations in the environments.

Random

In the first distribution, we randomly assign a reward between 1.0 and 0.0 to each location in the environment. This tests how well the two algorithms can take advantage of the structure of the environment without any patterns in the reward distribution.

Bait

The second distribution involved putting a "bait" location in each region. For each region in the environment, a "bait distance" d_{bait} was chosen randomly between a given d_{min} and d_{max}. All locations within that distance of the start location ("start" in a split path map, "center" in a looped asterisk map) were assigned a reward of 0.0. The closest location not within that distance (chosen randomly if multiple locations were equally close) was the "bait," and was assigned a reward of 1.0.

A mean reward of $\bar{R} = \frac{d_{bait} - d_{min}}{d_{max} - d_{min}}$ was assigned to that hallway . Every location in the hallway closer to the center than d_{min} was given a reward of 0.0. Every location farther than d_{min} was given a reward drawn from a normal distribution with mean $\bar{R}$ and standard deviation σ_R. This created a map in which the average reward of locations in a hallway scaled inversely with the highest marginal reward of any location in the hallway.

4.3.3 Results

We performed several tests to evaluate the Region-Switching RegionPlan algorithm. Each test compared two algorithms: the greedy algorithm described in Section **??**, and an implementation of the RS-RegionPlan algorithm described in Section **??** that used the same greedy algorithm as the region evaluation function. The tests were performed in all of the structures and reward distributions described in section **??**. The time limit t_{max} was 500 in all experiments, slightly more time than necessary to perform tasks at every location in a single region. In all cases, results are averaged over 100 trials.

Split Path Results

First, we tested the two algorithms in a split path environment. We varied the number of hallways, and each hallway featured thirty rooms. The cost to travel from the start location to the end location along any of the hallways was 400. Figure **??** shows how much reward each algorithm received with different numbers of hallways when rewards were randomly distributed. With only three hallways, the two algorithms' performances were roughly equal, although RS-RegionPlan performed slightly better on average. As the number of hallways increased, the greedy algorithm's performance stayed roughly the same, while the RS-RegionPlan yielded more reward.

Figure **??** shows how much reward each algorithm received with different numbers of hallways when rewards were distributed with bait locations that misrepresented the reward of other nodes in the hallway. In this case, the RS-RegionPlan algorithm performed much better than the greedy algorithm. Furthermore, RS-RegionPlan's performance improved significantly with the number of hallways, while the greedy algorithm's performance got worse as the number of hallways increased.

Looped Asterisk Results

Our second experiment was similar to the first, but took place in a looped asterisk environment. Once again, the number of hallways in the environment was varied while each hallway always had 30 rooms. The distance from the center location to the farthest location from the center in each hallway was 200, resulting in a total cost to visit each node in a hallway of just over 400 (200 each direction, plus the length of the edge connecting the two farthest locations).

The results with a random reward distribution are shown in Figure **??**. Our findings are similar to what we found with the split path topology: the two algorithms performed

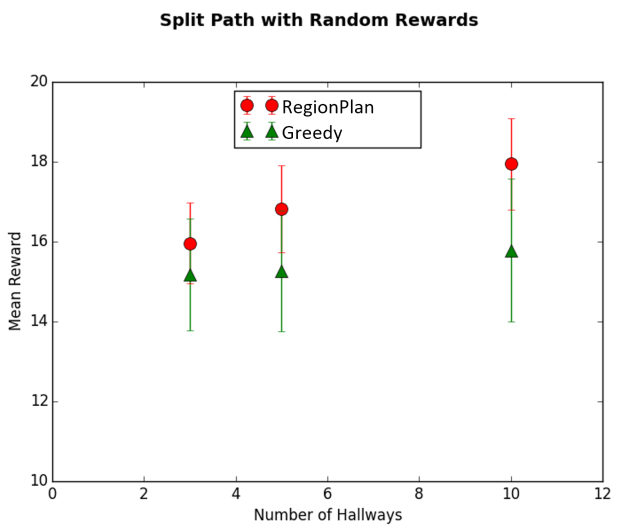


Figure 4.4: The greedy and RS-RegionPlan algorithms' performance in a split path topology with random rewards and varying numbers of hallways.

very comparably with a very small number of hallways. The RegionPlan algorithm showed improvement as the number of hallways increased, while the greedy algorithm showed no significant change.

Figure **??** shows the results when the rewards were distributed with bait locations. This data also matches the findings from the corresponding test with the split path topology. The RS-RegionPlan algorithm showed a slightly better performance than the greedy algorithm for a small number of hallways and improved as the number of hallways increased, while the greedy algorithm's performance got worse as the number of hallways increased.

4.3.4 Discussion

These results are effective in demonstrating strengths of region-based planning. They show that an implementation of RS-RegionPlan using a simple greedy algorithm for region evaluation yields superior plans to those the greedy algorithm would normally produce, and is robust to some particular reward distributions in which an ordinary greedy algorithm struggles.

In environments with randomly distributed rewards, the average expected reward of all locations in a hallway is constant as the number of hallways increases. However, the expected reward of the hallway with the highest reward increases as more hallways are added to the environment, due to the increased likelihood of an outlier hallway with very high reward existing.

The greedy algorithm did not show any noticeable improvement as the number of hallways increased in either topology with randomly distributed rewards. This indicates that the reward it received on average was based on the average reward of the hallways, rather than the maximum reward of the hallways, and therefor it was ineffective at creating a

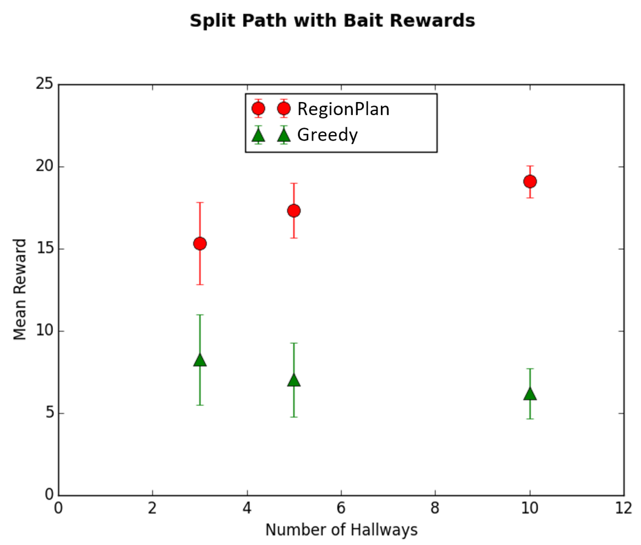


Figure 4.5: The greedy and RS-RegionPlan algorithms' performance in a split path topology with "bait" rewards and varying numbers of hallways.

good plan that took advantage of the reward available in the environment. On the other hand, RS-RegionPlan's performance improved as the number of hallways increased with randomly distributed rewards. This indicates that RS-RegionPlan's performance scaled with the reward of the best hallway, rather than the average reward of all hallways, and thus it was successfully performing a plan that performed tasks in a region with better-than-average rewards, even without any pattern in the reward distribution.

In environments with "bait" locations, the location in each hallway with the highest marginal reward when the agent is at the start location misrepresents the average reward of locations in that hallway. The higher the marginal reward of the "bait" location, the lower the average reward of the rest of the locations in the same hallway. As with the randomly distributed rewards, when the number of hallways in the environment increased, the average total expected reward of hallways in the environment stayed the same but the expected maximum reward of any hallway in the environment increased.

With this reward distribution, the reward received by the greedy algorithm decreased as the number of hallways increased in both topologies. This indicates that it was gathering reward from hallways with lower-than-average total reward, due to being tricked by the presence of a bait location with high marginal reward in those hallways. Meanwhile, RS-RegionPlan's performance increased with the number of hallways, showing that it was not fooled by the bait locations and was once again able to pick out the better-than-average regions to gather reward from.

These differences in performance between the two algorithms are particularly notable because both create their plans greedily. The RS-RegionPlan algorithm creates plans for individual regions using the same greedy algorithm we compared it to. The fact that it shows a noticeable improvement over the greedy algorithm, and is not tricked by the "bait

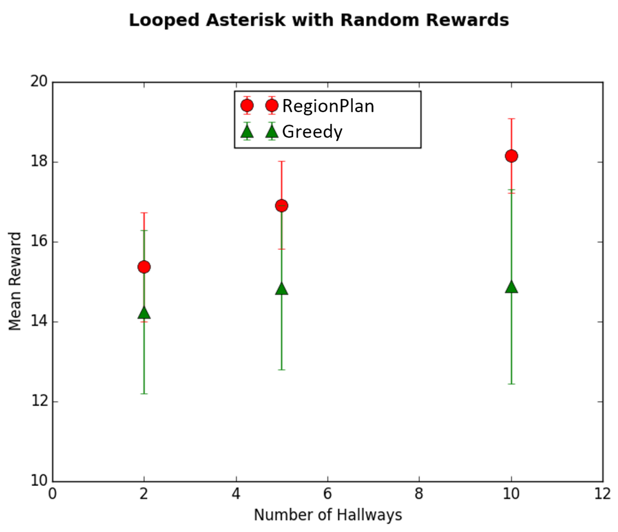


Figure 4.6: The greedy and RS-RegionPlan algorithms' performance in a looped asterisk topology with random rewards and varying numbers of hallways.

nodes" into traveling to sections of the environment with poor average reward, demonstrates the effectiveness of region-based planning.

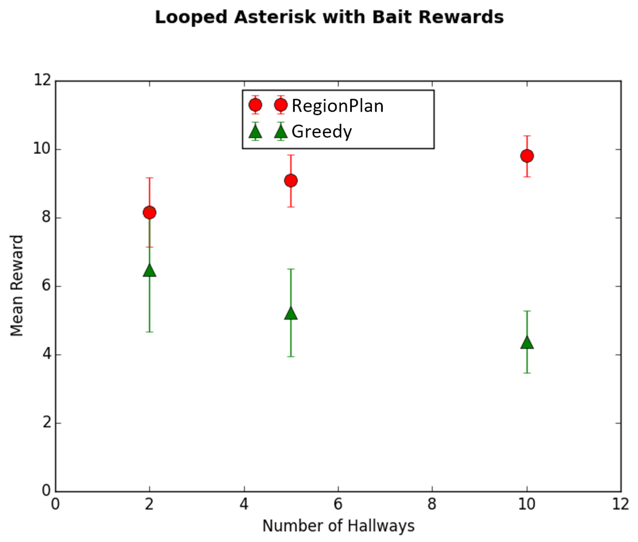


Figure 4.7: The greedy and RS-RegionPlan algorithms' performance in a looped asterisk topology with "bait" rewards and varying numbers of hallways.

Chapter 5

Route-Based Task Planning and Route-MLVNS

In Sections **??** and **??**, we discussed two limitations of common local search operations when used in a hallway-based environment. First, many search operations are simply ineffective due to the constraints on the agent's movement through the environment. Second, when a new locations is added to a plan, the agent will typically pass by many other locations on the way too and from that location, and those locations should be taken into account.

In this chapter, our goal is to create an approach for planning in hallway-based environments that addresses both of these issues. We call our approach route-based task planning (RBTP). The concept behind route-based task planning is to represent the possible solutions to a planning problem not as a sequence of actions, but rather as a route the robot travels through its environment. We can then evaluate routes by determining the maximum reward that can be received within a given length constraint by performing tasks along that route.

We implement route-based task planning with the Route-Based Multi-Level Variable Neighborhood Search algorithm (Route-MLVNS). Route-MLVNS finds a solution to the Orienteering Problem with Task Times by generating an initial route and performing a local search of the route using operations commonly used by heuristics for the Orienteering Problem. The fact that the solutions are represented as routes, rather than as sequences of tasks, changes the structure of the solution-space being searched, and thus the structure of the neighborhoods of each route, in a way that leads to improved performance in hallway-based environments.

Route-MLVNS has the additional feature of being able to return not just a single solution for a single length constraint, but a function that returns a solution for any length constraint within a continuous range. This is partly a feature of the Route-MLVNS algorithm itself, but also takes advantage of a property of Route-Based Task Planning: each route can represent multiple different plans with different lengths, which means the number of plans that can be returned by the function the algorithm produces is greater than the number of routes that must be computed and stored by the algorithm.

5.1 Problem Description

In this chapter we focus on the Orienteering Problem with Task Times (OP-TT) described in Section **??**. An agent operated in an environment defined by a graph G where nodes represent locations where the agent can received reward and edges give the time it takes for the agent to travel between locations. The agent can receive a reward $R(v)$ by traveling to the location v and spending a time of t_{task} there, but cannot receive reward from the same location more than once. The agent begins at a start location v_s, must arrive at the end location v_e within time t_{max}, and the goal is to maximize the reward received in that time.

We also consider a second variant of this problem, the Orienteering Problem with Task Times and Continuous Time Constraints (OP-CTC). Instead of taking in a single time limit t_{max}, the agent is give a range of time constraints from t_0 to t_{max}. The goal is to produce a function f such that, for any time $t_0 < t < t_{max}$, $f(t)$ returns a plan p that starts at v_s, ends at v_e, and receives as much reward as possible within time t. To our knowledge, this variant has not been studied in any previous literature.

This variant could have applications for scenarios which the agent has the ability to expand the time available to perform tasks at some cost. For example, in the Exploration Scheduling problem described in Section **??**, we required the robot to always schedule user requests optimally, but mentioned the possibility of allowing the robot to schedule user requests suboptimally if it improved exploration enough in future work. Computing a function for each interval of exploration that would generate plans for a range of time constraints would allow the agent to determine if it could receive more exploration reward, and how much more, by scheduling user requests suboptimally to change the lengths of the intervals of exploration.

We also assume that the environment the agent operates in is structured into intersections or dead ends connecting different hallways, as described in Chapter **??**. Our approach in this chapter is focused on addressing the issues that simple local search operations can have in hallway-based environments, as described in Section **??**, and considering the nodes that are on the way in any given plan, as described in Section **??**.

5.2 Route-Based Task Planning

In this section, we describe the concept of Route-Based Task Planning (RBTP), our approach to taking advantage of the features of a hallway-based environment structure in order to plan more effectively.

5.2.1 Terminology: Travel Routes, Waypoints, and Task Plans

Before describing RBTP, we will define terminology that we use. A **Travel Route** (or just "route") represents that path the agent takes through the environment to get from the start to the end location. It does not include the tasks that the agent performs in order to receive reward, only the locations that it passes.

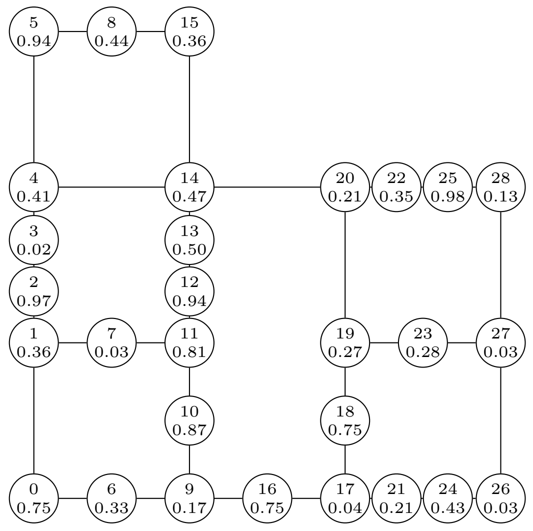


Figure 5.1: An example of a hallway-based environment generated using the algorithm in section **??**. For each node, the top number serves as a label for the node, while the bottom number is the expected reward of performing a task there.

To represent routes, we introduce the concept of waypoints. A **waypoint route** consists of a set of "waypoint" nodes, and represents the travel route that results from taking the shortest path that passes by each waypoint in order. Any node can be a waypoint, and there are no restrictions on how many waypoints can be in a waypoint route or what nodes can serve as waypoints in the same route (although it is possible for waypoints to be redundant and not actually change the path followed). For example, in the environment shown in Figure **??**, the waypoint route $w = [0, 4, 12, 22]$ represents the travel route created by following the shortest path from 0 to 4, then 4 to 12, then 12 to 22. We can find the list of all locations that the agent will pass while following the waypoint route w using Dijkstra's Algorithm, giving us $[0, 1, 2, 3, 4, 14, 13, 12, 13, 14, 20, 22]$. We call this list the **Full Route** of w and denote it with $w_{full}(w)$. In our work, if there are multiple paths of equal length connecting two waypoints we choose one arbitrarily. However, a slightly costlier but more effective alternative could be to consider each route and choose the one that yields the most reward.

We refer to the actual set of tasks that the agent performs as a **Task Plan**. We say that a task plan is "along" a travel route if the route contains all of the locations at which tasks are performed in the plan. For example, any tasks plan the contains only tasks at nodes in the list $[1, 2, 3, 4, 14, 13, 12, 13, 14, 20]$ is along the waypoint route $[0, 4, 12, 22]$.

5.2.2 Route-Based Task Planning

The core concept behind RBTP is that when evaluating a location, the agent should consider not just the reward of performing a task at that location, but all locations contained in the route to and from that location. To accomplish this, we evaluate locations as waypoints on a travel route, rather than destinations to perform tasks. The end goal is to find the route that contains the highest-reward task plan given a time constraint, rather than directly finding a task plan as other heuristics for the orienteering problem and its variants do. In a hallway-based environment, the space of travel routes is both smaller and structured differently from the space of possible task plans, making searching travel routes potentially more effective or efficient in environments with the right structure.

In order to determine the best task plan along a given waypoint route w for a time constraint t_{max}, we must determine how many tasks the agent has time to perform while traveling along w. This is given by

$$N_{tasks} = \text{Floor}\left(\frac{t_{max} - L(w)}{t_{task}}\right), \tag{5.1}$$

where $L(w)$ is the total length of the route (not including any task times). For example, suppose the route has length 100, $t_{max} = 145$, and $t_{task} = 10$. In that case, $N_{tasks} = 4$.

The optimal task plan contains the N nodes along the full route with the highest reward. For example, for the waypoint route $[0, 4, 12, 22]$ with $N_{tasks} = 4$ in the environment shown in Figure **??**, the best four nodes in the full route are 2, 12, 13, and 14. While 0 has a higher reward than 12 or 13, the robot may not perform a task at the start or end location.

For the purposes of our algorithm, we will define the reward $R(w, t_{max})$ of the waypoint route w with time constraint t_{max} to be the reward of the optimal task plan along W with the given time constraint. We will define the task plan length of the route $L_t(w, t_{max})$ to be the duration of the same task plan, including both travel times and task times.

5.2.3 Structure of the Waypoint Route Solution Space

One of the goals of RBTP is to address the issues that arise when performing simple local search operations in a hallway-based environment that we discussed in Section **??**. Performing local search operations on waypoint routes, as opposed to task plans, can avoid those issues. We will give two examples demonstrating situations in which simple local search operations are ineffective or inefficient when performed on task plans in a hwally-based environment, but effective when performed on waypoint routes.

Example: Remove/Insert Operation on Waypoint Route

Consider the environment shown in Figure **??**. Assume every edge has a length of 1, t_{task} is 0.1, and the agent is seeking to find the plan that starts at the node labeled "Start" and ends at the node labeled "'End" with time limit $t_{max} = 10.5$ that maximizes the reward received. The nodes outlined in green represent waypoints in a waypoint route, while the blue nodes represent nodes that are part of the best task plan along that waypoint route. Assume we have an algorithm that is doing a local search of the task plan or waypoint route shown using a "Replace" operation that removes one item in the current plan or route and replaces it with one not in the plan or route.

Consider an agent that is representing solutions in the form of a task plan, a list of the nodes at which it will stop to perform a task and receive reward. Assuming it is currently considering the solution $p = [Start, A, C, O, P, Q, End]$ shown by the nodes in blue. It is exploring new, similar solutions by performing a replace operation on p.

In this case, the agent can remove node A, C, O, P, or Q from the task plan, and add node B, D, I, or N to the plan. No matter which node is removed from the task plan, it will not be possible to add node E, F, G, H, J, K, L, or M using the replace operation.

It is possible to add those nodes to the plan using only the replace operation, but only through multiple very specific steps. For example, in order to consider a plan including E, F, G, or H, it would first need to remove A or C from the plan and insert either I or N into the plan. Then the next operation would need to remove whichever of A or C was not removed in the first step, and then E, F, G, or H could be inserted now that the plan no longer contains A, B, C, or D. Adding J, K, L, or M to the plan would require a even more steps, because O, P, and Q would all need to be removed from the plan before J, K, L, or M could be added.

On the other hand, consider an agent that is instead manipulating a waypoint route, rather than a task plan. Assume it currently has the waypoint route $w = [Start, C, N, End]$, shown by the nodes with thick green outlines, and has found $[Start, A, C, O, P, Q, End]$ to be the best task plan along that route. Now consider the same replace operation described above, but performed on the waypoint route instead, removing a single waypoint from w and adding a single waypoint not in $w_{full}(w)$ (since adding a waypoint that is already in $w_{full}(w)$ would not result in a new route).

In this case, a single Replace operation can remove C from w and insert E, F, G, or H. The same is true for removing N and inserting J, K, L, or M. Thus, by working with a waypoint route, rather than a task plan, an algorithm is able to make a fairly simple and intuitive change - remove one hallway from the route and add another hallway - with a single simple local search operation.

Example: Swap Operation on Waypoint Route

For our next example, we consider a "swap" operation, which consists of exchanging the position of two items within a plan. Typically, the purpose of a swap operation is to find and correct situations in which the locations visited by a plan are good, but the order is inefficient.

Consider the environment show in Figure **??**. Consider the waypoint route $w = [Start, F, A, M, End]$ shown, and the corresponding task plan $p = Start, F, B, A, G, I, N, End]$. This is clearly inefficient: the waypoint route $w' = [Start, A, F, M, End]$ and task plan $p' = [Start, B, A, F, G, I, N, End]$ would be able to receive the same reward in less time time.

This problem can easily be corrected by performing a single swap operation on w. Switching the positions of F and A in w results in the more efficient route w'.

On the other hand, consider performing a swap operation on p. Switching F with B, for example, results in an even longer plan of $[Start, A, F, B, G, J, N, End]$. The same is ture for switching F with B, or any other swap operation. In order to get from p to p' with swap operations, a minimum of two swaps need to be made, swapping F with B and G with A

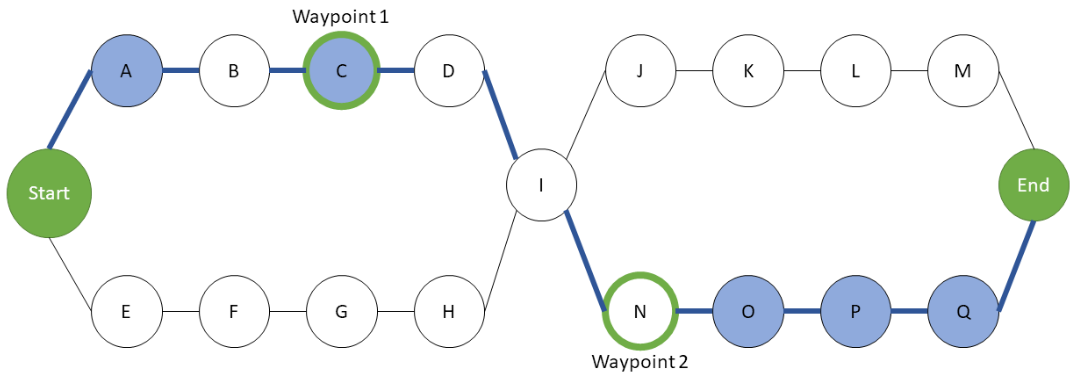


Figure 5.2: An example scenario that demonstrates how using RBTP changes the structure of the solution space. Nodes in blue represent tasks the agent performs, while nodes outline in green represent waypoints. It is possible to switch which hallway the agent travels down just by removing a single waypoint from the route and inserting another. Doing the same by removing and inserting tasks in a task plan would require a series of specific operations.

(swapping F with A and G with B would also result in a slightly different plan with the same length and reward).

The fact that the first swap results in a less efficient plan is significant, however. When a local search algorithm generates a new plan with an operation, it must determine whether to accept or reject the new plan. Because the first swap operation will always yield a longer plan, the algorithm may reject it before the second swap is tested. An algorithm that rejected the plan created by the first swap would be unable to discover the better plan p' using swap operations.

Overall, this case demonstrates another situation in which waypoint routes are potentially superior to task plans when performing a local search. A single simple operation on a waypoint route can correct a route that visits locations in an inefficient order, while correcting the problem on the corresponding task plan using the same operation requires multiple operations and will create a less efficient plan before it finds a more efficient one.

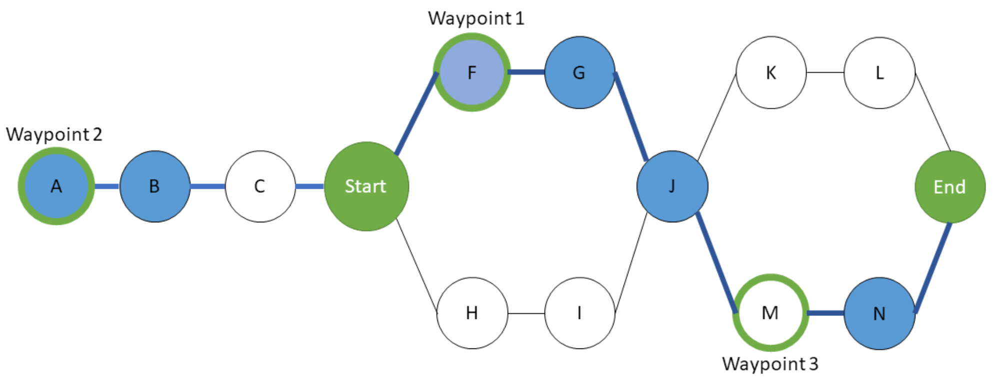


Figure 5.3: An example scenario that demonstrates how using RBTP changes the structure of the solution space. Nodes in blue represent tasks the agent performs, while nodes outline in green represent waypoints. By just changing the place of Waypoints 1 and 2, a more efficient route can be created. The same cannot be done by changing the place of two tasks in the task plan.

5.3 Route-Based Multi-Level Variable Neighborhood Search

We now contribute Route-Based Multi-Level Variable Neighborhood Search (Route-MLVNS), our algorithm that uses Route-Based Task Planning to solve the OP-TT and the OP-CTC in a structured hallway-based environment. Route-MLVNS is based on the existing MLVNS algorithm for solving the orienteering problem[**?**]. However, Route-MLVNS follows the principle of RBTP, using waypoint routes instead of task plans, and is capable of solving the OP-CTC, unlike MLVNS which can only solve the orienteering problem simultaneously for a discrete list of time constraints rather than a continue range.

5.3.1 Main Loop

The Route-MLVNS algorithm solves the OP-TT in a hallway-based environment using Route-Based Task Planning. It can also solve the OP-CTC: rather than producing a single plan, it produces a function that takes a time constraint as input, and gives a plan and reward as output. It achieves this in two ways: First the algorithm is capable of storing and updating a number of waypoint routes, and keeping track of the best waypoint route found for a range of time constraints. Second, each individual waypoint route can represent multiple task plans with different time constraints.

Pseudocode for the main loop of the algorithm is shown in Algorithm **??**. It takes a range of time constraints T, a start location v_s, and end location v_e, and an end constraint that determines when the algorithm stops (such as a number of iterations or a maximum amount of time spent running the algorithm). The algorithm also takes a list of neighborhood structures *nbhds*, each of which is a function that takes as input a waypoint route w, some number of waypoints in w, and some number of nodes not in $w_{full}(w)$, and returns a new waypoint route w'. The algorithm returns W_{best}, a list of the best waypoint route found for any time constraint in T. The algorithm has three primary steps:

Initialization: Create a set of initial waypoint routes using a simple algorithm.

Neighborhood Search and Update: Perform a local search of a stored waypoint route, using a series of neighborhood operations to look for new, better routes.

Perturbation: If no improvements are found during the neighborhood search of any stored waypoint route, generate new routes in order to avoid getting stuck at a locally optimal solution.

The algorithm begins with the initialization step (Lines **??** through **??**). The Neighborhood Search and Update (NSU) step is then repeated as long as there are any routes whose neighborhoods have not yet been searched (line **??**). If the algorithm has searched all stored routes without finding a new route that improves on any existing routes, then a perturbation step is executed to generate new routes and it returns to the NSU step (line **??**). After each iteration, the stored list of the best routes that have been found is updated (line **??**). This process repeats until the given end condition is reached, at which point it returns a function that takes a time constraint as input and gives the best stored route for that time constraint as output (lines **??** and **??**).

Algorithm 5 Route-MLVNS

```
1: procedure MLVNS($T, v_s, v_e, nbhds$, End Constraint)                ▷ Initialization
2:     $W_{searched} \leftarrow []$
3:     for $t_i \in T$ do
4:         $W_{current}(t_i) \leftarrow$ InitializeRoute($t_i, v_s, v_e$)
5:         $W_{best}(t_i) \leftarrow W_{current}(t_i)$
6:     while End Constraint is not met do
7:         if $W_{current}$contains any unsearched routes then
8:             $W_{current}, W_{searched} \leftarrow$ NbhdSearch($W_{current}, W_{searched}, nbhds$)
9:         else
10:            $W_{current} \leftarrow$ Perturb($W_{current}$, $T$, $G$, $v_s$, $v_e$, $t_{task}$, $n_{min}$, $n_{max}$, $W_{searched}$, $W_{max}$)
11:        $W_{best} \leftarrow$ Update($W_{best}, W_{current}$)
12:    return $P_{best}$
```

5.3.2 Initialization

The initialization step is given a set of initial time constraints and generates a set of waypoint routes, one for each given time constraint. Pseudocode for the algorithm to initialize a plan for a given time constraint t_{max} is shown in Figure **??**. The algorithm begins with a route containing only the start and end nodes (Line **??**). For each node not in the full route, it computes the shortest route that can be created by inserting that node as a new waypoint into the route, and determines the reward of the best task plan along the resulting route for the given time constraint (Lines **??** through **??**). It then picks the insertion that results in the task plan with the highest reward without violating the time constraint and performs that insertion (Lines **??** through **??**). This process is repeated until no insertion is performed (meaning all possible waypoint insertions either violate the time constraint or result in a lower-reward plan), at which point the route is returned.

Algorithm 6 Path-MLVNS Plan Initialization

```
1: procedure InitializeRoute($G$, $t_{max}, v_s, v_e$)
2:     $w \leftarrow [v_s, v_e]$
3:     $inserted \leftarrow True$
4:     while $inserted = True$ do
5:         $R(w) \leftarrow R(w, t_{max}), L(p) \leftarrow L_t(w, t_{max})$
6:         $R_{best} \leftarrow R(w)$
7:         $w_{best} \leftarrow w$
8:         $inserted \leftarrow False$
9:         for Node $v$ in $V - w_{full}(w)$ do
10:            $w_v \leftarrow$ shortest route inserting $n$ into $w$
11:            $R(w_v) \leftarrow R(w_v, t_{max})$
12:            $L(w_n) \leftarrow L_t(w_v, t_{max})$
13:            if $L(w_v) \leq t_{max}$ and $R(w_v) > R_{best}$ then
14:                $R_{best} \leftarrow R(w_v)$
15:                $p_{best} \leftarrow w_v$
16:                $inserted \leftarrow True$
17:        $w \leftarrow w_{best}$
18:    return $w$
```

5.3.3 Route Storage and Update

The Route-MLVNS algorithm stores two sets of routes: The current routes being being tested $W_{current}$, and the Best Routes found so far W_{best}. The set of current routes is updated throughout each iteration of the neighborhood search step, and changed completely during the perturbation step. The best routes are updated with any new routes found at the end of each neighborhood search or perturbation step.

Pseudocode for updating a set of stored routes W with a set of one or more new routes W_{new} is shown in Algorithm **??**. For each route $w_{new} \in W_{new}$, the algorithm determines the reward function $R(w_{new}, t)$ of the maximum reward that can be acquired while traveling along the route w_{new} in time t (line **??**). In our implementation, whenever the algorithm computed the reward of a route w, it stored the full route $w_{full}(w)$ and the computed reward. Whenever it found a new route with the same full route, it would simply use the stored reward rather than recomputing it. This reduces the computational time of the algorithm, at the cost of increasing the memory used.

The algorithm then determines if there is any t such that $R(w_{new}, t) > \max_{w \in W} R(w, t)$. If there is, it means w_{new} is the new optimal known plan for t, so w_{new} is added to W (Lines **??** through **??**)). If any routes in W_{new} were added to W, an additional cleanup step is performed. In the cleanup step, the algorithm calculates the "significance" of each $w \in W$ (including the new plans that were added) (Line **??**). We define the significance of a route $w \in W$ as

$$S(w, W) = \int_{t=0}^{t_{max}} B(w, W, t)dt, \tag{5.2}$$

where

$$B(w, W) = \begin{cases} 1 & \text{if } R(w, t) = \max_{w' \in W} R(w', t) \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

In other words, the significance of a route is the total span of time constraints for which that is the highest-reward route in W. Any route remaining in W that has a significance of 0 (i.e. any route that is not the best known route for any time constraint) is removed from W (Lines **??** through **??**). Additionally, if a maximum number of stored routes W_{max} has been set, then only the W_{max} routes with the highest significance are kept, and all others are removed (lines **??** through **??**).

There are two times during which the stored routes are updated using this function. During the Neighborhood Search step, the stored current routes $W_{current}$ are updated with new routes that are found. After each Neighborhood Search step, there is a special Update step in which the stored Best Routes W_{best} are updated with the stored Current Routes $W_{current}$. Finally, each perturbation step creates a new $W_{current}$ and updates it with a set of new routes that are generated.

5.3.4 Neighborhood Search

The neighborhood search step iterates through a sequence of neighborhood structures of a waypoint route, searching for new routes that improve on the existing stored routes. Each neighborhood structure is represented by an operation that makes a small change to a route. There are five neighborhood operations used, depicted in Figure **??**. These are the same five neighborhood operations used by the MLVNS algorithm, and are commonly used in local search algorithms when solving the orienteering problem[**?**, **?**]:

Algorithm 7 MLVNS Update

```
1: procedure UPDATE($W_{new}$, $W$, $W_{max}$)
2:     Compute $R(w_{new}, t)$
3:     for $w_{new} \in W_{new}$ do
4:         if $R(w_{new}, t) > \max_{w \in W} R(w, t)$ for any $t$ then
5:             Add $w_{new}$ to $W$
6:     $W_{removed} \leftarrow []$
7:     if any routes were added to $W$ then
8:         for $w_i \in W$ do
9:             Compute $S(w_i, W)$
10:            if $S(w_i, W) = 0$ then
11:                Remove $w_i$ from $W$
12:                Append $w_i$ to $W_{removed}$
13:        if $W_{max}$ was given then
14:            Remove all but the $W_{max}$ routes with the highest significance from $W$
15:            Append the removed routes to $W_{removed}$
16:    return $W$, $W_{removed}$
```

1. **Swap within Route:** Exchange the position of two waypoints within the route (Figure **??**).

2. **2-Opt:** Reverse the order of waypoints in a section of the route (Figure **??**).

3. **Insertion Move:** Move a waypoint from one position in the route to a different position (Figure **??**).

4. **Replacement:** Replace a waypoint in the route with a node not in the route (Figure **??**).

5. **Insertion:** Insert a node not in the route into the route as a waypoint (Figure **??**).

No neighborhood operation may ever change the first or last node of the route – it must always start at v_s and end at v_e. A "neighborhood structure" is the set of all waypoint routes that can be created with a single application of a neighborhood operation on a waypoint route.

Pseudocode for the neighborhood search step is shown in Algorithm **??**, which takes the stored current routes W, a list of routes that have already been searched in the neighborhood search step in the past $W_{searched}$, and a list of neighborhood structures *nbhds* and returns an updated version of W with any better routes found during the neighborhood search, as well as an updated version of $W_{searched}$ with the route that was searched added to it. It begins by choosing the route w_{next} in $W_{current}$ with the highest significance (as described in equations **??** and **??**) that has not already had its neighborhoods searched in a previous iteration (line **??**).

The algorithm then calculates the set of new routes W_{nbhd} within the first neighborhood structure of w_{next} (line **??**). It updates the stored plans $W_{current}$ with the plans in W_{nbhd} using the Update algorithm described in Section **??** (line **??**). If, in the process, w_{next} is removed from $W_{current}$ (because its significance is reduced to 0 or it is no longer among the W_{max} most significant routes when the new plans are added), the search ends (lines **??** and **??**). Otherwise, the next neighborhood is searched. This continues until w_{next} is removed from $W_{current}$ or all neighborhoods of w_{next} are searched, at which point $W_{current}$ is returned (line **??**).

Algorithm 8 MLVNS Neighborhood Search

```
1: procedure NeighborhoodSearch($W, W_{searched}, nbhds$)
2:     $w_{next} \leftarrow w$ such that $S(w, W) = \underset{w \in W - W_{searched}}{\text{argmax}} S(p, W)$
3:     for $nbhd \in nbhds$ do
4:         $W_{nbhd} \leftarrow$ plans in neighborhood structure of $w_{next}$
5:         $W \leftarrow$ Update($W_{nbhd}, W$)
6:         if $w_{next} \in W_{removed}$ then
7:             return $W$
8:     Add $w_{next}$ to $W_{searched}$
9:     return $W, W_{searched}$
```

5.3.5 Perturbation

The perturbation step is designed to prevent the algorithm from getting stuck at a locally optimal solution. When the algorithm has searched the neighborhoods of every route in $W_{current}$ without storing any new routes, the perturbation step is executed, creating an entirely new set of routes to search.

The perturbation step generates new plans in two ways: random deletion and random route generation. Pseudocode for the random deletion algorithm is shown in Algorithm **??**. It takes a waypoint route w and randomly selects a number of waypoints, with a minimum of one and a maximum of the total number of waypoints in the route, not counting the start and end locations (line **??**). It then randomly selects that many waypoints from the route (line **??**) and removes them (line **??**).

Pseudocode for random route generation is shown in Algorithm **??**. Random route generation takes various values required for route generation as input (the environment G, start and end locations v_s and v_e, a maximum plan length t_{max}, the task time t_{task}), as well a minimum and maximum number of nodes n_{min} and n_{max}, and a list of routes whose neighborhoods have already been searched. The algorithm begins by selecting a random number of waypoints n between n_{min} and n_{max} to be part of the random route (line **??**), and creating a new waypoint route w consisting only of the start and end locations (line **??**). It then attempts to insert random waypoints into w until n waypoints have been inserted or it has attempted to insert every possible waypoint (line **??**).

To attempt to insert a node, it first chooses a random node v_{next} in the environment that is not already part of the full route $w_{full}(w)$ (line **??**). It then inserts the node into a random spot in w (line **??**). If the resulting route w_{next} has not already had its negibhorhoods search, and its length is not greater than $t_{max} - t_{task}$ (the length required to perform at least one task without exceeding the time limit), then it is accepted and it becomes the new w and the process repeats if there are still more waypoints to insert (lines **??** through **??**). If w_{next} has already been searched before or is too long, then v_{next} is inserted into a different location in w, and so on until a new route is accepted or every possible insertion is tried (line **??**). New waypoints continue to be inserted until n have been inserted or all waypoints have been tried, at which point the final w is returned (line **??**).

Pseudocode for the perturbation step as a whole is shown in Algorithm **??**. The algorithm first generates one new route using random deletion from each plan in $W_{current}$ (lines **??** through **??**). It then generates one random route for each of the original time constraints used at initialization (lines **??** through **??**). Finally, it creates a new structure for storing routes by inserting all of the routes created through random deletion and generation into an empty structure using the Update procedure described in Algorithm **??** (lines **??** through **??**).

Algorithm 9 MLVNS Random Deletion

```
1: procedure RandomDeletion($w$)
2:     $n \leftarrow$ random integer between 1 and $(|w| - 2)$
3:     $w_{chosen} \leftarrow$ randomly chosen set of$n$ nodes in $w$
4:     $w_{delete} \leftarrow w - w_{chosen}$
5:     return $w_{delete}$
```

5.3.6 Differences from MLVNS

The Route-MLVNS algorithm is based on the MLVNS algorithm for solving the orienteering problem for multiple time constraints simultaneously presented by Liang et al.[**?**]. The MLVNS algorithm has a similar main loop of initializing a set of solutions, performing neighborhood search and update steps until no improvement is found, and then performing a random perturbation step to generate a new set of solutions. Our Route-MLVNS algorithm also uses the same neighborhood operations as the MLVNS algorithm, although those operations are commonly used in heuristics for the orienteering problem and not specific to the MLVNS algorithm[**?**].

However, there are key differences that distinguish the Route-MLVNS algorithm. The most significant is Route-MLVNS's use of Route-Based Task Planning and waypoint routes. While MLVNS represents solutions as task plans and its neighborhood operations manipulate tasks, Route-MLVNS represents solutions as waypoint routes and its neighborhood operations manipulate the waypoints. As demonstrated in Section **??**, the same operations can have very different effects when used on waypoint routes instead of task plans.

Algorithm 10 MLVNS Random Generation

```
1: procedure RandomGeneration($G$, $v_s$, $v_e$, $t_{max}$, $t_{task}$, $n_{min}$, $n_{max}$, $W_{searched}$)
2:     $n \leftarrow$ random integer between $n_{min}$ and $n_{max}$
3:     $w \leftarrow [v_s, v_e]$
4:     $n_{inserted} \leftarrow 0$
5:     while $n_{inserted} < n$ do
6:         $v_{next} \leftarrow$ random node in $V$ not in $w_{full}(w)$
7:         repeat
8:             $w_{next} \leftarrow$ Insert $v_{next}$ into random spot in $w$
9:             if $w_{next} \notin W_{searched}$ and $L(w_{next}) \leq t_{max} - t_{task}$ then        ▷ Accept Insertion
10:                $w \leftarrow w_{next}$
11:                $n_{inserted} \leftarrow n_{inserted} + 1$
12:            else
13:                Reject Insertion
14:        until Insertion is accepted or all indices are checked
15:    return $w$
```

The second key difference is that MLVNS is designed only to solve the orienteering problem for a given list of discrete time constraints. It stores only one plan for each time constraint, and only seeks to maximize the reward at each given time constraint. Route-MLVNS, on the other hand, is designed to solve the orienteering problem for a continuous range of time constraints. It does this by doing two things differently from MLVNS. First, it chooses which waypoint routes to store based on their "significance," as opposed to MLVNS which simply keeps the best solution found at each discrete time constraint. Second, the waypoint routes stored in W_{best} by Route-MLVNS each potentially represent multiple different task plans for different time constraints. In contrast, each solution stored by MLVNS is just a single task plan representing a single solution to the orienteering problem at a single time constraint.

While those are the major differences, the two algorithms differ in many minor ways too. For example, because the number of routes stored by the Route-MLVNS algorithm is variable, it searches the neighborhoods of only a single waypoint route each neighborhood search and update step, while MLVNS iterates through each stored plan in every iteration. The MLVNS algorithm's perturbation step also only contains the random deletion method, as opposed to Route-MLVNS which features both random deletion and random generation.

5.4 Experimental Results

We performed experiments in simulation in order to determine the effectiveness of the Route-MLVNS algorithm. In these experiments, we compared it to the MLVNS algorithm for solving the orienteering problem for multiple time constraints described by Liang et al.[**?**].

Algorithm 11 MLVNS Perturbation

```
1: procedure PERTURB($W_{current}$, $T$, $G$, $v_s$, $v_e$, $t_{task}$, $n_{min}$, $n_{max}$, $W_{searched}$, $W_{max}$)
2:     $W_{new} \leftarrow []$
3:     for $w_{current} \in W_{current}$ do
4:         $w_{delete} \leftarrow$ RandomDeletion($w_{current}$)
5:         Add $w_{delete}$ to $W_{new}$
6:     for $t \in T$ do
7:         $w_{random} \leftarrow$ RandomGeneration($G$, $v_s$, $v_e$, $t$, $t_{task}$, $n_{min}$, $n_{max}$, $W_{searched}$)
8:         Append $w_{random}$ to $W_{new}$
9:     $W_{perturbed} \leftarrow []$
10:    $W_{perturbed} \leftarrow$ Update($W_{new}, W, W_{max}$)
11:    return $W_{perturbed}$
```

5.4.1 Environments

We compared the two algorithms in three different types of environment structures: Procedurally generated networks of hallways in a grid-based setup, and two maps of actual office building floors in which a service robot operates. In all environments, the start and end location were chosen randomly for each trial such that there was sufficient time to travel from the start location to the end location while performing at least one task on the way for all time constraints. Each location was always assigned a random reward between 0.0 and 1.0.

Procedurally-Generated Environments

In order to increase the variety of our experiments, we used an algorithm to procedurally-generate simulated hallway-based environments. To do this, we used the following process:

1. Generate an $X \times Y$ grid of evenly-spaced nodes, each with edges connecting it to the orthogonally adjacent nodes in the grid.

2. Delete a random node from the graph such that the graph remains connected after deletion. Repeat until N nodes have been deleted, or until no more nodes can be deleted without breaking the graph's connectivity.

3. Randomly delete up to E edges while ensuring the graph remains connected in the same manner as nodes were deleted in step 2.

4. For each remaining edge in the environment, choose a random integer h between h_{min} and h_{max}. Replace that edge with a hallway containing h evenly-spaced new nodes (if $h = 0$ no change is made).

An example of an environment generated using this method with a 4×4 grid, 2 deleted nodes, 1 deleted edge, $h_{min} = 0$, and $h_{max} = 2$ is shown in Figure **??**. In the actual experiments, we used parameters of $X = Y = 5$, $N = 1$, $E = 2$, $h_{min} = 1$, and $h_{max} = 5$.

Real Office Environment

We also performed tests using two real maps of office buildings in which service robots operate. The first is the 7th floor of the Gates Hillman Center at Carnegie Mellon University. An image of the floorplan is shown in Figure **??**, while a graph version of the environment is shown in Figure **??**. This map contains 84 offices at which the robot can perform tasks. The maximum distance between any two rooms in the map is 166.38 meters.

The second is the 4th floor of Newell Simon Hall at Carnegie Mellon University. This map contains 60 offices at which the robot can perform tasks, and has a maximum distance of 244.37 meters. The environment is shown as a graph in figure **??**. The fourth floor of Newell Simon Hall has a more unusual layout than the Gates Hillman Center. The environment resembles a very long, winding hallway with short hallways or individual locations branching off from it, rather than a network of hallways like the other environments tested.

In both environments, we assumed that the robot travels at a speed of one meter per second.

5.4.2 Single Time Constraint Tests

We first present the results of tests in which each algorithm was used to find only a single plan for a single time constraint (solving the regular OP-TT rather than the OP-CTC). Figures **??** through **??** show the results of these tests. The tests took place in environments where the time to perform a task was 10 seconds, and we tested time limits of 300 seconds and 500 seconds in each environment. The end constraint was after the algorithm had been running for two minutes, including initialization time. The results are each averaged over 10 trials with random starting points, end points, and rewards between 0.0 and 1.0 assigned randomly to each node in the environment. For the procedurally-generated grids a new environment was generated for each test.

As can be seen, in all six tests, the Route-MLVNS Algorithm received more reward than the MLVNS algorithm, with the difference being greater than the variance in all cases except the tests in Newell Simon Hall. The smallest proportional difference was in the Gates Hillman Center with a plan length of 500, where on average Route-MLVNS received about 17.5% more reward than MLVNS. The largest difference was in the Gates Hillman Center with a plan length of 300, where Route-MLVNS received about 81.6% more reward than MLVNS on average.

5.4.3 Continuous Time Constraint Tests

One of the notable features of MLVNS and Route-MLVNS not present in most algorithms that solve the orienteering problem is the ability to solve the OP-CTC, solving the orienteering problem for a continuous range of time constraints simultaneously. We performed additional tests in to compare the two algorithms in their ability to solve the OP-CTC. Figures **??** through **??** show the results of these tests.

In each test, each algorithm began by creating 6 plans during initialization for time constraints of 100, 200, 300, 400, 500, and 600. Because the MLVNS algorithm can only solve the orienteering problem for a discrete list of time constraints, it only found plans for each of those six given time constraints. Meanwhile, the Route-MLVNS algorithm is not limited to the specific time constraints given as part of initialization, but can instead be given a continuous range of time constraints. It was given a range from 0 to 700, and was limited to storing no more than 10 routes. Note that because of the nature of Route-Based Task Planning, a single route can represent multiple possible plans at different time constraints.

The rest of the parameters were the same as in the single-plan trials. Each algorithm was allowed to spend 120 seconds planning including initialization, the cost of performing a task was 10, and tests were performed using environments based on the 7th floor of the Gates Hillman and the 4th floor of Newell Simon Hall at Carnegie Mellon University, as well as procedurally-generated grids using the process described in Section **??**.

In all three experiments, the Route-MLVNS algorithm received more reward on average than the MLVNS algorithm at nearly all time constraints. The one exception was at time constraints of about 100 second, where the MLVNS algorithm performed very slightly better on average in all three environments.

These results also clearly demonstrate the benefits of the Route-MLVNS algorithm finding solutions for a continuous range of time constraints rather than a discrete list. The average reward received by the Route-MLVNS algorithm increased approximately linearly with the time constraint, while the MLVNS algorithm's average reward was a step function, only increasing at the given time constraints. As a result, the difference in reward between the two algorithms increased as the time constraint increased between each pair of time constraints. However, even at the given time constraints that the MLVNS algorithm found plans for, the Route-MLVNS algorithm still received more reward on average in all three environments at all length constraints except 100, especially in the grid and Gates Hillman Center environments. In the Newell Simon Hall environment, the difference between the two algorithms was smaller, and they came within one standard deviation of each other at the given time constraints.

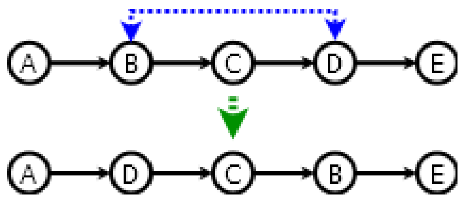
5.5 Conclusion

In this chapter, we presented the concept of Route-Based Task Planning, an approach to planning in hallway-based environments such as office buildings and streets. In Route-Based Task Planning, we treat solutions not as a sequence of tasks, but as a sequence of "waypoints" representing a path taken through the environment. We evaluate a path for a given time constraint by calculating the best set of tasks along the path to perform without exceeding the constraint.

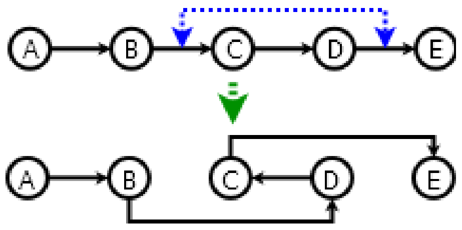
We presented the Route-MLVNS algorithm, an algorithm designed for solving the Orienteering Problem with Task Times in hallway-based environments using Route-Based Task Planning. Route-MLVNS is also capable of solving the Orienteering Problem with Task Times and Continuous Time Constraints. We compared Route-MLVNS in simulation to the

MLVNS algorithm from the literature that is also capable of solving the orienteering problem simultaneously for a set of discrete time constraints and found it to received more reward when computing solutions for a single time constraint or a range of time constraints. The Route-MLVNS algorithm's performance was especially strong in environments that consisted of an interconnected network of hallways. The gap between Route-MLVNS and MLVNS was not as strong in the Newell-Simon Center environment, which resembled a long, winding hallway with smaller hallways branching off from it, rather than a network of interconnected hallways like the other environments tested.

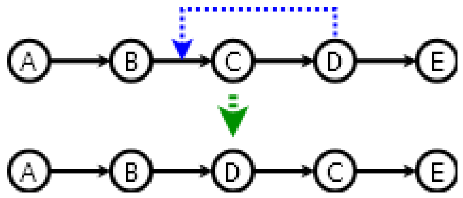
Overall, we believe that Route-MLVNS is a useful algorithm for solving the orienteering problem in hallway-based environments, especially in situations in which we wish to solve the orienteering problem for a range of length constraints simultaneously. We believe route-based task planning is an approach that may also have further applications for planning beyond the Route-MLVNS algorithm that we hope to explore in the future.



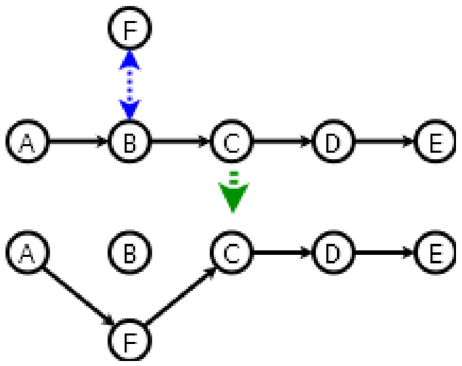
(a) The Swap neighborhood operation.



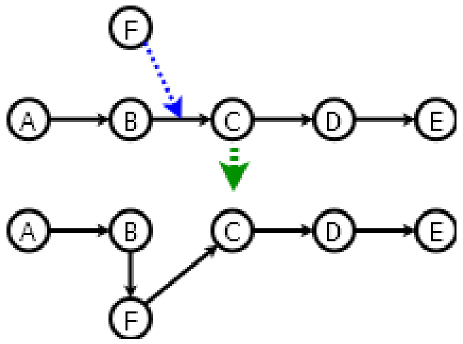
(b) The 2-Opt neighborhood operation.



(c) The Insertion Move neighborhood operation.



(d) The Replacement neighborhood operation.



(e) The Insertion neighborhood operation.

Figure 5.4: Illustrations of the neighborhood structures used by the Route-MLVNS algorithm.

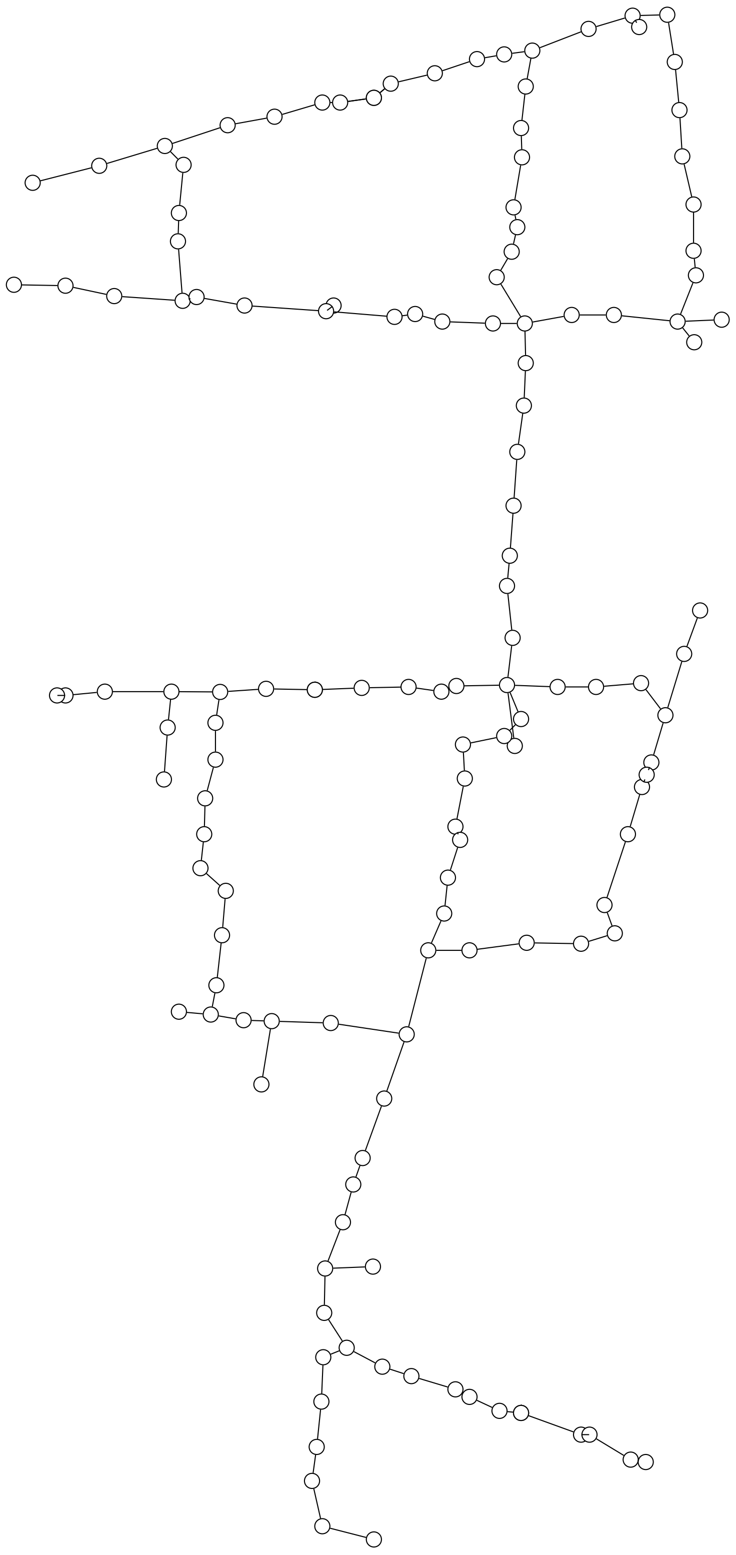


Figure 5.5: A graph representation of the seventh floor of the Gates Hillman Center at Carnegie Mellon University.

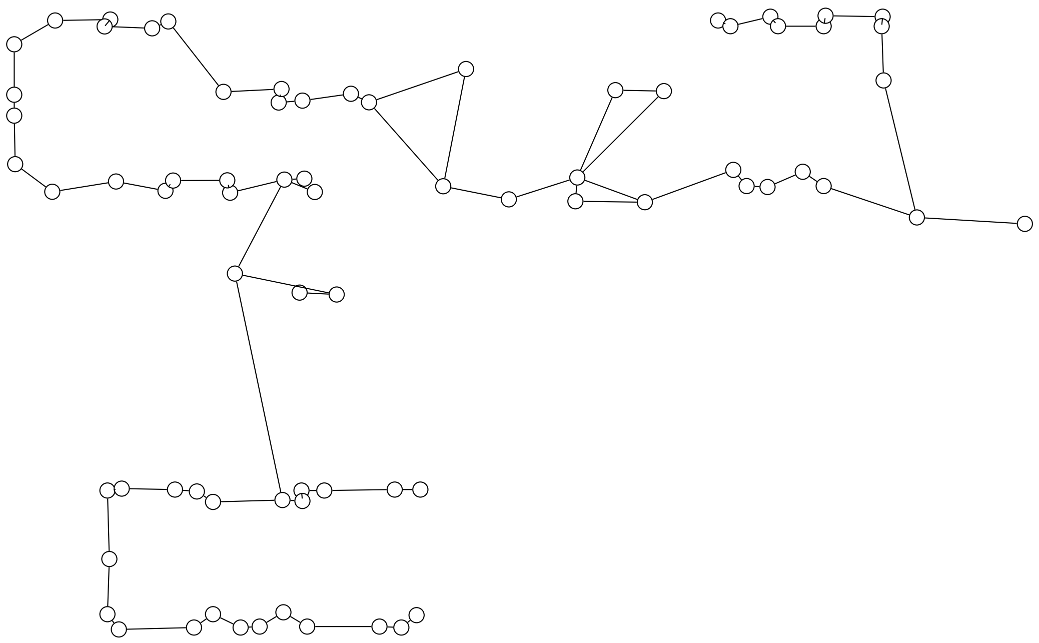


Figure 5.6: A graph representation of the fourth floor of Newell Simon Hall at Carnegie Mellon University.

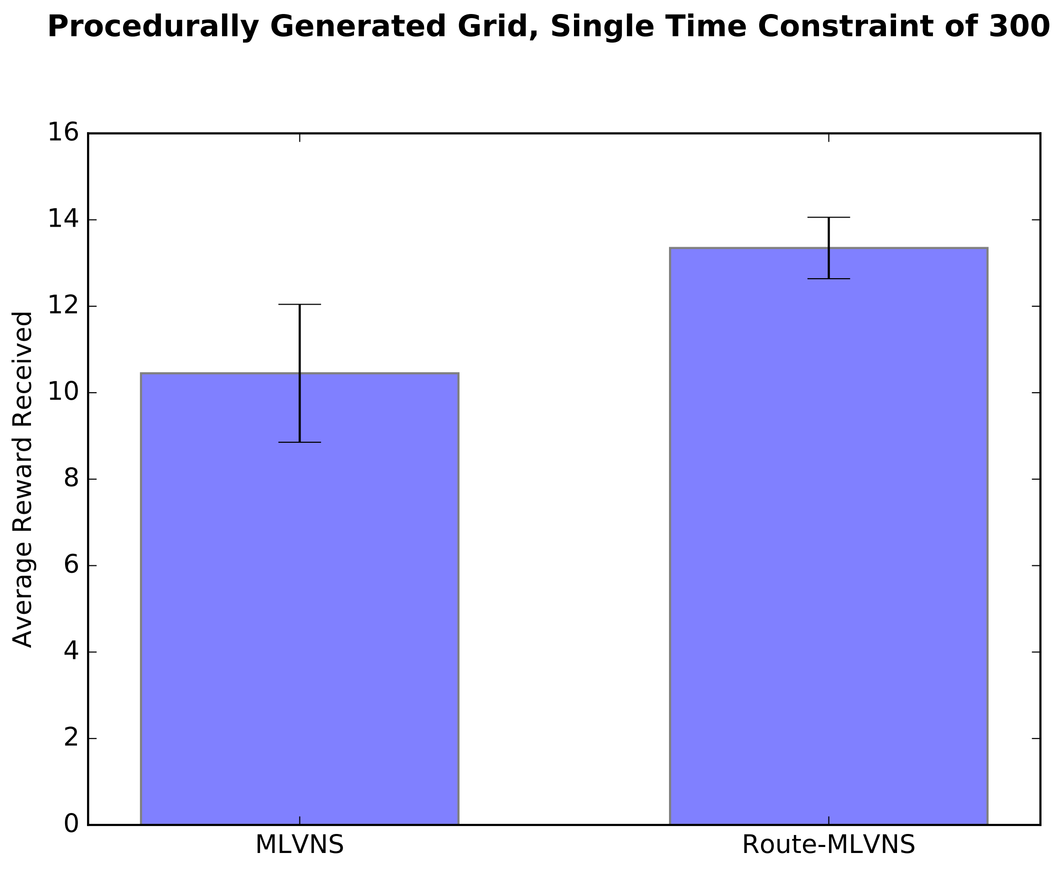


Figure 5.7: The average reward received by Route-MLVNS and MLVNS finding a plan of length 300 in a procedurally-generated 5x5 grid.

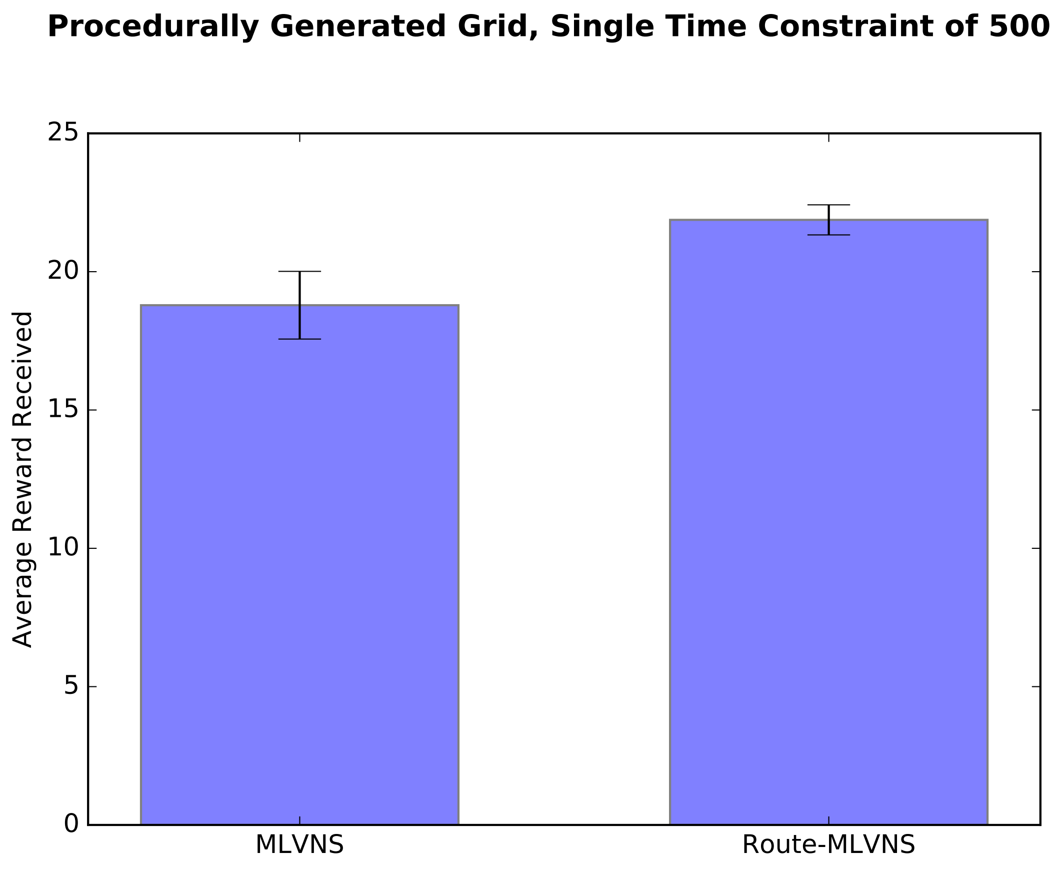


Figure 5.8: The average reward received by Route-MLVNS and MLVNS finding a plan of length 500 in a procedurally-generated 5x5 grid.

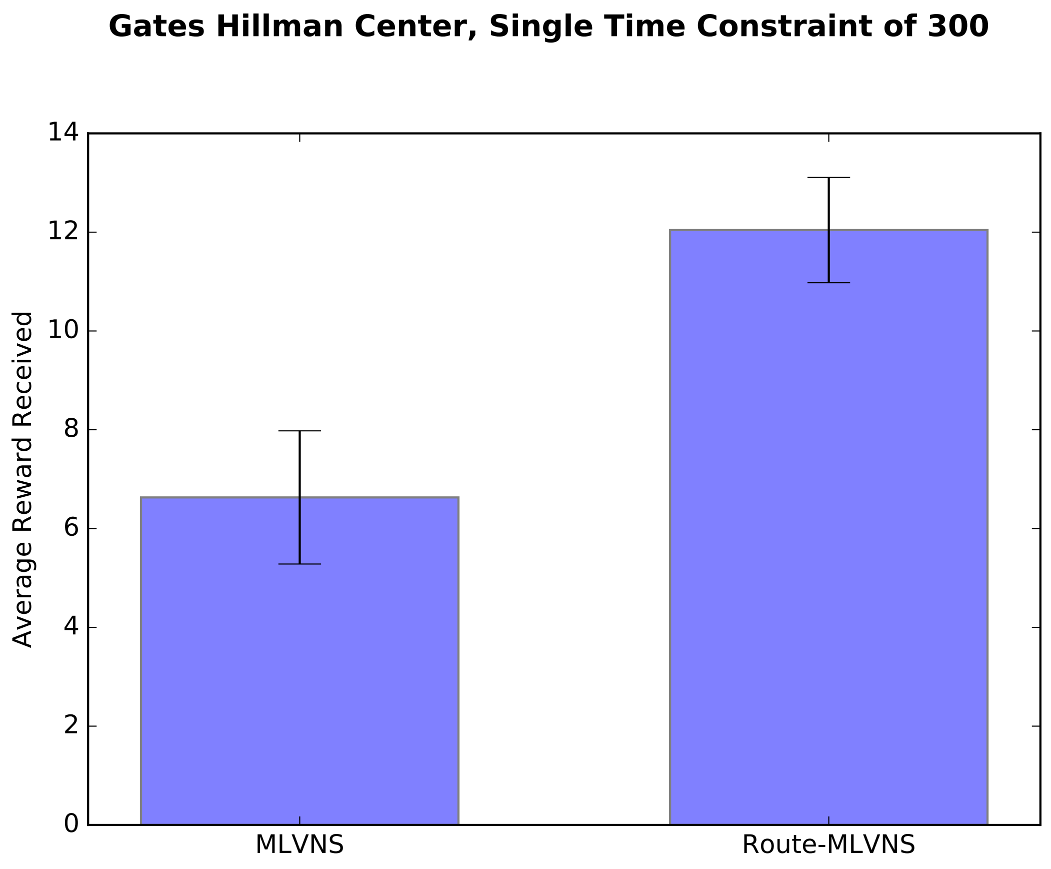


Figure 5.9: The average reward received by Route-MLVNS and MLVNS finding a plan of length 300 in a the 7th floor of the Gates Hillman Center.

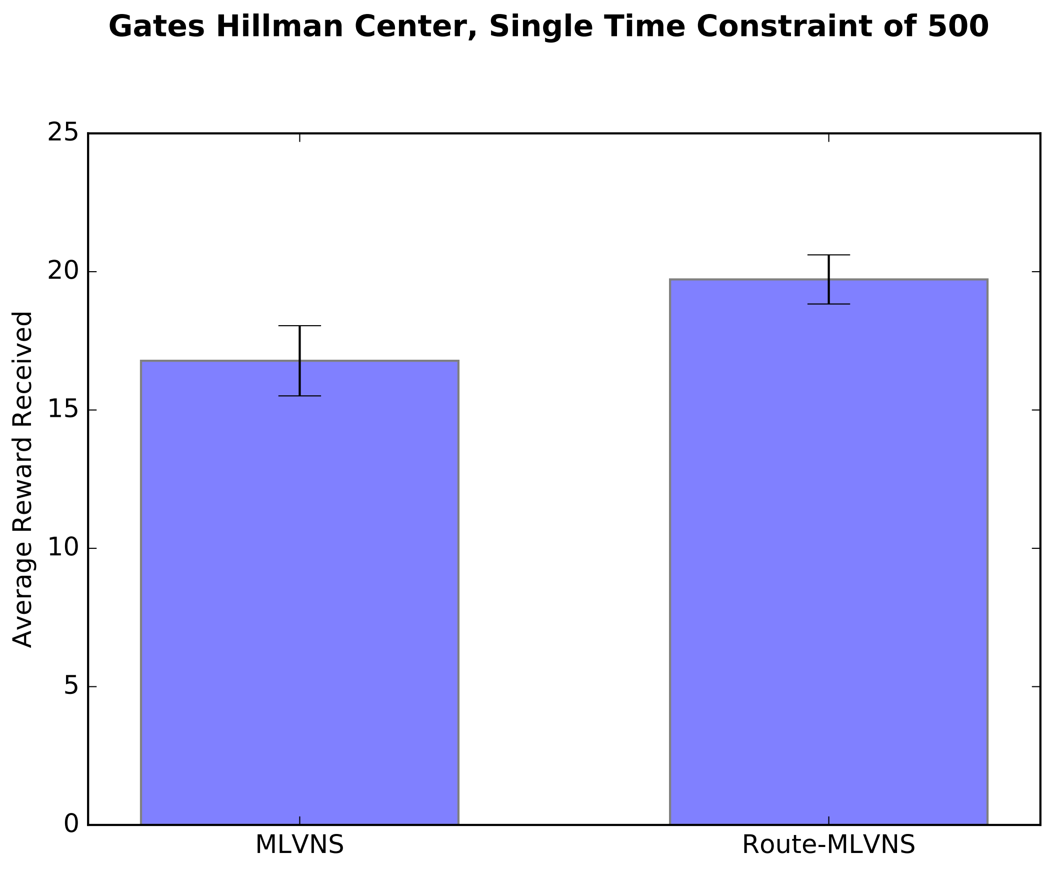


Figure 5.10: The average reward received by Route-MLVNS and MLVNS finding a plan of length 500 in the 7th floor of the Gates Hillman Center.

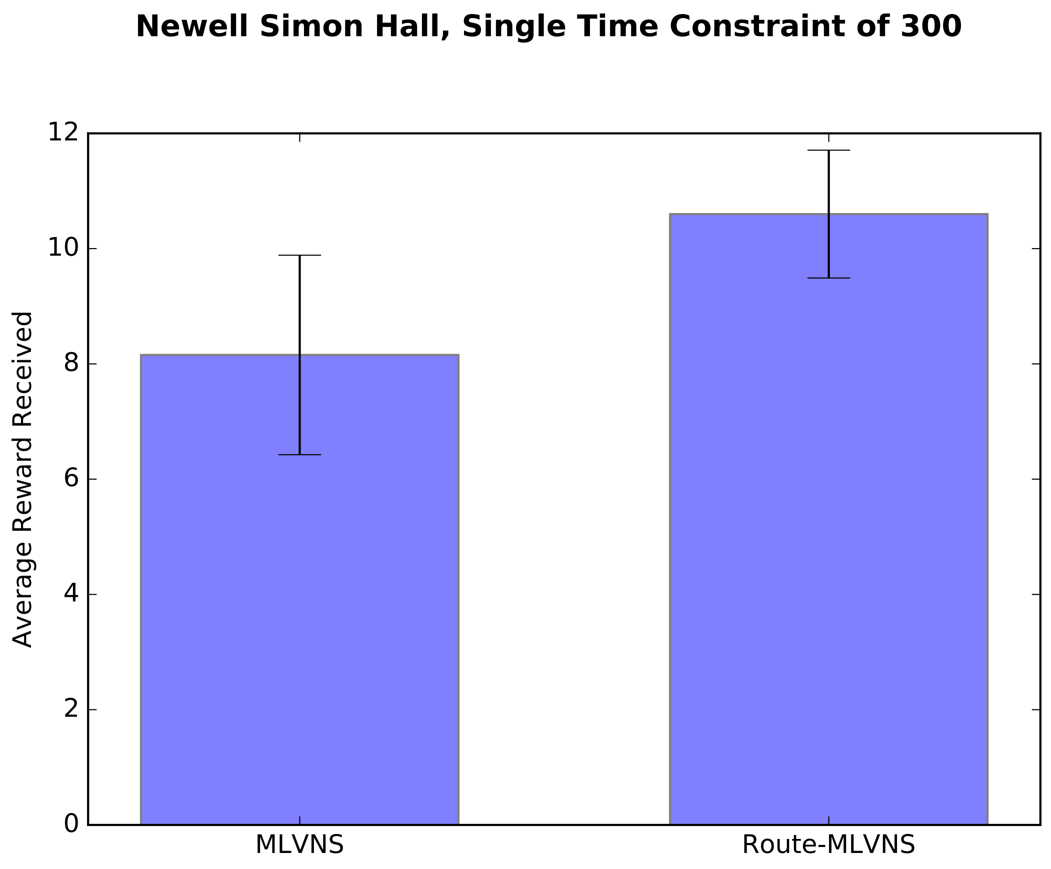


Figure 5.11: The average reward received by Route-MLVNS and MLVNS finding a plan of length 300 in the 4th floor of Newell Simon Hall.

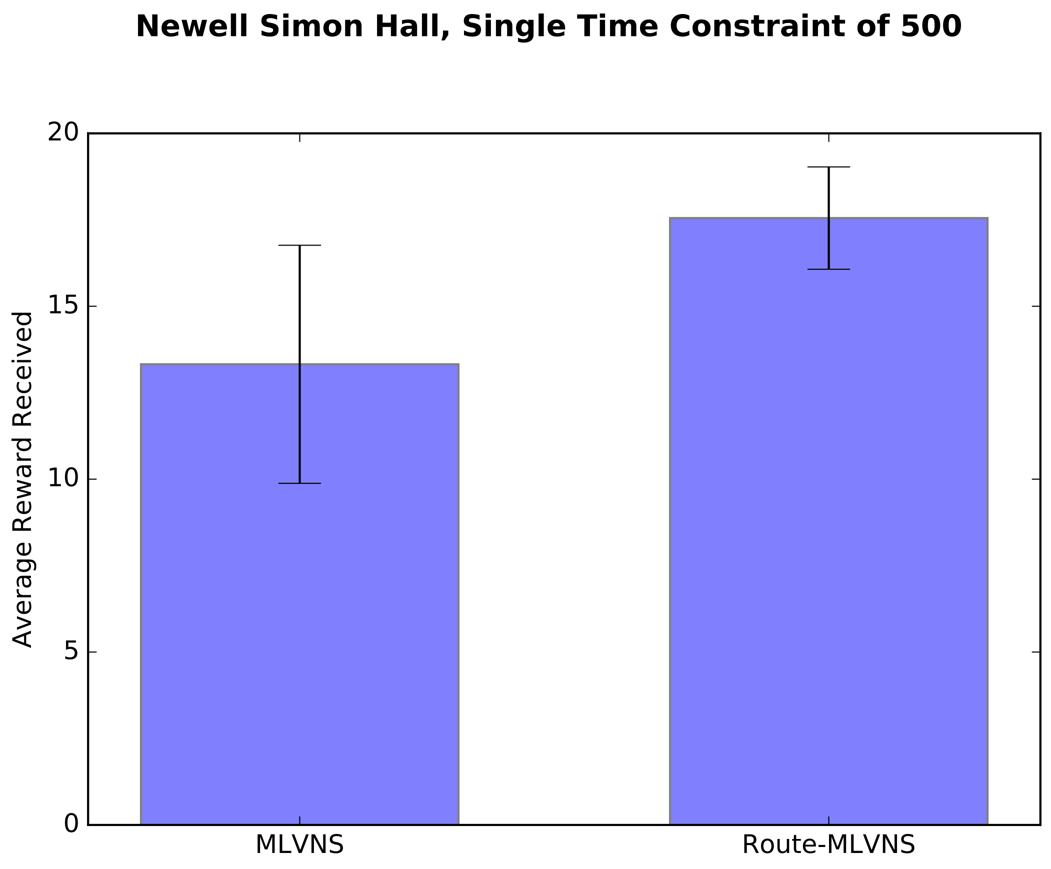


Figure 5.12: The average reward received by Route-MLVNS and MLVNS finding a plan of length 500 in the 4th floor of Newell Simon Hall.

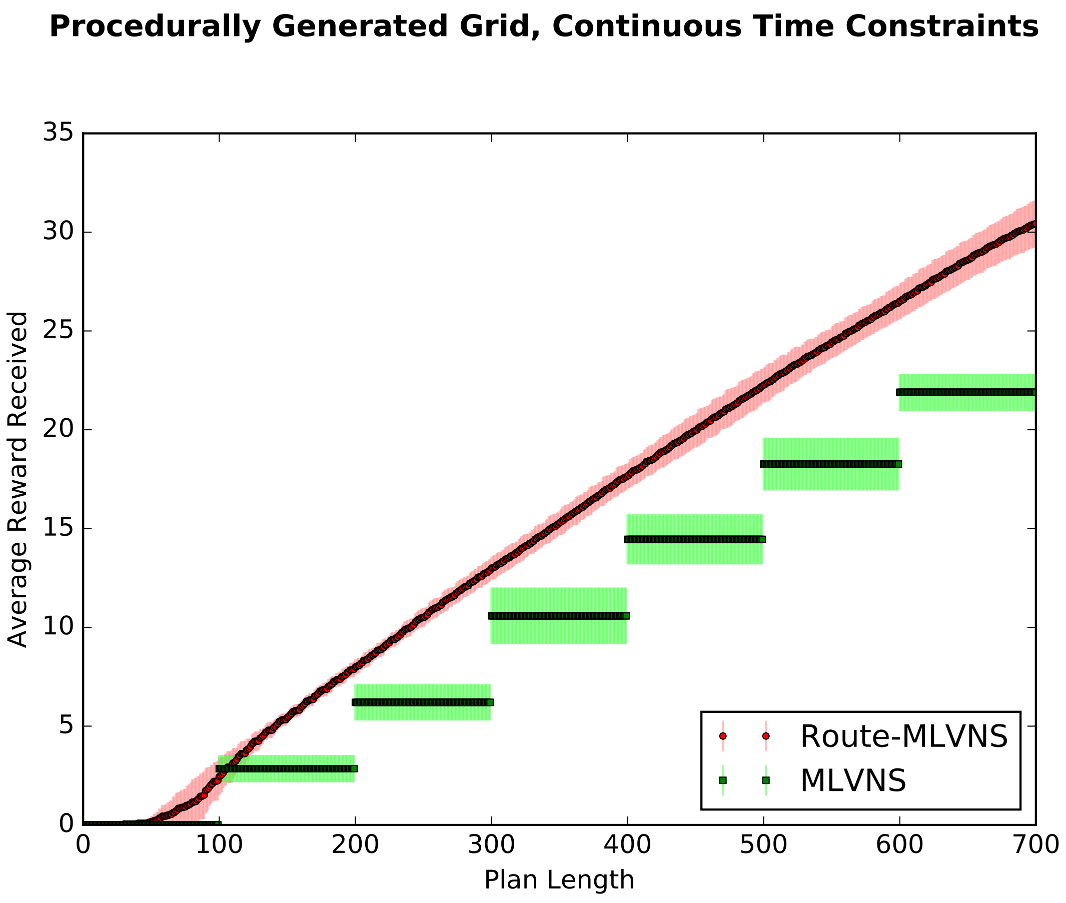


Figure 5.13: The average reward received by Route-MLVNS and MLVNS finding a set of plans for a range of length constraints in a procedurally-generated 5x5 grid.

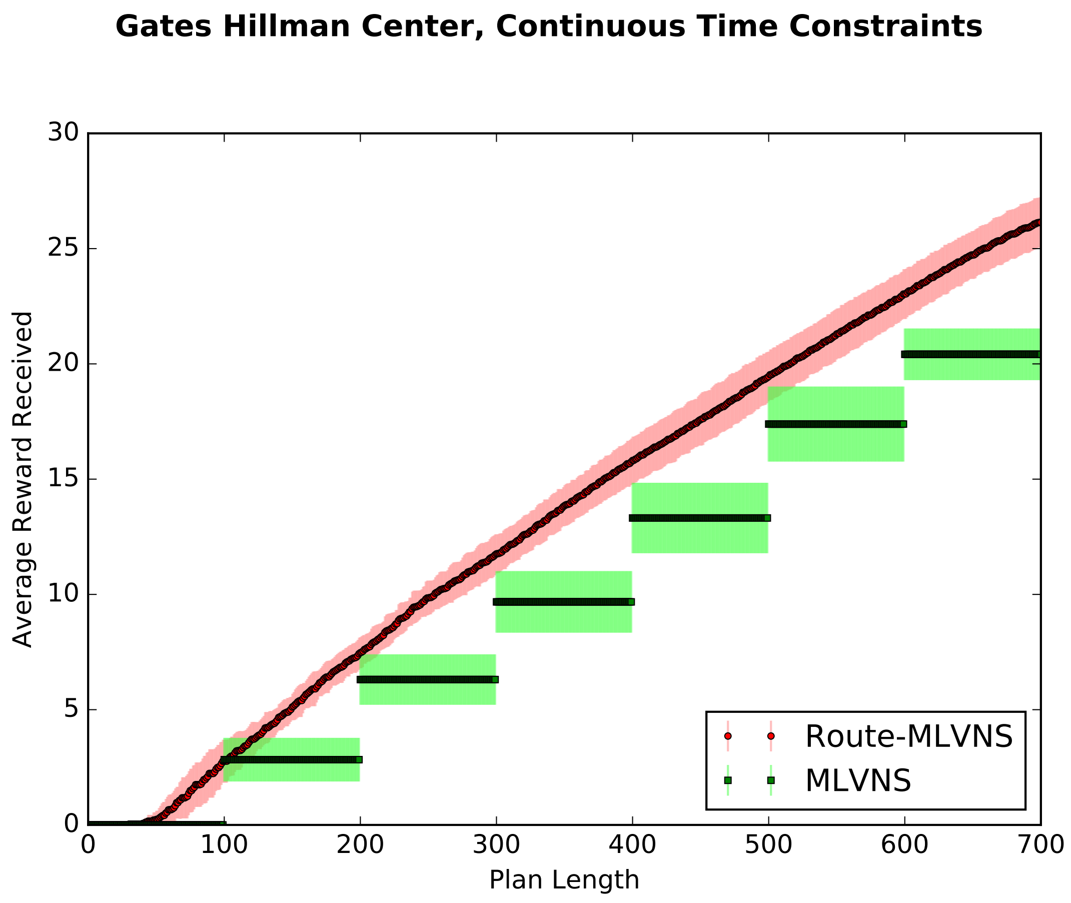


Figure 5.14: The average reward received by Route-MLVNS and MLVNS finding a set of plans for a range of length constraints in the seventh floor of the Gates-Hillman Center.

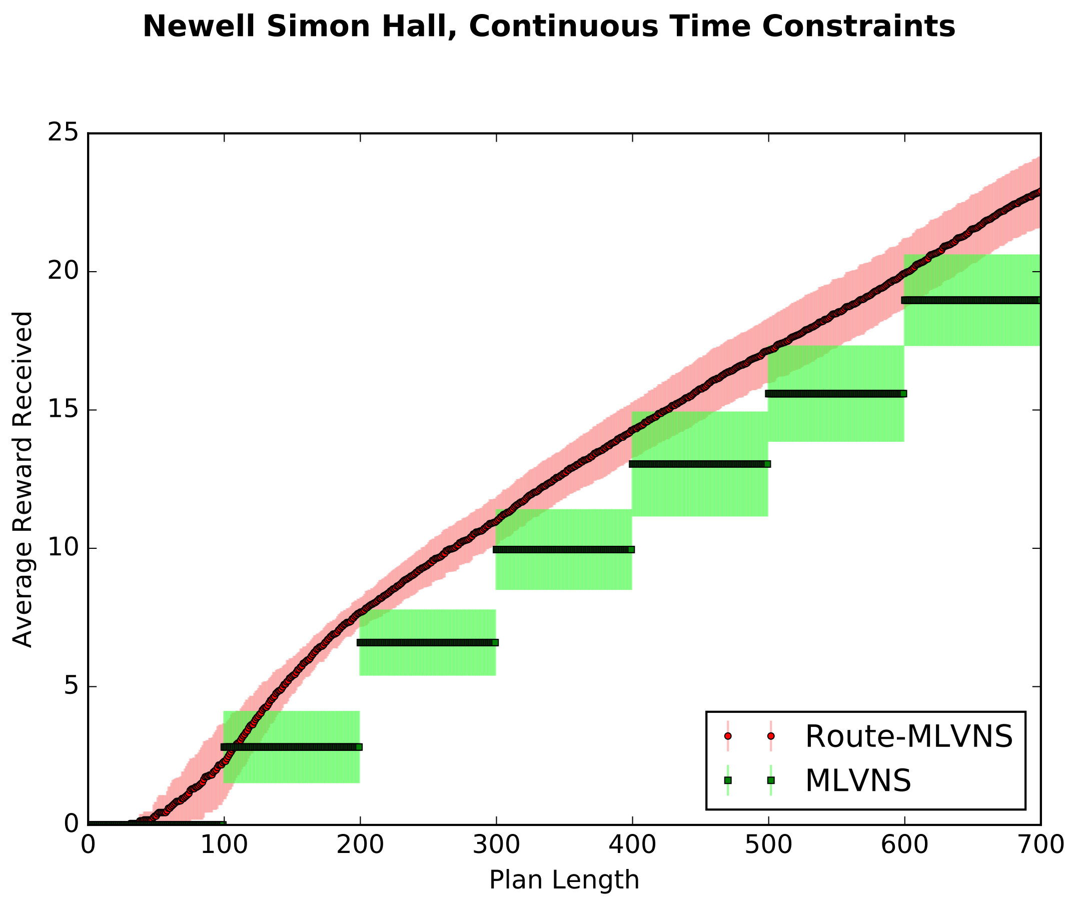


Figure 5.15: The average reward received by Route-MLVNS and MLVNS finding a set of plans for a range of length constraints in the fourth floor of Newell Simon Hall.

Chapter 6

Learning Reward Models

The the previous chapters, the agent knew the expected reward of each task it could perform. In Chapters **??** and **??**, the agent was given the expected reward it could receive from each location in the environment. In Chapter **??**, the reward functions for users requests were not known, but they were learned from performing exploration tasks, and the agent was given a formula for calculating the reward of an exploration task based on the data it had. The exploration and user requests were also kept separate, with clear priorities — the agent was never forced to choose between performing a task with a known high reward or one that would give it more information.

In this chapter, we consider the situation where an agent is seeking to maximize the reward it receives from tasks in the same way as in Chapters **??** and **??**, but the expected reward of each task is initially unknown. We consider a series of iterations in which the agent repeatedly solves planning problems in the same environment, and the goal is to maximize the reward received over time. In order to do so, the agent must use the results of its actions to create a model of the expected reward of its actions, and balance exploration — performing tasks that yield useful information — and exploitation — performing tasks that it believes will give a high reward.

We formalize this problem and develop a solution in which we treat it as a combination of the orienteering problem with task times and the multi-armed bandit problem. We create two new algorithms — Planning UCB1 and Planning Thompson Sampling — by adapting existing algorithms that are effective for solving the multi-armed bandit algorithm so that they can be combined with an appropriate algorithm for solving the orienteering problem with task times in order to learn a model of the environment and maximize the reward received over time[**?**, **?**].

6.1 Problem Description

We begin by formalizing the problem we are studying in this chapter. First, we will describe the problem itself, then we will describe how the structure of the environment relates to our approach.

6.1.1 Problem Formalization

We call the problem that we study in this chapter the Iterated Orienteering Problem with Unknown Stochastic Rewards (IOP-USR). Like the Exploration Scheduling problem described in Section **??**, the IOP-USR consists of repeated iterations of a planning problem within the same environment.

Each iteration of the problem is an instance of the Orienteering Problem with Task Times (OP-TT) described in Section **??**, except with unknown stochastic rewards. Like in the OP-TT, agent operates in an environment G with nodes V representing locations where it can receive reward and edges E representing the time it takes to travel between locations. The agent is given a start location v_s, an end location v_e, and a time limit t_{max}. The agent can spent a time of t_{task} to perform a task at a location to receive reward, but it cannot receive reward from the same location more than once per iteration. The goal is to find a plan that starts at v_s, ends at v_e, has length less than t_{max}, and maximizes the reward received.

However, unlike in the OP-TT, the agent is not given the expected reward of each location, and the rewards are not deterministic. Each location v_i has a reward function $0 \leq R(v_i) \leq 1$. When the agent performs a task at v_i, it receives a reward of 1 with probability $R(v_i)$, otherwise it receives a reward of 0. The agent begins with no knowledge of $R(v_i)$ for any location. To successfully maximize the reward received, the agent needs to use the results of its past tasks to learn a model of the expected reward $R(v_i)$ for each location.

The learning component of the problem resembles a multi-armed bandit problem. In a multi-armed bandit problem, an agent has a number of levers it can pull. Each step, it can pull a lever, and when it does, the lever yields a reward from a random distribution. The goal is to maximize the reward received over time. To do so, the agent must use the results when it has pulled each lever in the past to estimate the expected reward. The challenge comes in balancing learning and receiving reward. If the agent focuses purely on performing the tasks with the most reward based on its past results, it risks ignoring tasks that it believes have a lower reward but actually have a higher reward. If it focuses purely on gathering information and learning an accurate model, it never takes advantage of the model to actually improve the reward receiving.

In a classic multi-armed bandit problem, however, the agent can pull any lever it wants in each step. In our scenario, the constraints of the planning problem complicate things considerably. The need to travel to a "lever" in order to receive reward from it, the need to reach the specified end location within the given time limit, and the inability to receive reward from the same location more than once per iteration all significantly change the nature of the problem.

6.1.2 Environment Structure

As with the previous chapters of this thesis, we assume the environment G that the agent operates in is structured. Once again, we specifically focus on environments with hallways and bottlenecks, as defined and discussed in Chapter **??**. Unlike in previous chapters, however, in this chapter we do not seek to specifically exploit the structure of the environment when solving the problem. We do, however, consider the structure of the environment in two ways.

First, we seek a solution to the problem that does not prevent the exploitation of the structure of the environment. This problem has two components: the planning component resembling the OP-TT, and the learning component resembling a multi-armed bandit. In previous chapters of this thesis, we presented algorithms for solving the Orienteering Problem with Task Times by exploiting the structure of the environment. We would like to address the learning component of the problem in a way that is still compatible with those methods. More specifically, our goal is to address the issue of unknown stochastic rewards in such a way that each iteration of the IOP-USR can be treated as an ordinary OP-TT problem. Doing so would then allow an OP-TT algorithm to be chosen that exploits the structure of the environment, if appropriate.

Second, the structure of the environment does have an impact on the distribution of the samples that the agent will receive when solving the learning component of the problem. We mentioned this impact briefly in Section **??**, but we will discuss it in more detail here. The algorithms we contribute in this chapter do not seek to exploit these properties, but we will consider them when analyzing our experimental results.

Opportunities and Hallways

Whenever the agent's plan includes traveling past a location v_i, it can perform a task at that location cost of only t_{task}, with no additional costs related to traveling. In Section **??** we said that v_i was "on the way" in that plan. When a node is on the way for the agent's plan, we will call it an **opportunity** for v_i.

As before, we define a hallway as a set of nodes $\{v_0, v_1, \dots, v_{n-1}, v_n\}$. where each node $v_i \in \{v_1, \dots, v_{n-1}\}$ has exactly two edges, one connecting it to v_{i-1} and one connecting it to v_{i+1}, while v_1 and v_n have either exactly one edge (a dead end) or three or more edges (an intersection). An important property of hallways is that any plan that travels through a hallway contains opportunities for every location in the hallway. If the environment is comprised primarily of hallways, then any path between two locations in the environment will traverse a number of hallways. As a result, there are a very large number of locations on the way for any plan.

Additionally, opportunities in an environment comprised of hallways and bottlenecks are often dependent and non-uniform. They are dependent because, if one location in a hallway is on the way for the agent's plan, other locations in the hallway typically will be too. They are non-uniform because the agent will typically have more opportunities to perform tasks at locations on or near bottlenecks.

We expect the dependence and non-uniformity of opportunities in an environment with hallways and bottlenecks to affect the data that an agent is able to gather when operating in such an environment. In an ordinary multi-armed bandit problem, the agent can pull any lever it wants. In the IOP-USR, the tasks it performs will be affected by the structure of the environment. The more opportunities it has to perform a task, the more often it will likely include that task in its plan, and the more data it will have to learn its model of the expected reward of that task. Thus, the data the agent acquires over the course of many planning iterations is likely to be dependent and non-uniform as well.

We will now present two examples of simple hallway-based environment structures and discuss the dependent, non-uniform nature of opportunities in them in more detail.

Example: Hallway Sequence Environment

Consider the environment shown in Figure **??**. This structure, which we call the "Hallway Sequence," contains a sequence of intersections which are connected by varying numbers of hallways. The first set of hallways acts as a choice between non-overlapping paths as described in Section **??**, and the last hallway acts as a bottleneck as described in Section **??**, since the agent must still always pass through that hallway to get from one end of the environment to the other.

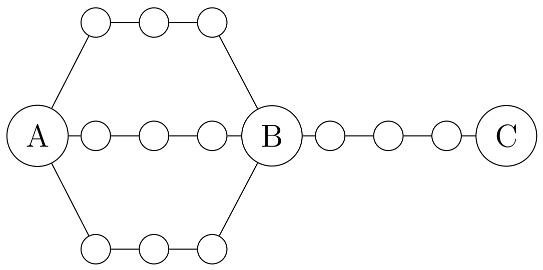
Task opportunities in this environment are dependent because the opportunities for locations on each hallway are correlated with each other. Each time the agent travels down a hallway, it will get an opportunity to perform a task at every location on the hallway. Furthermore, the opportunities at the center of a hallway are directly related to the opportunities at one end. In order to reach the center node in one of the hallways, the agent must pass by at least one of the nodes on either side of it. Thus, the number of opportunities the agent has at the center node will always be less than or equal to the total number of opportunities it has at the two adjacent nodes.

Opportunities in this environment are non-uniform because the agent will always receive more opportunities for locations in "bottleneck" hallways with fewer or no alternatives. In the example in Figure **??**, there is only a single hallway connecting nodes B and C. Every time the agent travels from B to C or vice versa, it will have opportunities to perform tasks at all three nodes on the hallway. On the other hand, there are three hallways connecting nodes A and B. When the agent travels between those nodes, it will only have opportunities at the nodes in one of the three hallways. If the agent travels back and forth repeatedly between A and C, then the number of opportunities it will have at the nodes between B and C will be three times the average opportunities at each node between A and B, which will likely result in the agent learning a much more accurate model of the rewards of tasks between B and C than it will learn of the rewards of tasks between A and B.

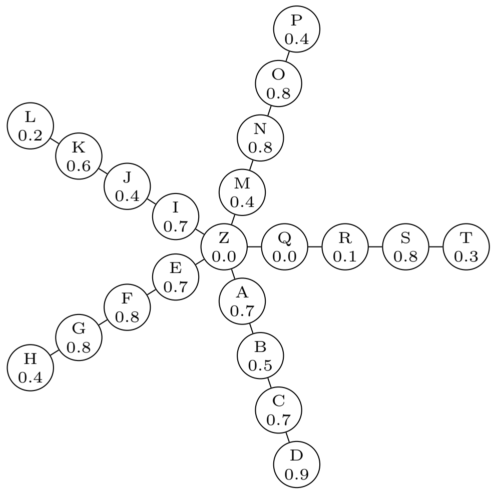
Example: Star Environment

A second example environment structure, which we call the "Star," is shown in Figure **??**. In this structure, there is a single center room acting as a bottleneck, labeled "Z" in the image, with a number of hallways branching out from it.

Like the hallway sequence environment, opportunities for an agent in this environment are dependent and non-uniform. Both the dependence and non-uniformity come from the fact that traveling to the far end of a hallway requires passing every other location on the hallway. The agent cannot travel from node Z to node D without having opportunities



(a) An example of a Hallway Sequence environment with three intersections, labeled A, B, and C, three hallways between A and B, one hallway between B and C, and three non-intersection nodes per hallway. The hallways between A and B serve as a non-overlapping choice, while the hallway between B and C acts as a bottleneck.



(b) An example of a Star environment with five hallways, each containing four nodes (not including the center). The center node Z acts as a bottleneck that must be passed through any time the agent travels from one hallway to another. Rewards for each node are included for a later example.

Figure 6.1: Environment structures with non-uniform, dependent opportunities. These are also examples of the environment structures used in our experiments to evaluate the Planning Thompson Sampling and Planning UCB1 algorithms.

at nodes A, B, and C. The result is that the farther a node is from the center, the fewer opportunities we would expect the agent to have there.

In fact, if the start and end location of the planning problem are never in the same hallway, then it is guaranteed that the number of opportunities the robot has at any node will be less than or equal to the number of opportunities at the adjacent node closer to the center. The number of opportunities at node D is guaranteed to be less than or equal to the number of opportunities at node C, because every plan including node D must also include an opportunity at node C because the agent must pass C to get between D and any other hallway. Under these conditions it is also impossible for any node in the environment to have more opportunities than Z.

6.2 Algorithms

In this section, we contribute two algorithms, "Planning Thompson Sampling" and "Planning UCB1," for solving the Iterated Orienteering Problem with Unknown Stochastic Rewards. We first describe the original Thompson Sampling and UCB1 algorithms used in standard multi-armed bandit problems, and how they can be used by an agent constrained by time and location in a structured environment to naively generate solutions to the IOP-USR[?, ?, ?]. We then describe our own planning versions of those algorithms, which plan ahead in order to better take into account the constraints on the agent and solve the IOP-USR more effectively.

In our approach to this problem, we treat each iteration of the IOP-USR independently. The problem takes place over a number of iterations, with each iteration being its own planning problem with a new start location v_s, end location v_e, and time limit t_{max}. The algorithms we present in this section are each for solving a single iteration, and we solve the problem as a whole by simply using the algorithm each iteration, with the only input from previous iterations being the data that the agent has acquired from performing tasks in the past.

6.2.1 Naive Thompson Sampling and Naive UCB1

Thompson Sampling and UCB1 are both existing algorithms that have been shown to be effective for solving the multi-armed bandit problem. Both of them can be implemented as a process that repeatedly assigns a value to each lever in the multi-armed bandit, and then pulls the lever, receives a reward, recalculates the assigned values, and repeats the process.

We can apply this process directly to an iteration of the IOP-USR. We call this process the "naive algorithm." Pseudocode for the Naive Algorithm is shown in Algorithm **??**. It has five main steps:

1. Use a given function $R(v)$ to assign a value to each function in the environment.
2. Choose the location with the highest assigned value that can be received without violating the constraints of the planning problem (lines **??** and **??**).
3. Add a task at the chosen location to the plan and update the current time and location (lines **??** through **??**).
4. Repeat steps 1-3 until it is not possible to perform any more tasks in the time remaining (line **??**).
5. Add the end location to the plan (line **??**).

We call this algorithm "naive" because it only takes into account the constraints on time and location when considering whether or not an action is allowed, without considering how the constraints affects its future actions. We implement two versions of it using different reward functions $R(v)$, one based on the Thompson Sampling multi-armed bandit algorithm and one based on the UCB1 multi-armed bandit algorithm. We call these algorithms **Naive Thompson Sampling** and **Naive UCB1**, respectively. Both follow the same steps in Algorithm **??**, differing only in how they calculate the given reward function $R(v)$.

Algorithm 12 The naive learning algorithm

```
1: procedure NaiveLearning($G$, $v_s$, $v_e$, $t_{max}$, $t_{task}$, $R$)
2:     $p \leftarrow [v_s]$
3:     $v_{last} \leftarrow v_s$
4:     $t \leftarrow 0$
5:     repeat
6:         $V_{possible} \leftarrow v \in V$ such that $D(v_{last}, v) + t_{task} + D(v, v_e) \leq t_{max} - t$
7:         $v_{next} \leftarrow \underset{v \in V_{possible}}{\text{argmax}} R(v)$
8:         Append $v_{next}$ to $p$
9:         $t \leftarrow t + D(v_{last}, v_{next}) + t_{task}$
10:        $v_{last} \leftarrow v_{next}$
11:    until No location is added to $p$
12:    Append $v_{end}$ to $p$
13:    return $p$
```

Thompson Sampling

Thompson sampling is a probability-matching algorithm in which the agent randomly chooses an action such that the probability of an action being chosen is proportional to the probability that the action is optimal, given the information available[?, ?]. This is accomplished by sampling from the distribution of possible expected rewards based on the samples the agent has gathered from a location.

For any location v_i, the function $B_i(n_{1,i}, n_{0,i})$ gives the probability distribution of the true expected reward of performing a task at that location given that the agent has received a reward of 1 from tasks there $n_{1,i}$ times and 0 from tasks there $n_{0,i}$ times, where $B(x, y) = \frac{(x-1)!(y-1)!}{(x+y-1)!}$ is the beta function. In our work, if $n_{1,i}$ or $n_{0,i}$ was 0, then a value of 0.5 was used instead. To assign rewards in step 1 of the naive algorithm, the algorithm samples a reward $r'_i \sim B_i(n_{1,i}, n_{0,i})$ for each location v_i and assigns the sampled reward to that location $R(v_i) = r'_i$. As a result, the probability of location v_i having the highest sampled reward, and thus being chosen in Step 2 of the naive algorithm, is equal to the probability that v_i has the highest true expected reward of any location given the rewards the agent has received from tasks in the past.

In the ordinary multi-armed bandit problem, Thompson sampling does not necessarily provide any guaranteed bounds on regret. Nonetheless, it has been show to be competitive with other multi-armed bandit algorithms.

UCB1

UCB1 is an optimistic algorithm that intentionally overestimates the expected reward of a location based on the uncertainty of its model of that reward[?]. That way, the agent will be biased towards gathering data from locations for which the reward is highly uncertain in order to improve its model.

Ordinarily UCB1 initializes by pulling each lever once in a multi-armed bandit problem. This is not necessarily possible in the IOP-USR, due to the agent's ability to perform a task being constrained by time and location, so instead we initialize by assigning a reward of $R(v_i) = 1$ to any location at which the agent has not yet performed a task. Once the agent has performed a task at a location v_i, it assigns the location a value of $R(v_i) = r_i + \beta\sqrt{\frac{2\ln n}{n_i}}$, where $r_i = \frac{n_{1,i}}{n_i}$ is the average observed reward from that location, n is the total number of tasks the agent has ever performed at any location, n_i is the number of tasks the agent has performed at location v_i, and β is a tuning parameter.

In the ordinary multi-armed bandit problem with $K > 1$ levers, the expected regret of UCB1 is at most

$$\left[8 \sum_{i:\mu_i<\mu*} \left(\frac{\ln n}{\Delta_i}\right)\right] + \left(1 + \frac{\pi^2}{3}\right)\left(\sum_{j=1}^{K} \Delta_j\right), \tag{6.1}$$

where μ_i is the expected reward of lever v_i, u^* is the maximum expected reward of any lever, n is the total number of lever pulls, and $\Delta_i = \mu * - \mu_i$[**?**]. However, this does not apply to the IOP-USR. Due to the constraints imposed on the agent by the planning component of the problem, the cost of pulling a lever in not uniform, so the same regret does not necessarily apply. Nonetheless, the core concept of overestimating the reward of nodes based on the number of samples is promising and we believe it is worth testing in the context of the IOP-USR despite the lack of guarantees on regret.

6.2.2 Planning Thompson Sampling and Planning UCB1

The naive algorithm represents an algorithm applying multi-armed bandit algorithms to the IOP-USR in the most direct way possible. Because the multi-armed bandit problem does not have any constraints on which levers the agent can pull or how frequently it can pull them, the naive algorithm does not take any of the constraints on time or location on the agent into account, except to ensure that it does not violate them.

We now contribute a new algorithm that plans ahead, taking into account how the constraints that the agent faces affect its future actions over a full iteration of the IOP-USR instead of only its next action. This algorithm is designed to address the fact that the constraints on the agent not only restrict the actions available to it at any given moment, but also the future actions that will be available after it performs a task. We call this algorithm "the planning algorithm."

The planning algorithm has the following steps:

1. Use a given function $R(v)$ to assign a value to each function in the environment.

2. Compute a plan that maximizes the assigned value received while reaching the end location in time.

3. Execute the computed plan.

Step 1 of this algorithm is the same as step 1 of the naive algorithm. The agent assigns reward $R(v_i)$ to each location v_i using an appropriate algorithm. As with the naive algorithm, the two functions we test in this chapter are Thompson Sampling and UCB1, as described in section **??**. We call the two versions of the algorithm "Planning Thompson Sampling" and "Planning UCB1."

However, unlike the naive algorithm, which always travels to the location with the highest assigned reward to perform a task as long as it doesn't violate the constraints of the planning problem, the planning algorithms plan ahead. Specifically, they treat the problem of maximizing the assigned value within the time and location constraints as an instance of the Orienteering Problem with Task Times.

The method used to compute the plan can be any algorithm that solves the OP-TT. In this way, it is much like the *RgnEval* function used in the RegionPlan algorithms as described in Section **??**. Much like the *RgnEval* function of the RegionPlan algorithms, the method used to find a plan for the planning algorithm can be chosen to be appropriate for the situation and environment in which the problem takes place. The algorithms presented in the previous chapters of this thesis — the Task Graph algorithm RS-RegionPlan algorithm, and Route-MLVNS algorithm === could all potentially be used if they are appropriate for the structure of the environment.

In this chapter, however, we perform experiments in the Hallway Sequence and Star environment structures described in Sections **??** and **??**. In these environments, under the conditions used in our experiments, it is possible to calculate the optimal plan in a short amount of time. Thus, rather than using a heuristic or approximation algorithm in step 2 of the planning algorithm, we are able to find the optimal solution in our experiments.

Maximizing Reward in the Hallway Sequence Environment

For our tests in the Hallway Sequence environments, as described in Section **??**, the start location v_s was always the intersection on one end of the environment (A in Figure **??**), and the end location v_e was always the intersection on the opposite end (C in Figure **??**). Additionally, the time limit t_{max}, edge lengths, and time to perform a task t_{task} were all chosen so that the agent only had time to travel directly from the start location to the end location and perform tasks at a given number of rooms m along the way, without time to perform tasks at any locations not on the path taken.

With these restrictions, the possible routes the agent can take from the start location to the end location can be easily enumerated. In the environment shown in Figure **??**, there are only three possible routes from A to C, one for each of the hallways between the A and B. For each possible path, the agent can find the optimal plan of tasks to perform along the route in the same manner as the optimal task plan along a waypoint route was calculated in Section **??**, by selecting the m tasks along the route with the highest reward.

Maximizing Reward in the Star Environment

For our tests in star environments, as described in Section **??**, the start and end locations v_s and v_e were always the center of the environment (Z in Figure **??**), and the edge lengths were always integers. This allowed us to find the plan that maximized the assigned values using a mixed integer program (MIP) that can be solved in a fairly short amount of time (the Orienteering Problem can be solved with an MIP in the general case, but it is a more complex MIP that takes significantly longer to solve).

The MIP has one variable v_i for each location corresponding to whether that location is contained in the plan, and one variable h_j for each hallway corresponding to the total distance traveled by the agent along that hallway before returning to the center node. The objective function is to maximize the total reward

$$\sum_{v_i} R(v_i)v_i,$$

where $R(v_i)$ is the reward assigned to the location v_i. For each variable v_i, there is a constraint

$$D(C, v_i)v_i \leq h_j,$$

where $D(C, u_i)$ is the distance from the center node to the node v_i, and h_j is the hallway variable corresponding to the hallway that contains v_i. These constraints ensure that each hallway variable h_j accurately represents the distance the robot must travel along the corresponding hallway to visit the offices in the plan. Finally, there is one constraint ensuring that the final plan meets the time constraint:

$$\sum_{h_j} 2h_j + \sum_{v_i} v_i t_{task} \leq t_{max}.$$

The term $\sum_{h_j} 2h_j$ represents the total time the agent spends traveling, while $\sum_{v_i} v_i t_{task}$ represents the total time spent performing tasks.

The solution to this MIP represents the locations at which tasks are performed in the optimal plan. To create the actual plan from the solution, for each hallway in the environment, we add the locations where tasks are performed to the plan in order of their distance from the center node. The order in which the hallways are visited does not matter.

This MIP is easier to solve than the general one that can be used to solve the Orienteering Problem because it is not necessary to find the exact order the nodes are visited in. Instead, the MIP only needs to solve for the nodes that are included in the optimal plan, because in this environment the distance of the shortest plan that includes a given set of nodes can easily be calculated without actually determining the order of the nodes. The order can then be easily found afterwards once the nodes are known.

6.2.3 Planning and Naive Algorithm Example

We now present an example of the naive and planning algorithms. Consider an agent operating in the environment shown in Figure **??**. Assume that $v_s = v_e = Z$, $t_{task} = 10$, and

$t_{max} = 100$, and that the the assigned values $R(v_i)$ are shown in the figure.

The naive algorithm begins by choosing the node with the highest assigned value, D, and travels there to perform a task. This has a cost of 50 (40 to travel and 10 to perform the task), so it has a remaining time of 50 before it must return to Z. The only nodes it can perform a task at in that time are A, B, and C. A and C are tied with an assigned value of 0.7, so it selects one at random. Regardless of its choice, it is only left with enough time to return to Z without performing any further tasks. The total expected reward of the plan is 1.6.

The planning algorithm, on the other hand, computes the optimal plan using the MIP described in section **??**. In this case, that plan is to offer services to the nodes E, F, and G, yielding an expected reward of 2.3.

6.3 Experimental Results

We performed experiments in simulation in order to compare Naive UCB1 and Thompson Sampling with Planning UCB1 and Thompson Sampling in the star and hallway sequence environment structures. In these experiments, for each location in the environment v, the probability $R(v)$ of a task yielding a reward of 1 at that location was randomly chosen between 0.0 and 1.0.

6.3.1 Algorithms

Our experiments featured the four algorithms described in Sections **??** and **??**: Naive Thompson Sampling, Planning Thompson Sampling, Naive UCB1, and Planning UCB1. In the hallway sequence environment, a value of 0.2 was used for β for both UCB1 algorithms, while in the star environment, a value of 0.1 was used, both determined empirically.

We also tested three control algorithms. The **Greedy** algorithm only exploited and never explored. It assigned a reward to each location equal to the average reward it had received from that location in previous iterations, then found the plan that maximized the expected reward using the algorithms described in Section **??**. If the greedy algorithm had never previously performed a task at a location, it assigned a default value of 1.0.

The second control algorithm was **Random**, which always generated a random plan, effectively only exploring and never exploiting. In the hallway sequence environment, a random plan was generated by choosing a random path from the start to the end, and then randomly choosing m location along that path, where m is the number of offices the agent had time to visit. In star environments, a random plan was generated by assigning a random "reward" to each location in the environment, and then finding the plan that maximized the random reward received.

The third control algorithm was **Epsilon**[?]. Each iteration, the epsilon algorithm would use the random algorithm with probability ϵ and the greedy algorithm with probability $1-\epsilon$. The value of ϵ was given by the equation $\epsilon = \frac{\log T}{T}$, where T is the number of iterations that have passed.

6.3.2 Performance Metrics

We compared the algorithms using two primary metrics. The first metric, "reward received," is the average expected reward of the plan executed by the agent at each iteration. We used the expected reward of the agent's plan based on the probability of each task succeeding, rather than actual number of successful tasks, in order to reduce the variance in the results due to randomness. The reward received serves as a metric for how well the algorithm balanced exploration and exploitation in order to receive as much reward as possible over time.

The second metric, "best reward," is the true expected reward of the plan with the best expected reward according to the agent's model. This is computed by using the greedy algorithm to create a plan using the data gathered by the agent so far. In other words, if the agent ceased exploring and switched to pure exploitation using the greedy algorithm at any given iteration, "best reward" is the expected reward it would receive. This serves as a metric of how effective the agent's model is for choosing a high-reward plan, and thus how successful the agent has been in exploring the environment.

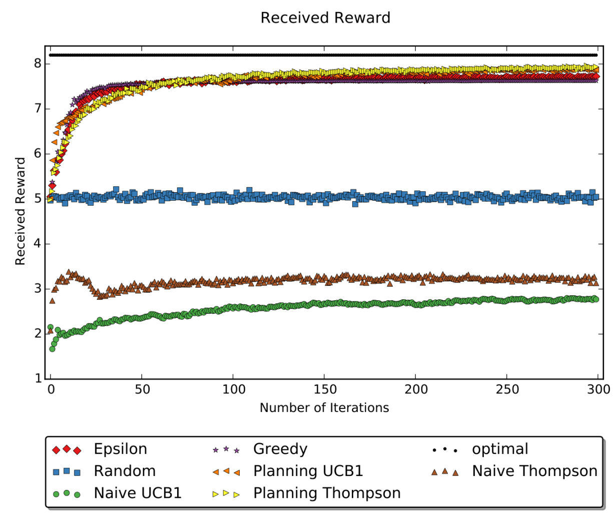
We also observed how frequently the agent performed tasks at each location when using each algorithm, and compared it to how often each location was part of the true optimal plan, computed using the algorithms described in Section **??** using the true expected reward of each location. The purpose of this metric was not to gauge performance, but to observe the interaction between the algorithms and the structure of the environment.

6.3.3 Hallway Sequence Results

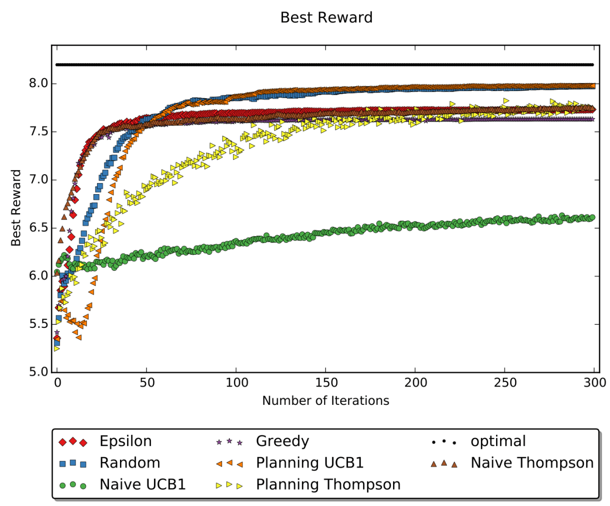
Our experiments in hallway sequence environments (described in Section **??**) ran for 300 iterations each, and our results were averaged over 200 trials. The environment consisted of three intersections, with multiple hallways connecting the first two intersections and a single hallway connecting the second to the third, like the environment shown in Figure **??**. Each hallway had ten nodes, not including the intersections. Each iteration, the agent chose a route from the first intersection to the last intersection, and had time to perform tasks at up to ten nodes along that route, but not enough time to perform tasks at any locations on other routes.

Figure **??** shows the average reward received by each algorithm over time in an environment with five hallways between the first two intersections. Greedy and Epsilon performed very well at very low numbers of iterations, but within the first 100 iterations on average were overtaken by Planning UCB1 and Planning Thompson Sampling, which performed roughly equally. Naive UCB1 and Thompson Sampling had terrible performances, receiving less reward than even the Random algorithm, but between the two the Naive Thompson Sampling received more reward. Increasing the number of hallways between the first two intersections did not significantly affect these results.

Figure **??** shows the average best reward of each algorithm. In this case, Random exploration and Planning UCB1 had the best performances. Planning Thompson Sampling had a poor best reward at small numbers of iterations, but improved dramatically over time. Its



(a) The expected reward received by the agent. Planning Thompson Sampling and Planning UCB1 had the best performances.



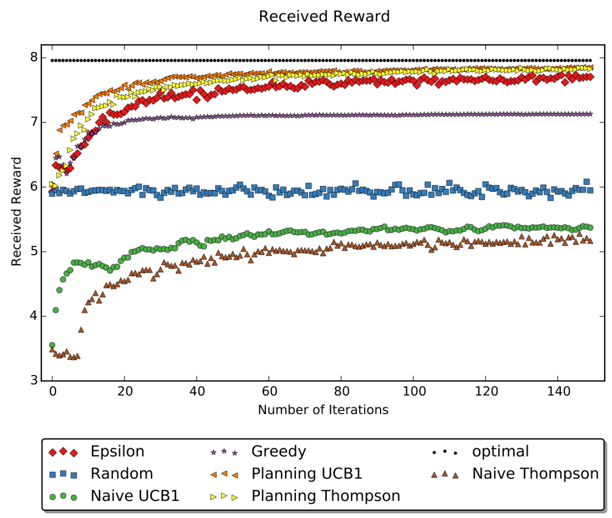
(b) The reward of the best plan according to the agent's model. Random exploration and Planning UCB1 had the best performances.

Figure 6.2: Results in a simulated IOP-USR in a Hallway Sequence environment.

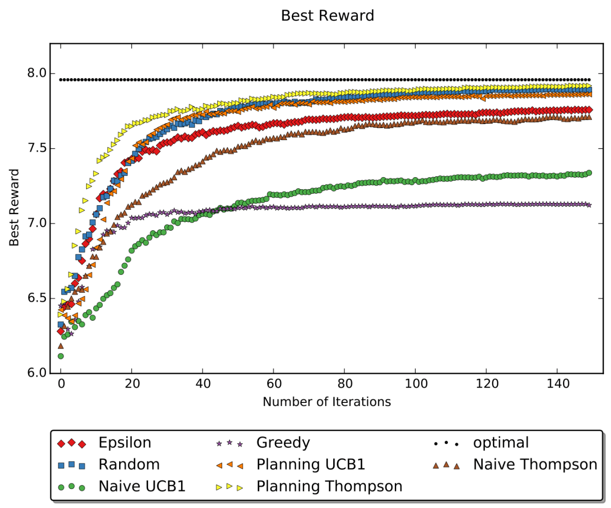
best reward continued to improve with more iterations, but did not overtake Planning UCB1 or Random. Naive UCB1 once again performed poorly, but Naive Thompson Sampling was surprisingly effective, with comparable performance to Epsilon. Increasing the number of hallways between the first two intersections caused a significant decline in the performance of the Planning Thompson Sampling algorithm by this metric, without any significant effect on the other algorithms.

We can gain some insight into our results by looking at how often each algorithm performed tasks at each location on average, which can be seen in Appendix **??**. The frequency with which Planning Thompson Sampling and Planning UCB1 visited each location in the environment corresponded roughly to how often each location was contained in the optimal plan, between 10% and 15% of iterations for most locations in the first set of hallways and between 30% and 45% of the time for locations in the final hallway. Random exploration, meanwhile, performed tasks at all of the rooms in the first five hallways in 9-10% of the iterations, and the other rooms 48% of the time. Epsilon's visiting frequencies were in between those of random exploration and the planning algorithms.

The two naive algorithms had very low visiting frequencies in general, often performing tasks at fewer than the allowed 10 locations per iteration. They had especially low visiting frequencies for locations closer to the start location. The Naive UCB1 and Naive Thompson Sampling algorithms both visited the locations in the first set of hallways between 1% and 4% of the time in most cases, with fewer visits for the closest ones. The nearest rooms in the final hallway were visited less than 10% of the time, but this increased as the rooms got farther from the start location, with the location immediately before the end location being visited in more than 50% of their plans.



(a) The expected reward received by the robot. Planning Thompson Sampling and Planning UCB1 had the best performances.



(b) The reward of the best plan according to the robot's model. Random exploration, Planning Thompson Sampling, and Planning UCB1 had the best performances.

Figure 6.3: Results in a simulated IOP-USR in a Star environment.

From these results, we can conclude that the naive algorithms frequently jumped straight to locations farther from the start location without performing services at other offices along the way. Because the scenario is set up to prevent backtracking, they missed many locations that were on the way where they could have performed tasks at a low cost. The planning algorithms, on the other hand, visited locations roughly proportionally to how frequently they had opportunities to do so, indicating that they took advantage of the "on the way" opportunities present in the environment structure.

6.3.4 Star Results

Our experiments in Star environments, as shown in Figure **??**, ran for 150 iterations each, and our results were averaged over 200 trials. The results shown in Figure **??** are for environments with six hallways, each consisting of six locations not including the center, and the edges were all of length 15, while t_{task} was 20 and t_{end} was 500. The start and end location were both always the center node. We tested environments with as few as four or as many as eight hallways, but did not see a significant change in the results.

Figure **??** shows the average reward received by each algorithm over time. As with our results in the Hallway Sequence environment, Planning UCB1 and Planning Thompson Sampling received the most reward, although Epsilon was close behind.

Figure **??** shows the average best reward of each algorithm. Planning Thompson Sampling, Planning UCB1, and Random once again had the best performance and all reached about the same near-optimal reward after 150 iterations, but Planning Thompson Sampling performed better for very low numbers of iterations. As in the Hallway Sequence environments, Naive UCB1 performed extremely poorly by this metric, while Naive Thompson Sampling performed surprisingly well, although still worse than either of the planning or random exploration algorithms.

Examining the visiting frequencies (shown in Appendix **??**), we found that locations closer to the center were part of the optimal plan more often, with those adjacent to the center node being part of the optimal plan typically between 50% and 60% of the time while the locations at the ends of the hallways were all part of the optimal plan in less than 10% of trials. Once again, Planning Thompson Sampling and Planning UCB1 visited locations with a very similar frequency to how often they appeared in the optimal plan. Random exploration also had visiting frequencies very close to the optimal plan.

The naive algorithms once again had lower overall visiting frequencies. In this case, the biggest issue was with locations close to the center. While the naive algorithms both visited locations far from the center in about 10% of plans on average, slightly higher than the frequency with which those locations appeared in the optimal plan, they visited the locations closest to the center in 15-25% of their plans, much less than the frequency with which those locations appeared in the optimal plan.

The locations closest to the center of the environment are the ones that are most often on the way in the robot's plan, since they are on the way for any plan that travels farther down the hallway. In the star environment, we can see that the naive algorithms frequently bypassed opportunities to visit on the way locations, while the planning algorithms did not, much like in the hallway sequence environment. In the case of the star environment, there is a very clear correlation between how frequently a location is on the way and how big the difference in visiting frequency is between the naive and planning algorithms. This further goes to show that the naive algorithms failed to take advantage of the presence of on the way opportunities due to the environment's structure, while the planning algorithms succeeded in doing so.

6.3.5 Analysis

In both environment structures, the planning algorithms clearly outperformed the naive algorithms according to both performance metrics. Looking at the the visiting frequencies, we can see a pattern: the naive algorithms severely under-visited certain locations, those closest to the center in the star environment and those closest to the start in the hallway sequence environment. These locations are ones that would be frequently on the way for the robot's plans. From this, we can infer that the naive algorithms' poor performance was due to them not taking advantage of the on the way opportunities to perform tasks provided by the environment structure.

One notable unexpected result is the performance of Planning Thompson Sampling according to the "best reward" metric. Most notably, it performed poorly by that metric in the hallway sequence environment, but very well in the star environment. This shows that there is a relationship between the structure of the environment and the ideal learning algorithm to use — an algorithm that performs well in one environment might not perform well in another environment. We believe this warrants further exploration in future work.

6.4 Conclusion

In this chapter, we presented the Iterated Orienteering Problem with Unknown Stochastic Rewards, a planning problem that occurs over many iterations in which each iteration resembles the Orienteering Problem with Task Times, but the agent does not know the expected reward of performing a task at each location in the environment. Instead, the agent must repeatedly solve for and execute plans in order to gather data on the results of its actions and learn a model of the expected reward of each task in order to maximize the reward received over time. In order to do this, the agent must balance exploration — gathering information that will allow it to improve its model — and exploitation — performing tasks that will yield a high reward according to its model.

To address this problem, we treated the tasks the agent could perform as if they were levers in a multi-armed bandit problem. We contributed two algorithms to solve the problem, Planning Thompson Sampling and Planning UCB1. These algorithms are based on the existing UCB1 and Thompson Sampling algorithms used in standard multi-armed bandit problems, but modified to plan ahead over an entire iteration of the IOP-USR instead of only choosing a single action at a time. We compared our planning algorithms to "naive" ones that treated the problem more directly as an instance of a multi-armed bandit problem in simulation. Our results showed that our planning versions of the algorithms performed better in terms of both reward received and effectiveness of the model learned. Analysis of the locations visited by each algorithm showed that the naive the algorithms were not taking advantage of cases where locations could be conveniently visited on the way to another location, indicating that our planning algorithms worked better within the structure of the environment.

Chapter 7

Related Work

In this chapter, we discuss other work that relates to our research. We will begin by discussing research that has been done on the Orienteering Problem and its variants. We will then discuss existing research on Multi-Armed Bandits and other learning problems that related to our work in Chapter **??**. Finally, we will discuss work in robotics that relates to our original motivation of utilizing a service robot's spare time, highlighting in particular some work that considers elements of the structure of the environment.

7.1 The Orienteering Problem

Much of this thesis focuses on variants of the Orienteering Problem. In this section, we will first discuss research that has been done on the ordinary Orienteering Problem. We will then discuss some variants of the Orienteering Problem that have been studied that relate to our work.

7.1.1 The Regular Orienteering Problem

The Orienteering problem, sometimes also known as the Selective Traveling Salesman Problem, the Maximum Collection Problem, or the Bank Robber Problem, is a problem in which an agent operating in an environment represented by a graph seeks to gather as much reward as possible from nodes in the environment within a given time limit, starting at a given start location and ending at a given end location[**?**, **?**, **?**, **?**]. While our own work is motivated by service robots, the orienteering problem and its variants have a very wide range of other applications, including routing technicians to service customers, athlete recruiting from schools, planning routes for tour guides in cities, preventing wildfires, and, as the name would suggest, the sport of orienteering[**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

The orienteering problem was initially proposed by Tsiligirides et al., who presented several algorithms to solve it: "The Stochastic algortihm," which uses a Monte-Carlo method to randomly generate a large number of routes and picks the best, "The Deterministic Algorithm," which divides the environment into sectors formed by two concentric circles and an arc and creates plans contained in each sector, and "The Route-Improvement Algorithm," which begins with an initial route and uses simple operations to change it in search of improvement[?]. Notably, the Deterministic Algorithm follows a somewhat similar idea to our RegionPlan algorithm presented in Chapter ??, but their "sectors" are notably not based on the structure of the environment, and in fact appear to assume an open, connected environment.

The orienteering problem is NP hard, and finding an optimal solution in an acceptable amount of time is often unfeasible[?]. Some research has focused on finding more efficient optimal solutions, such as using Branch-and-Cut algorithms[?, ?]. However, most often solutions to the orienteering problem are found using approximations or heuristics.

Some research on approximations for the orienteering problem has focused on cases where the start and/or end location are not fixed. Fomin et al. created a $(2 + \epsilon)$-approximation algorithm for both the ordinary orienteering problem and the time-dependent orienteering problem, in which the time to travel between two nodes depends on the start time from which the agent left the previous node, but in their work there was not a fixed start or end location[?]. Instead, the agent's goal was to either find any path through the environment (with any start and end location), or a cycle (with the same start and end location). Blum et al. found a 4-approximation algorithm for the rooted orienteering problem, in which only the start location is given, as well as showing that the problem is APX-hard[?].

A 3-approximation algorithm for the point-to-point orienteering problem, in which both the start and end location are given, was finally found by Bansal et al.[?]. This was eventually improved on by Chekuri et al., who found a $(2 + \epsilon)$-approximation for the point-to-point orienteering problem, as well as approximations for the orienteering problem in directed graphs and the orienteering problem with time windows (in which reward from each node can only be received at during a specific window of time during the planning interval)[?].

While approximations are useful for the guarantees they provide, even Chekuri et al.'s approximation algorithm can only guarantee a solution with a reward of just under half of the optimal reward. Thus, much research has focused on finding heuristics for the orienteering problem that do not offer specific guarantees, but may perform better than existing approximations in practice.

One of the most common approaches to finding solutions to the orienteering problem is to initialize a plan or set of plans and then perform local search of the solution space around the initial plans seeking improvements[?]. The Route-MLVNS algorithm we contributed in Chapter ?? falls into this category. One of the early benchmark algorithms for solving the Orienteering problem in this manner was Chao et al.'s "Five Step Algorithm" which initializes with a simple greedy algorithm, and then repeatedly searches for better plans using three different operations (a "two-point exchange," a "one point movement," and a cleanup step using 2-opt) with a re-initialization step to avoid getting stuck at a local maximum[?, ?].

Of particular relevance to our research is the Multi-Level Variable Neighborhood Search (MLVNS) algorithm created by Liang et al., on which our Route-MLVNS algorithm presented in Chapter ??[?] was based. Ordinary Variable Neighborhood Search algorithms are based around the idea of searching various local neighborhood structures of an existing plan[?, ?]. MLVNS expands on the idea by generating and searching the neighborhood of multiple plans at once with different time constraints. To our knowledge, MLVNS is the only algorithm specifically designed to solve the orienteering simultaneously for multiple different time constraints.

Another very relevant algorithm is the Adaptive Variable Neighborhood Search presented by Santini et al.[?]. The algorithm combines clustering the Dbscan density-based clustering algorithm with a VNS approach in which the neighborhood operations consist of deleting sections of plans and then repairing them[?]. The algorithm is specifically designed for cycles — scenarios in which the start and end location are the same — as opposed to our work which allows for the start and end location to be different. The intuition behind the use of clustering, however, resembles the intuition behind our Route-MLVNS algorithm: the local search operations should be able to remove or add multiple locations to a plan at a time, and when the agent travels to a location that is distant from the start and end location, it should consider other, nearby locations as well.

Other approaches that follow a similar pattern of generating an initial plan and performing a local search include Greedy Randomized Adaptive Search Procedures (GRASP) and Tabu search[?, ?, ?]. Research has also been done in solving the orienteering problem with genetic algorithms, ant colony optimization algorithms, and partical swarm algorithms[?, ?, ?, ?, ?, ?].

7.1.2 Variants of the Orienteering Problem

Many variants of the orienteering problem have also been studied. The most common is the Team Orienteering Problem, in which there are multiple agents seeking to find a set of plans that maximize the combined reward received. Xu et al. proposed a $(1 + (1/e)^{\frac{1}{2+\epsilon}})$-approximation for the team orienteering problem[?]. Many of the types of approaches that have been applied to the single-agent orienteering problem have also been applied to the team orienteering problem, including particle swarm algorithms, branch and cut algorithms, genetic algorithms, and an augmented large neighborhood search algorithm[?, ?, ?, ?, ?, ?, ?, ?].

A variety of other, more specialized variants of the orienteering problem have also been studied. The multi-profit orienteering problem studied by Kim et al. could be seen as a generalization of the Orienteering Problem with Task Times that we studied in this thesis[?]. In it, the agent must spend time at a location to receive reward, but rather than spending a fixed amount of time, the agent can receive more reward the longer it spends at the location. Kim et al. provided a two-step hybrid solution that first chose the nodes to visit using a simulated annealing approach, followed by a visiting time optimization algorithm to choose how long to spend at each location.

The probabilistic orienteering problem is another one related to our research. Like our work in Chapter ??, the probabilistic orienteering problem is one in which locations yield stochastic rewards. However, in our work, the agent sought only to maximize the expected reward $R(p)$, and there was no difference in cost between succeeding or failing at a task. In the probabilistic orienteering problem, the agent seeks to maximize a profit $R(p) - c * L(p)$, where c is a given constant and $L(p)$ is the length of the plan. The agent must commit to a plan before it starts executing it, but after committing to the plan, it discovers which tasks will succeed and which will not, and does not have to spent time travelling to the tasks that will not succeed. Angelleli et al. presented two branch-and-cut algorithms for solving the probabilistic orienteering problem, one that finds the optimal solution and a second, faster metaheuristic one[**?**]. Meanwhile, Chou et al. solved it with a tabu search algorithm[**?**].

The set orienteering problem is a variant that is a less general version of the problem of planning over a task graph in a single exploration interval when using the Task Graph algorithm we contributed in Chapter **??**. In the set orienteering problem, locations are grouped into sets. Each set has a single reward, which can be acquired by visiting any location in that set, and can only be acquired once. In the Task Graph algorithm, in the case in which any group of mutually exclusive tasks all have the same reward, planning over the tasks is an instance of the set orienteering problem with task times and a directed graph. In our experiments with the Task Graph algorithm, the intent was that the task graph constructed be simple enough that sophisticated algorithms are unnecessary to find a good solution. However, Carrabs et al. provide a biased-key genetic algorithm approach that solves it effectively and could be useful if applying the Task Graph algorithm to scenarios in which the Task Graph is more complex[**?**].

7.2 Reinforcement Learning

In Chapter **??**, we considered a reinforcement learning scenario in which an agent performing repeated iterations of the orienteering problem must balance "exploration" and "exploitation" in order to maximize the reward received over time in an environment with unknown stochastic reward functions.

We are unaware of any previous research studying the issue of exploration versus exploitation specifically in the context of an iterated orienteering problem. However, a significant amount of research has been done on other related reinforcement problems.

7.2.1 Multi-Armed Bandits

In a multi-armed bandit problem, an agent has a number of "levers" it can pull. Pulling a lever has a fixed cost C, and each lever yields a stochastic reward form some function. The agent begins with no knowledge of the reward functions for each lever, and the goal is to maximize the reward received within a certain cost limit.

In Chapter **??**, we adapted two methods of solving a multi-armed bandit algorithm for use in the Iterated Orienteering Problem with Unknown Stochastic Rewards (IOP-USR). First, we adapted Thompson Sampling, a probability-matching algorithm which computes the probability distribution of possible true expected rewards for each lever based on the observations the agent has made. Thompson sampling itself is actually a very old concept[**?**, **?**]. Nonetheless, it has shown to be a useful approach even now, and Bayesian statistics that have been developed since the original algorithm allow it to be applied to a wider variety of probability distributions than before[**?**, **?**, **?**].

The other algorithm we adapted in Chapter **??** was UCB1[**?**, **?**]. The core idea behind UCB1 is to assign a higher reward to levers for which the agent has fewer samples in order to encourage exploration. In an ordinary multi-armed bandit problem, UCB1 provides theoretical guarantees about the regret of the reward received, although these guarantees do not necessarily apply in the IOP-USR. Other variants of UCB1 following a similar principle also exist and may be worth considering in future work on the IOP-USR, such as UCB-V[**?**, **?**].

Other approaches also exist that could potentially be well-suited for use with the planning algorithm we contributed in Chapter **??**. For example, Interval Estimation is a type of algorithm that estimates a confidence interval of the expected reward of a lever, and treats the reward as if it were the upper bound of the interval[**?**, **?**]. Because it addresses exploitation versus exploration by assigning a value to each lever and choosing the one with the highest assigned value, it could easily be used with our planning algorithm.

In some variants of the multi-armed bandit problem, such as the parametric multi-armed bandit problem, the rewards of the levers are not independent from each other[**?**]. An instance of the IOP-USR with the same property could also exist. However, as long as approaches to solving these variants can still be represented as assigning a value to each lever and choosing the lever with the highest assigned value, they are still compatible with our approach to the IOP-USR.

Another notable variant of the multi-armed bandit problem is the multi-armed bandit with metric switching costs described by Guha et al.[**?**]. In this scenario, there is a cost for the agent to switch to pull a different lever based on the lever it most recently pulled. This problem can represent a situation in which the agent's ability to pull levers is constrained by the need to travel through the environment, just like the IOP-USR. However, unlike in the IOP-USR, in the multi-armed bandit with metric switching costs, the agent can repeatedly pull the same lever, and is not constrained by the need to reach a specific end location at a specific time. Overall, the additional constraints of the IOP-USR dramatically change the problem.

7.2.2 Markov Decision Processes

Reinforcement learning situations in which the an agent can perform actions that yield different reward depending on its current state are often represented by Markov Decision Processes (MDP)[**?**, **?**, **?**]. An MDP consists of a set of states S, a set of actions A, a reward function $R : S \times A \rightarrow \mathbb{R}$ that gives the reward received for performing an action a in a state s, and a transition function $T : S \times A \rightarrow \Pi(s)$ that gives the probability of the state changing to any given state s' when the action a is performed in state s.

The IOP-USR could, theoretically, be represented as an MDP. The state would need to include the agent's current location, the end location, and the remaining time until the agent needs to be at the end location. Doing so, however, would be impractical. The space of possible remaining times is continuous, and even if we represented it discretely, the number of possible states would scale hugely with the time limit and the size of the environment.

Meanwhile, the reward function itself is relatively static, only being influenced by the location where the agent performs a task and whether or not it can travel to the end location in time. Ultimately, representing the problem as an MDP would result in the size and complexity of the reward and transition functions being much larger and more complicated than we believe necessary.

Variants of MDPs designed for handling problems like the IOP-USR with very large state spaces with different independent variables, such as Hierarchical MDPs or Factored MDPs, do exist, and could potentially be used as an alternative way to solve the IOP-USR from our own approach[?, ?, ?]. However, we chose our own approach of treating the problem as independent orienteering and multi-armed bandit problems due to the way it naturally reduces the problem into two smaller ones, as well as the flexibility of letting the orienteering algorithm and multi-armed bandit algorithm be chosen independently according to the needs of the problem (such as choosing an orienteering algorithm that exploits the structure of the environment).

7.3 Service Robots

The focus of this thesis is on planning in structured environments, rather than service robots in particular. However, as service robots are the original motivation behind our research, we will discuss some of the existing work that has been done on service robots for which our research might be useful.

7.3.1 Navigation

Our work revolves around an agent that must effectively navigate and travel through an environment represented by a graph. In our research, we focus on the agent choosing which locations in the environment it will travel to, and simply assume that the agent is capable of getting there, but navigating an environment can be a challenging problem itself.

One of the most basic requirements of navigating an environment represented by a graph is the ability to find the shortest path between any two nodes. This can be accomplished by a variety of algorithms such as A*, Weighted A*, or Dijkstra's Algorithm[?, ?, ?]. Depending on the exact circumstances, different algorithms may apply, such as D* in dynamic environments or M* in multi-agent problems[?, ?].

Another problem that arises for robots navigating an environment is localization: the robot must be able to accurately determine where, in the environment, it is. Localization can use a wide variety of sensors depending on the robot, including regular cameras, depth cameras, wifi signals, GPS systems, odometers with wheel encoders, and even humans answering questions[?, ?, ?, ?, ?, ?, ?, ?]. The exact sensors and algorithms used can depend on the robot and its environment. For example, in many environments a Markovian algorithm such as a particle filter may be appropriate, but in environments where some features are not static, such as chairs and tables in an office building, non-Markovian algorithms may be necessary[?, ?].

Ultimately, in our research we assume that the agent in question is able to reliably find the shortest path between any two locations in its environment, accurately predict the amount of time it will take to follow the path, and travel between those two points without issue.

7.3.2 CoBot

The original inspiration for our research is the CoBots, autonomous mobile service robots that perform services scheduled by users[?, ?]. Users can schedule a variety of services for the CoBots through an online interface, choosing the type of service they wish it to perform, the location to perform it, and a window of time in which they would like it to be performed. The CoBot then uses a mixed integer program (MIP) to schedule user requests in order to minimize the time from the start of scheduling to the completion of the final task[?].

While the CoBots are idle when no user task has been scheduled, they are capable of autonomously performing useful tasks such as offering potentially desired objects to users or gathering data on their environment. Notably, Wang et al. studied the ability of the CoBots to navigate their environment and gather data on the strength of wifi signals at different locations while under a time constraint[?]. In their work, Wang noted that in planning problems in which a robot is gathering continuous location-dependent data like wifi signal strengths, the environment would often be represented as a grid. However, they concluded that a graph would be a more effective way to represent the hallway-based office environments in which their experiments took place. By representing the environment as a graph, they turned the problem they were solving into a variant of the orienteering problem. They did not take the environment's hallway-based structure into account when finding a solution to the orienteering problem, however, instead using an approximation algorithm.

7.3.3 Dora the Explorer

The concept of a robot that schedules its own information-gathering tasks, as in the Exploration Scheduling problem we presented in Chapter ??, has been explored by Hawes et al. with the Dora the Explorer robot[?]. Dora the Explorer specializes in performing find-and-fetch tasks, and is capable of scanning a room to determine what objects it can find there. In order to gather information about where different objects can be found, Dora must generate its own information-gathering goals to execute in its spare time[?]. Then, when performing find-and-fetch tasks, Dora must plan the task while taking into account the possible uncertainty of the model, and being prepared to replan if its old plan is rendered invalid[?].

7.3.4 Area Sweeping

One of the observations in our work is that, in a hallway-based graph environment, the shortest path between any two nodes will typically have a number of other nodes on the way, as discussed in Section ??. While we are not aware of this property being explored in graph-based environments, it has been explored in grid-based environments. Specifically it has been considered in the continuous area sweeping problem, in which a robot repeatedly visits all locations in an area represented by a grid, seeking to observe events that yield reward. Algorithms for continuous area sweeping, such as the initial one proposed by Ahmadi et al. and a more resent Deep R-Learning approach by Shah et al., evaluate the expected reward of traveling to a location by considering not just the location itself, but others it will observe on the way while traveling[**?**, **?**, **?**]. While grid environments are very different from hallway-based graph environments, this is still a case of existing work that evaluates actions by considering the reward of not just the destination, but also locations passed on the way while traveling.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the contributions of this thesis and discuss possible avenues of future work.

8.1 Contributions

The key contributions of our thesis, as discussed in Chapter **??**, are:

8.1.1 The Task Graph Algorithm

The Task Graph algorithm is a planning algorithm that exploits the structure of the environment in which the agent is planning by grouping actions it can take into larger "tasks" based on the structure of the environment. Rather than planning over the set of actions it can perform, the agent then plans over the set of tasks.

The tasks are created based on the intuition that, in certain types of structured environments, there frequently exist actions that are only worth performing if the agent performs additional, nearby actions. For example, once an agent has chosen to travel part way down the hallway, it is most likely optimal to traverse the entire hallway. Thus, the actions the agent can perform in that hallway are grouped into a task and considered as a single unit when planning.

We tested the Task Graph algorithm in a simulated problem we call "Exploration Scheduling," in which an agent with a mix of mandatory and optional tasks must gather information by exploring the environment and use that information to improve the scheduling of the mandatory user requests. We found that the Task Graph algorithm was effective for solving the problem in an environment modeled after a real office building.

8.1.2 The RegionPlan Algorithm

The RegionPlan algorithm is a planning algorithm designed to exploit the structure of the environment an agent operates in by dividing it into sections. The agent creates a separate plan for each section, and then chooses one of them to execute, or constructs a new plan by performing portions of the created plans in sequence.

Like the Task Graph algorithm, the RegionPlan algorithm is based on the idea that, in a structured environment, once the agent has chosen to enter a certain section of the environment, it is logical to commit to staying there. For example, if the agent is choosing between multiple routes to get from one location to another in a hallway-based environment, once it has started down one route it should follow that route to completion, rather than backtracking to switch to another route. In a multi-story environment, where the agent must choose which floor it travels to, it should choose a floor with the intent to stay there and not consider plans which involve frequent switching of floors.

We tested the RegionPlan algorithm in simulation of the Orienteering Problem with Task Times, in specific environment structures. We demonstrated scenarios in which the restrictions on the agent's plan due to the RegionPlan algorithm enabled it to avoid getting stuck at certain locally optimal plans and receive increased reward.

8.1.3 Route-MLVNS

Our third approach to exploiting the structure of an agent's environment was Route-Based Task Planning. In a hallway-based environment, whenever an agent travels from one location to another, it will pass by a number of other locations along the way. The core concept of Route-Based Task Planning is that, when evaluating the possibility of traveling to a location, the agent should consider not only the reward it can receive from the destination itself, but also the potential reward it can receive from the other locations it passes along the way. We accomplish this by treating solutions not as a sequence of actions, but rather a path traveled through the environment along which tasks can be performed.

We employ the concept of Route-Based Task Planning with the Route-Based Multi-Level Variable Neighborhood Search (Route-MLVNS) algorithm. Route-MLVNS is a variable neighborhood search algorithm designed to solve the orienteering problem with task times in hallway-based environments. Like other variable neighborhood search algorithms that have been used to solve the orienteering problem, Route-MLVNS creates a plan using a simple initialization algorithm, then repeatedly searches neighborhoods of that plan for one with a higher reward, gradually exploring the solution space and using a perturbation method to avoid getting stuck at local maxima.

However, rather than representing and manipulating plans as sequences of tasks, Route-MLVNS instead works with "waypoint routes," sequences of locations that only represent where the agent travels and not the tasks it performs. The process of evaluating a waypoint route considers all locations the agent passes while following the route, rather than only a fixed set of tasks. Furthermore, the structure of the solution space of waypoint routes differs from the solution space of task plans, allowing simple local search operations to traverse the solution space in a more effective manner than an algorithm performing the same operations on task plans.

Waypoint routes have the additional property that a single waypoint route can represent multiple different plans with different time limits. As a result, Route-MLVNS can return not just a single plan, but a function that takes a time constraint and outputs a plan. We take advantage of this by allowing the algorithm to explore a set of plans over a continuous range of time constraints, rather than a single plan, and return a function that returns the best plan found for any time constraint in the range.

We compared the effectiveness of Route-MLVNS to an MLVNS algorithm from the literature that is capable of solving the orienteering problem for multiple discrete time constraints simultaneously. Route-MLVNS was able to find plans that yielded more rewards in the same amount of planning time in both procedurally-generated hallway-based environments and maps based on real office buildings, for both single time constraints and continuous ranges of time constraints.

8.1.4 Learning Unknown Rewards in a Structured Environment

Our final contribution was two algorithms, Planning UCB1 and Planning Thompson Sampling, designed for scenarios in which an agent must solve repeated iterations of the orienteering problem with task times in a structured environment with stochastic rewards and no initial knowledge of the reward functions. To maximize the reward received over time, the agent had to balance exploration and exploitation, gathering sufficient data to learn a model of the reward functions while also taking advantage of that model to gather reward.

We did not specifically seek to exploit the structure of the agent's environment in this scenario. Instead, we considered the environment structure in other ways. First, we sought a solution to the trade-off between exploration and exploitation that was independent of the planning algorithm used to solve each iteration of the orienteering problem. Our goal was to enable the planning algorithm used by the agent to be selected based on the structure of the environment without any concern about how it would impact the agent's ability to learn a model of the reward functions over time. Second, we considered the relationship between the structure of the environment and the distribution of data the agent received when analyzing our results.

The two algorithms we contributed, Planning UCB1 and Planning Thompson Sampling, took the same core approach. With both algorithms, we treated the tasks the agent could perform in the environment as if they were levers in a multi-armed bandit problem. Our approach was to assign values to each action, taking into account both the expected reward of the action according to the agent's existing data and the potential value of gathering more data from each action, using algorithms that have proven effective in regular multi-armed bandit problems. However, rather than simply performing the action with the highest assigned value, the agent would treat the assigned value as a reward function, and find a plan that maximized the assigned reward received by solving the orienteering problem.

We tested Planning Thompson Sampling and Planning UCB1 in structured hallway-based environments in which the orienteering problem could reasonably be solved optimally, and found that they were effective at both gathering reward and learning a model of the reward functions of the environment.

A notable property we found in the structured environments is that the nodes in the environment were not equally likely to be part of the optimal plan, regardless of their reward.

In a star-shaped environment, nodes closer to the center were more likely to be part of the optimal plan. Meanwhile, in an environment where each iteration force the agent to choose between multiple hallways, nodes in hallways that formed bottlenecks were more likely to be part of the optimal plan.

We found that the frequency with which Planning Thompson Sampling and Planning UCB1 gathered reward at each location in the environment roughly corresponded to how often that location was part of the optimal plan in both environment structures studied. Meanwhile, other algorithms tested often had a different distribution of tasks performed.

8.2 Future Work

In this section, we discuss promising directions in we believe our work can be expanded on with further research.

8.2.1 Further Applications of Concepts and Algorithms

One of our goals when creating the algorithms of this thesis was for them we be modular in nature. We sought to create algorithms for which the key concept behind the algorithm, and the way in which it exploited the environment's structure, was not intrinsically tied to the details of our particular implementation of it.

Our motivation in doing so was two-fold: First, because environments could be structured in such a way that parts of our algorithms could apply, but not the whole. For example, an environment could exist in which the use of the core concept of the Task Graph algorithm — the construction of the task graph and planning use of it in planning — could be appropriate, but the resulting task graph could be sufficiently complex that choosing a plan of tasks requires a more sophisticated algorithm than the simple hill-climbing one used in our experiments.

Second, while our work was focused on specific planning problems, primarily the Orienteering Problem with Task Times and the Exploration Scheduling Problem, we believe that our work can also apply to a wider variety of problems. There are many variants of the orienteering problem in particular for which the exact algorithms we have presented do not apply, but could easily be extended, and possibly combined with existing research, to aid in solving them in structured environments. The RegionPlan algorithm, for example, could be applied to the team orienteering problem, assigning each agent a plan for a different region. And many approaches used to solve variants of the orienteering problem could easily be tested using the concept of route-based task-planning and applying the algorithms to waypoint routes rather than task plans.

Overall, we believe that the approaches described in this thesis can be applied to a wide variety of scenarios beyond the ones we tested ourselves that warrant further exploration.

8.2.2 Automated Analysis of Environment Structure

The focus of this thesis was identifying certain structural properties that many real-world environments have and presenting algorithms that can exploit those properties when appropriate. However, not all environments have a structure that is ideal for exploitation, and when environments do have an exploitable structure the decision must still be made how best to exploit it.

A potentially valuable path for future work would be to research how environments with structures with the potential to be exploited could be identified automatically. Ideally, algorithms could be found that not only determine if an environment has an exploitable structure, but can also identify the best planning algorithms that can be used to exploit it.

Furthermore, some of our algorithms require some analysis of the environment structure to be done to determine what inputs should be given to the algorithm. Specifically, the tasks in the Task Graph algorithm and the regions in the RegionPlan algorithm were chosen by a human based on the structure of the environment in our experiments. This is another process that could benefit from being automated in future research, creating variants of the Task Graph or RegionPlan algorithms that select the tasks or regions used automatically rather than requiring them to be given as input.

Overall, we believe that studying the situations in which the environment structure should be exploited, and reducing the amount of human input required to identify when a problem takes place in an an environment with an exploitable structure and choose an appropriate algorithm, would be an effective way to widen the range of uses our algorithms have and increase the ease with which they can be used.

8.2.3 Exploiting Environment Structure to Improve Learning

In Chapter **??**, we examined the relationship between the structure of the environment, the frequency with which locations in the environment appeared in the optimal plan for an iteration of the orienteering problem, and the opportunities available to the agent to gather data from each location in the environment. However, the actual algorithms we contributed were not designed to exploit the structure of the environment, only to be independent from the planning algorithm used so that it could be chosen according to the environment's structure.

In our own experiments in Chapter **??**, we concluded that the Planning Thompson Sampling and Planning UCB1 algorithms we contributed were effective at handling the constraints of the environment structure. We came to this conclusion based on the fact that those algorithms performed tasks at each location in the environment at roughly the same rate those locations appeared in the optimal solution.

However, as more situations and environment structures are explored in the future, including larger environments, it is possible we will find environments in which the structure of the environment and its impact on the data gathered by the agent is something that could be considered and exploited more directly. Ultimately, our results very clearly showed that the structure of the environment does have an impact on how often the agent will visit, and should visit, each location of the environment. We believe this relationship is worth investigating further and may be exploitable by learning algorithms in the same way we showed that the structure of the environment can be exploited by planning algorithms.

For example, one consequence of a structured environment that we discussed is that the agent will naturally pass by some locations, and have the opportunity to learn about them, more than others. It could be desirable to counteract this by having the agent value information about locations that it passes less often more highly.

8.2.4 Testing in Real-World Situations

Our research in this thesis was focused on the general problem of solving planning problems in structured environments, and exploiting the environment structure to improve planning. We demonstrated the value of exploiting the structure of environments in simulation.

However, one of the primary motivations of our work is the structured nature of many real-world environments, which could be exploited by robots that operate in them. Naturally, in the future, we would like the ideas in this thesis to be applied to real robots. Doing so may involve overcoming some challenges, however. For example, the time it takes a robot to travel from location to location may be uncertain, or even time-dependent, in real-world environments, due to factors such as vehicle or foot traffic. The time it takes to perform tasks may similarly be variable and uncertain. A robot may also have unanticipated changes to its schedule, such as a new user request being scheduled in the middle of the robot's spare time.

Ultimately, there will likely be many challenges, both anticipated and unanticipated, that occur in applying our algorithms to real robots. However, we believe the flexibility and modularity of our algorithms will allow them to be adapted to tackle any challenges that they face.

8.3 Final Thoughts

Existing research on reward-gathering planning problems such as the orienteering problem rarely considers the structure of the environment, let alone attempts to exploit the properties of that structure to improve planning. However, many environments in which planning problems take place do have notable structural properties. In this thesis, we showed that those properties can be exploited to improve planning in a variety of ways. We believe that the particular algorithms and approaches described in this thesis, as well as the general observations on the impact of an the structure of an agent's environment on its ability to plan and learn, will be valuable tools for further research in the future.

Appendix A

Visiting Frequencies of IOP-USR Tests

This appendix contains the visiting frequencies for each node for each environment and algorithm used in the experiments described in Section ??. Each image is a map of the environment with a number on each node representing how often a task was performed at that location by each algorithm.

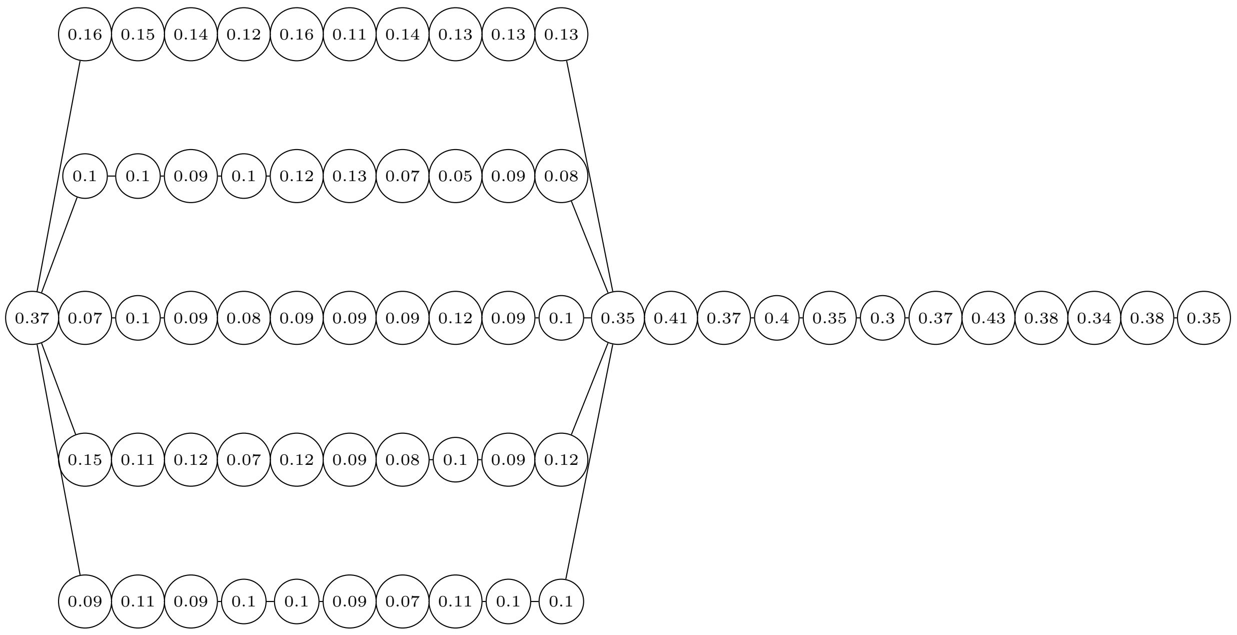


Figure A.1: The average frequency with which each location was part of the optimal plan in the hallway sequence environment.

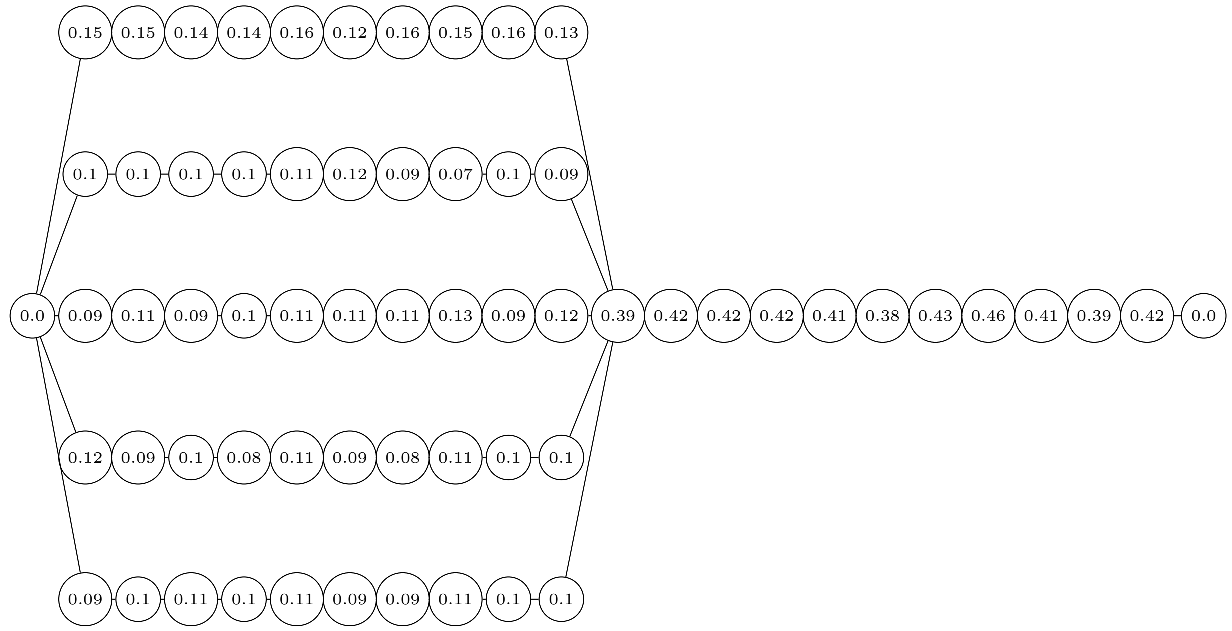


Figure A.2: The average frequency with which the Planning Thompson Sampling algorithm performed a task at each location in the hallway sequence environment.

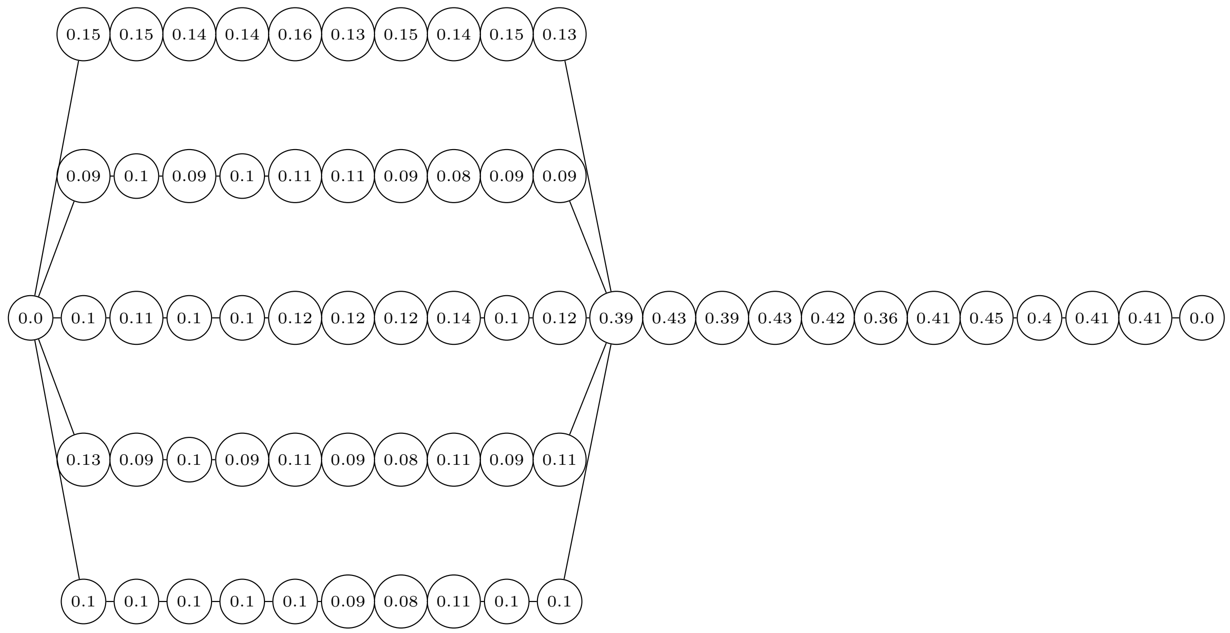


Figure A.3: The average frequency with which the Planning UCB1 algorithm performed a task at each location in the hallway sequence environment.

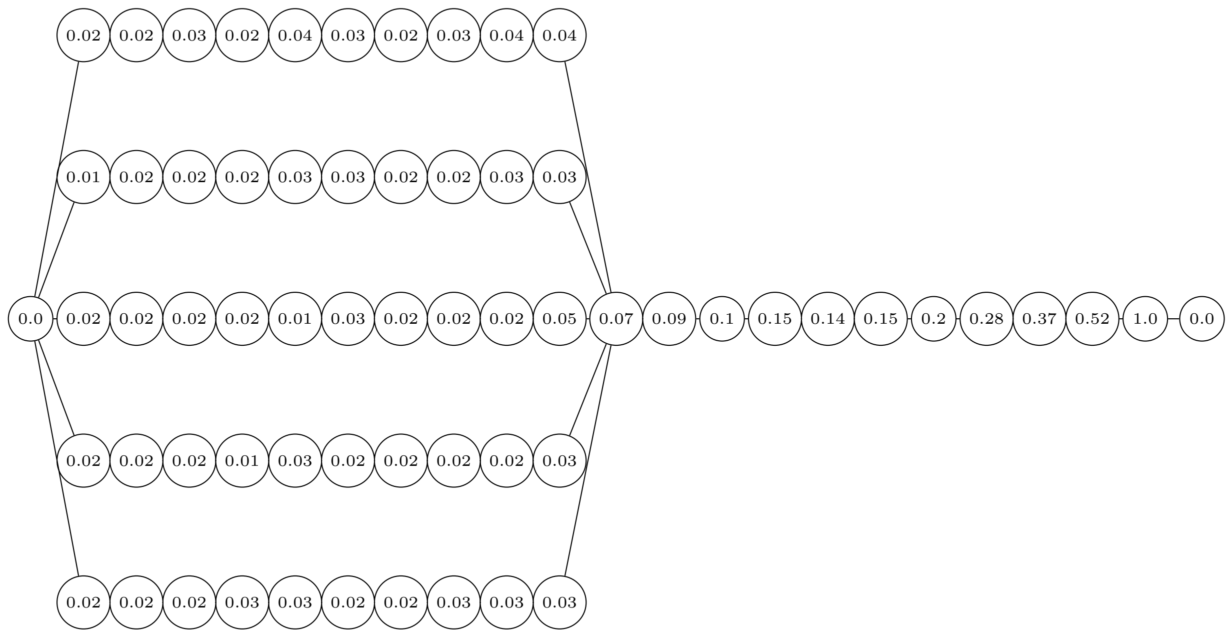


Figure A.4: The average frequency with which the Naive Thompson Sampling algorithm performed a task at each location in the hallway sequence environment.

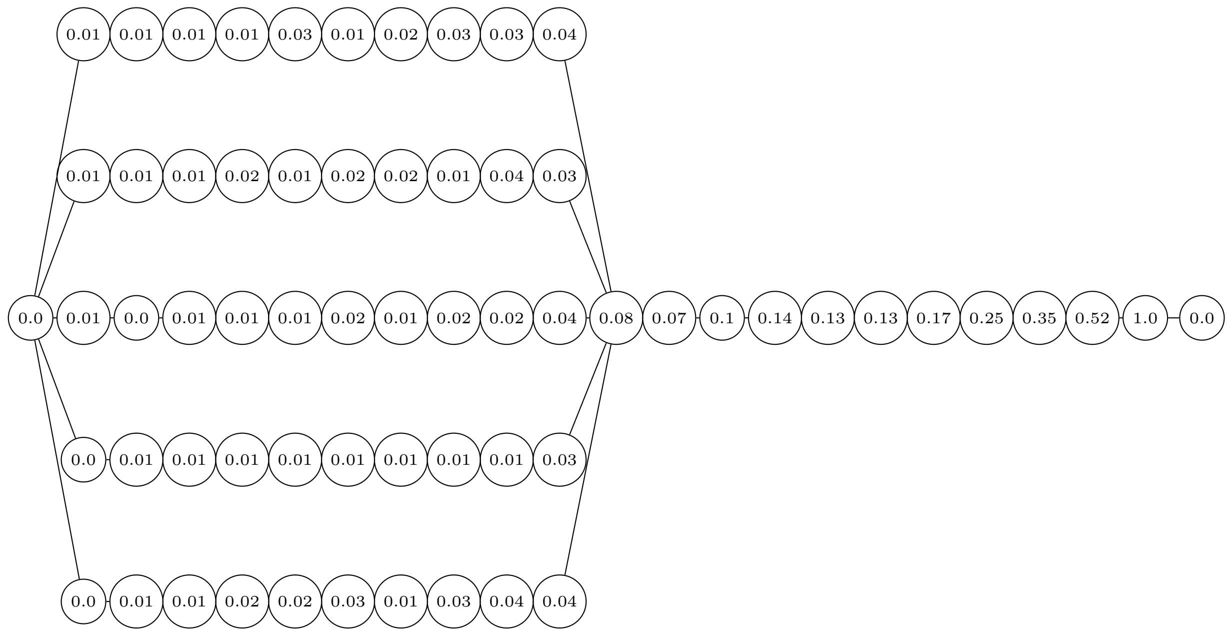


Figure A.5: The average frequency with which the Naive UCB1 algorithm performed a task at each location in the hallway sequence environment.

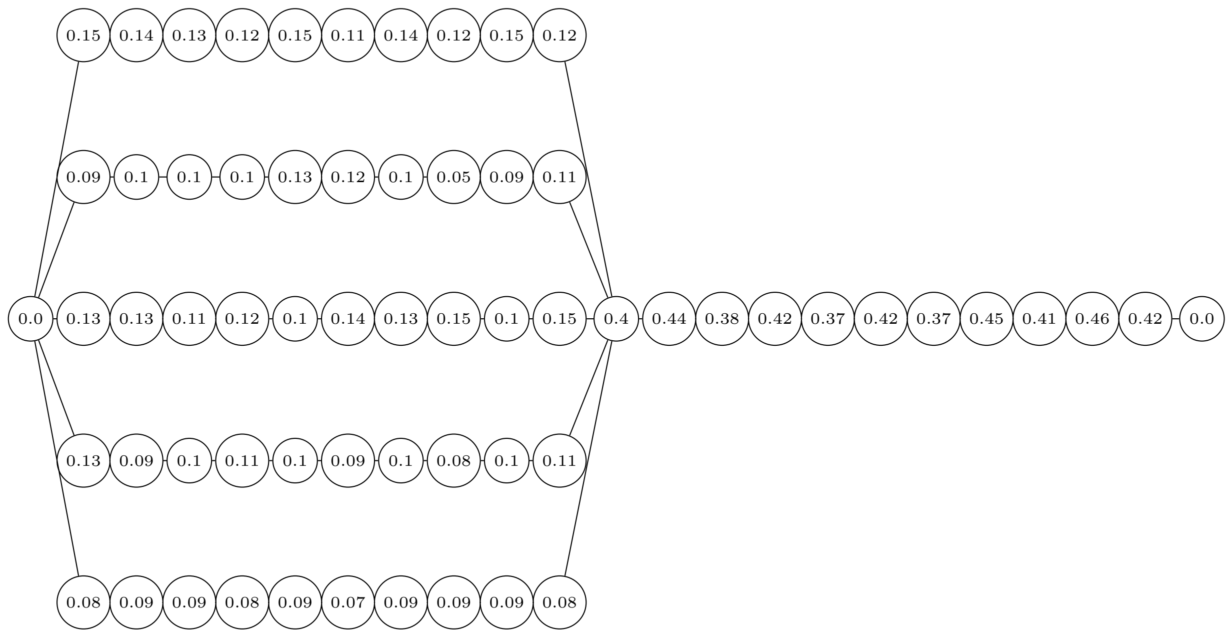


Figure A.6: The average frequency with which the Greedy algorithm performed a task at each location in the hallway sequence environment.

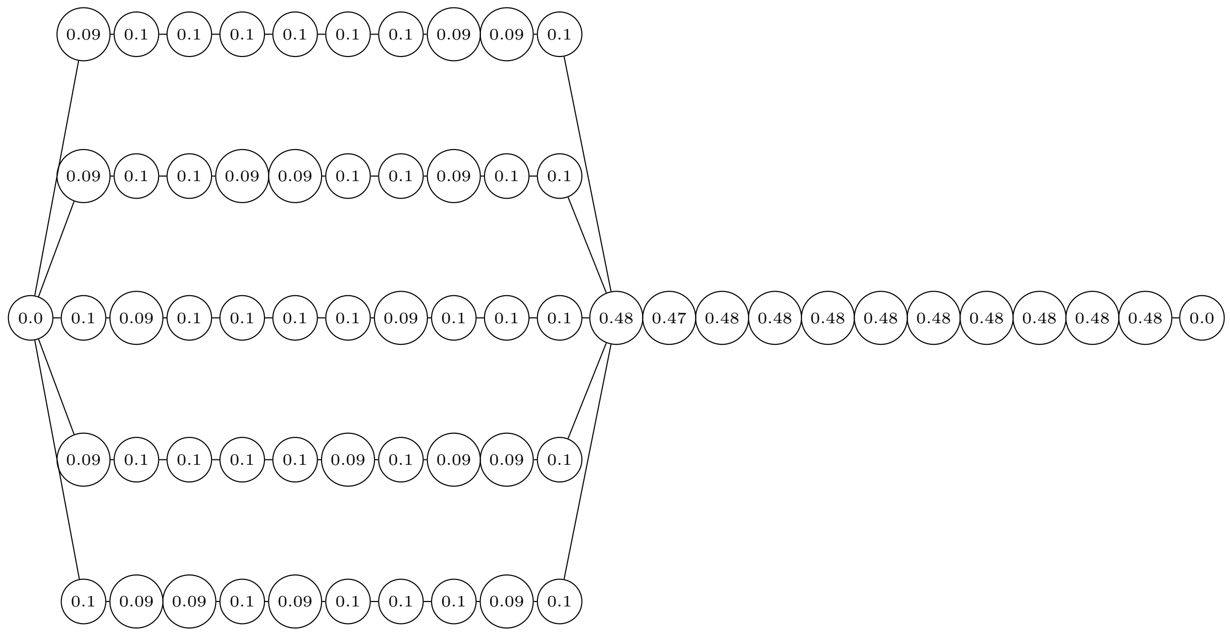


Figure A.7: The average frequency with which the Random algorithm performed a task at each location in the hallway sequence environment.

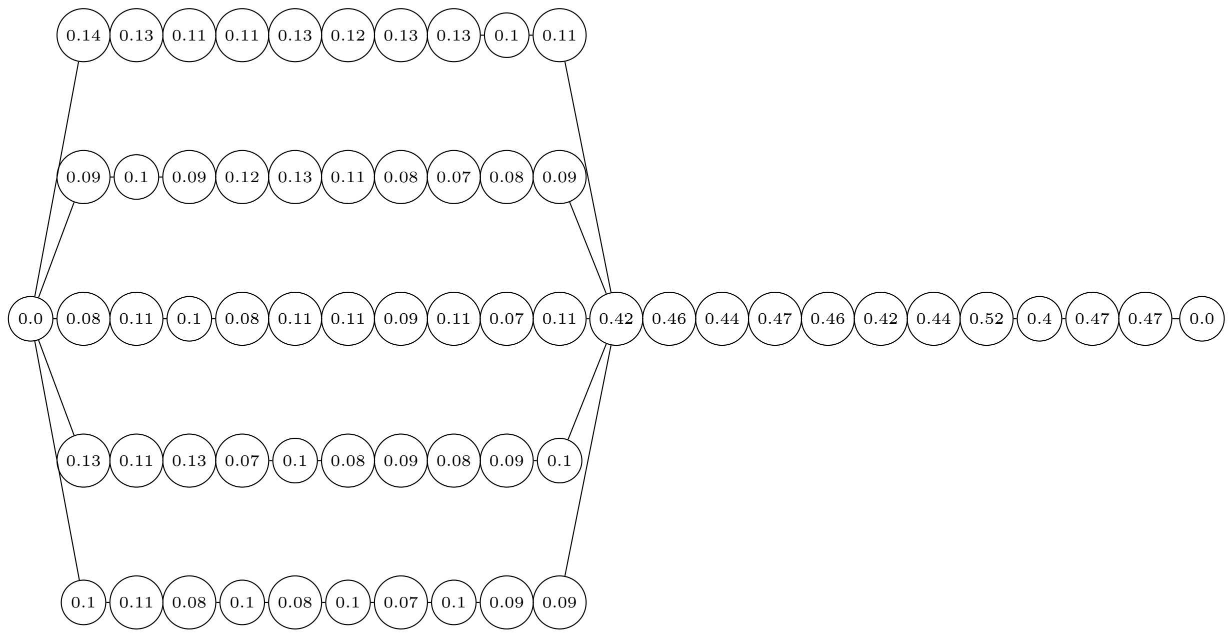


Figure A.8: The average frequency with which the Epsilon algorithm performed a task at each location in the hallway sequence environment.

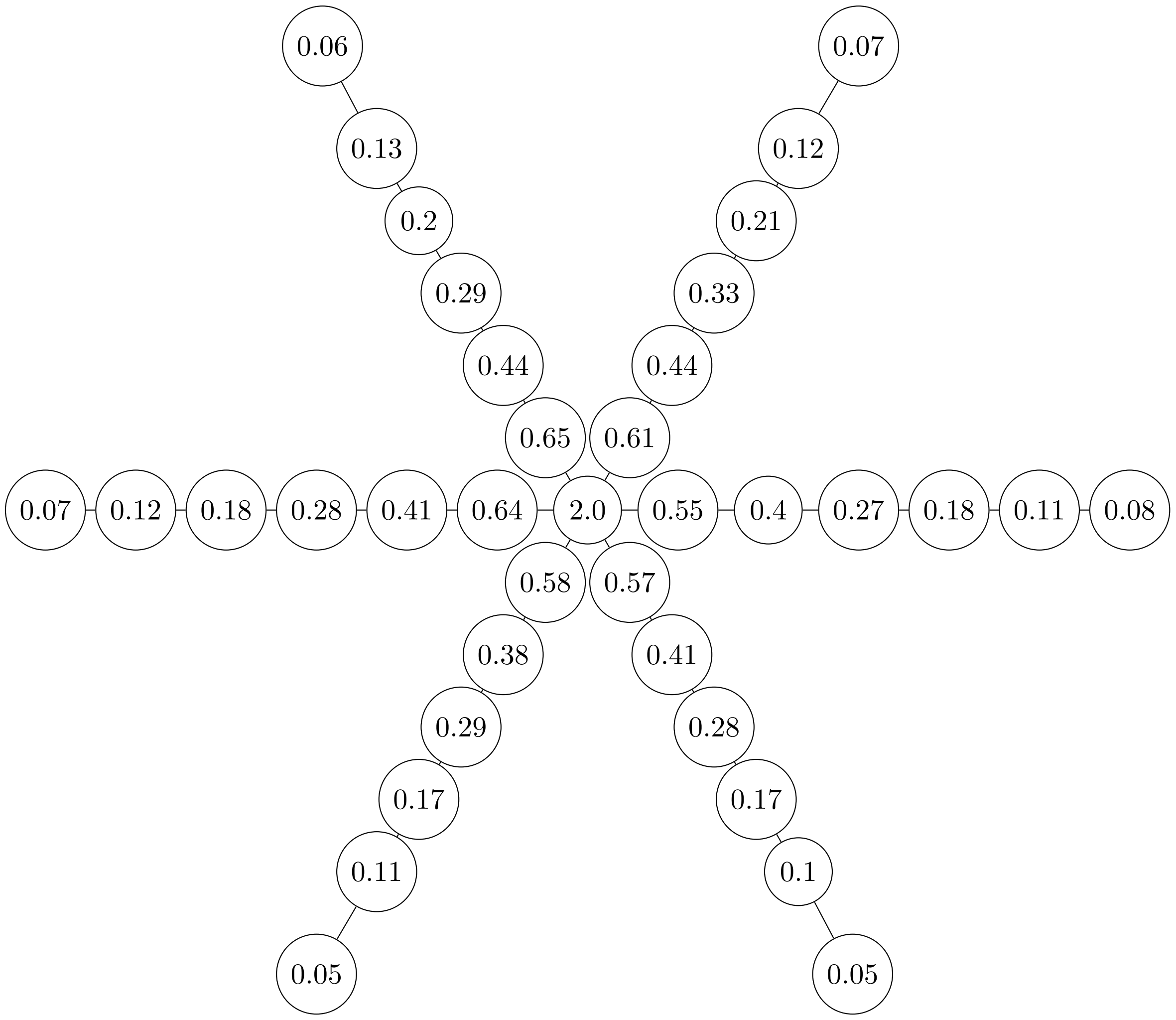


Figure A.9: The average frequency with which each location was part of the optimal plan in the star environment.

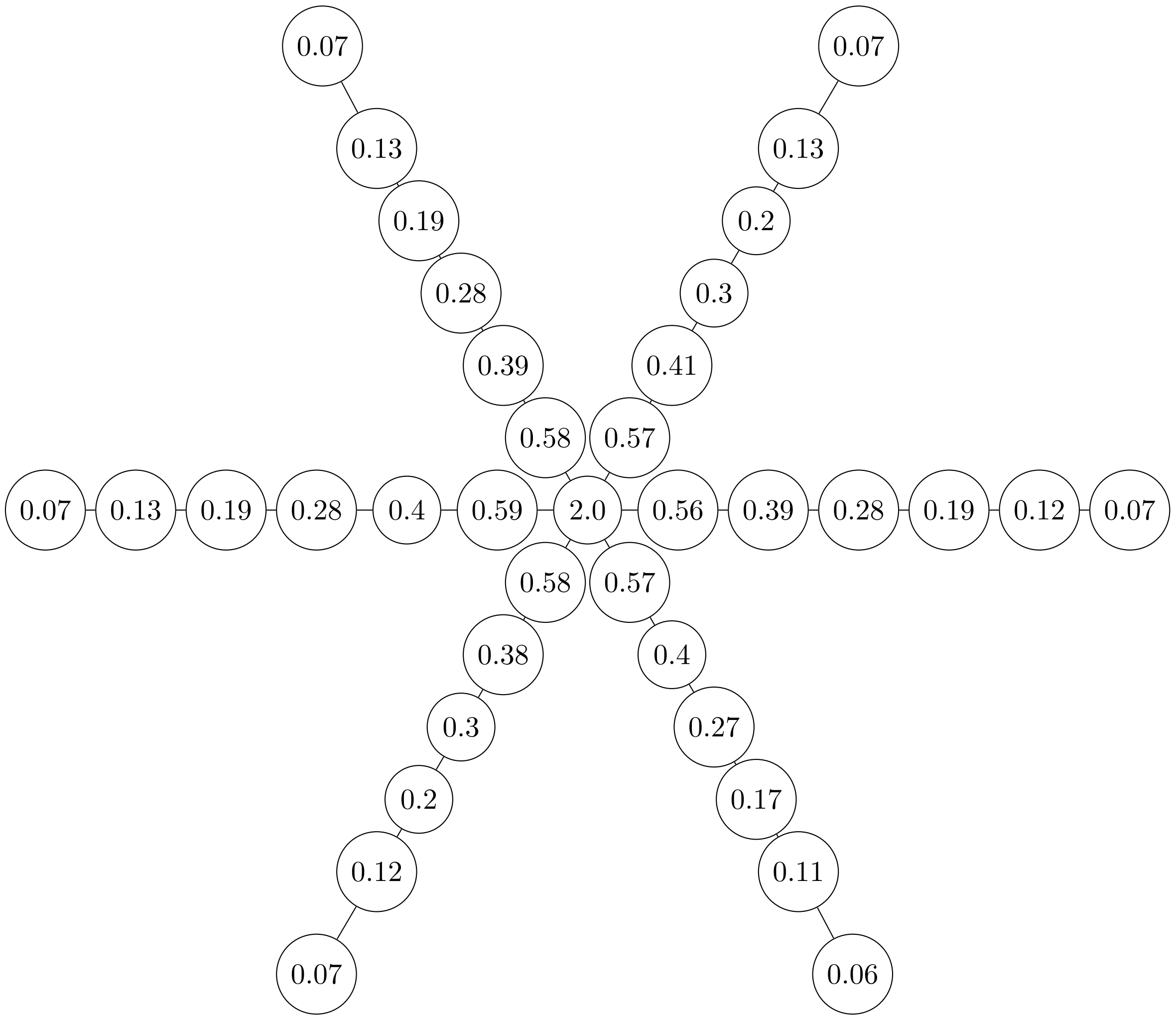


Figure A.10: The average frequency with which the Planning Thompson Sampling algorithm performed a task at each location in the star environment.

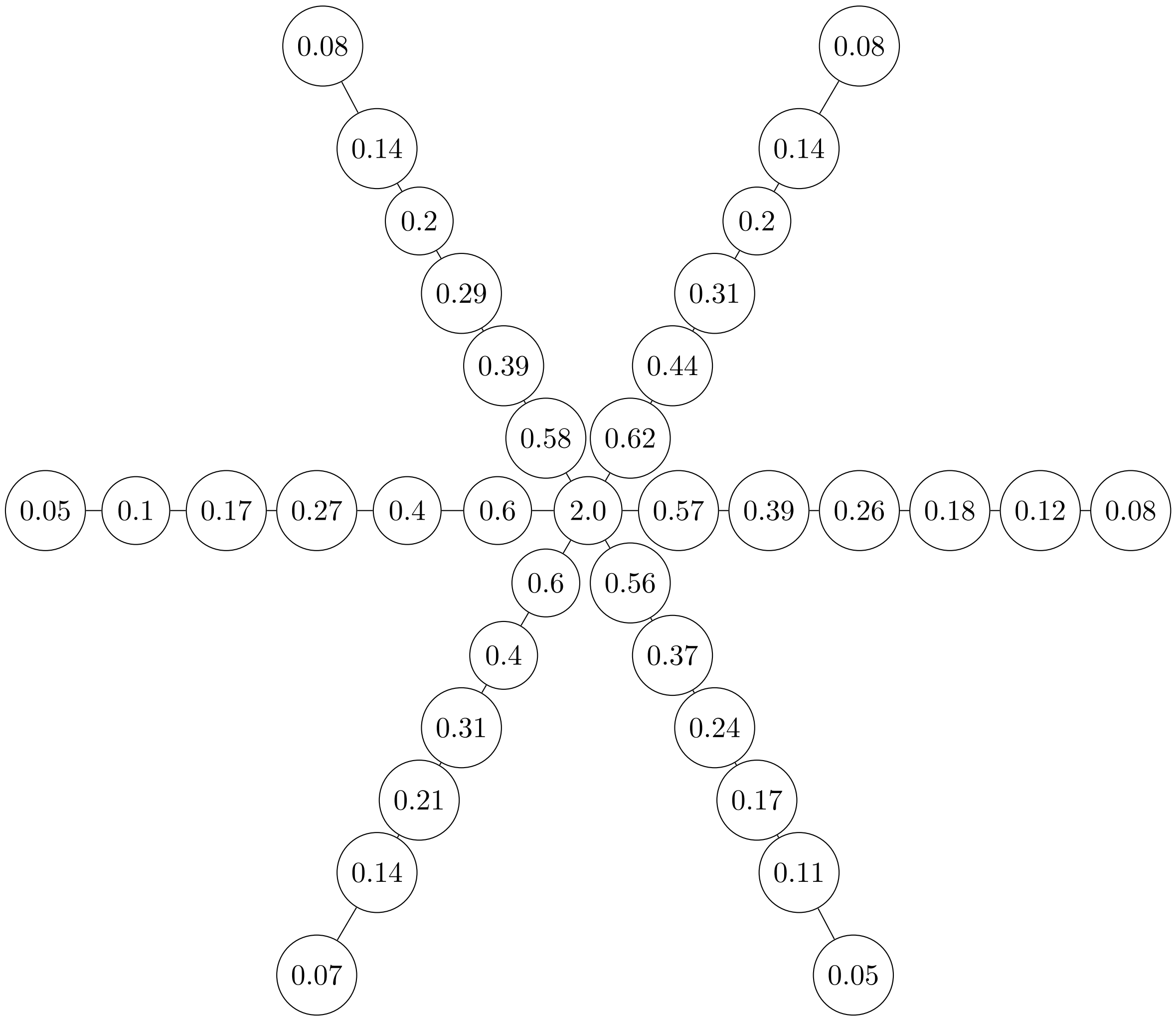


Figure A.11: The average frequency with which the Planning UCB1 algorithm performed a task at each location in the star environment.

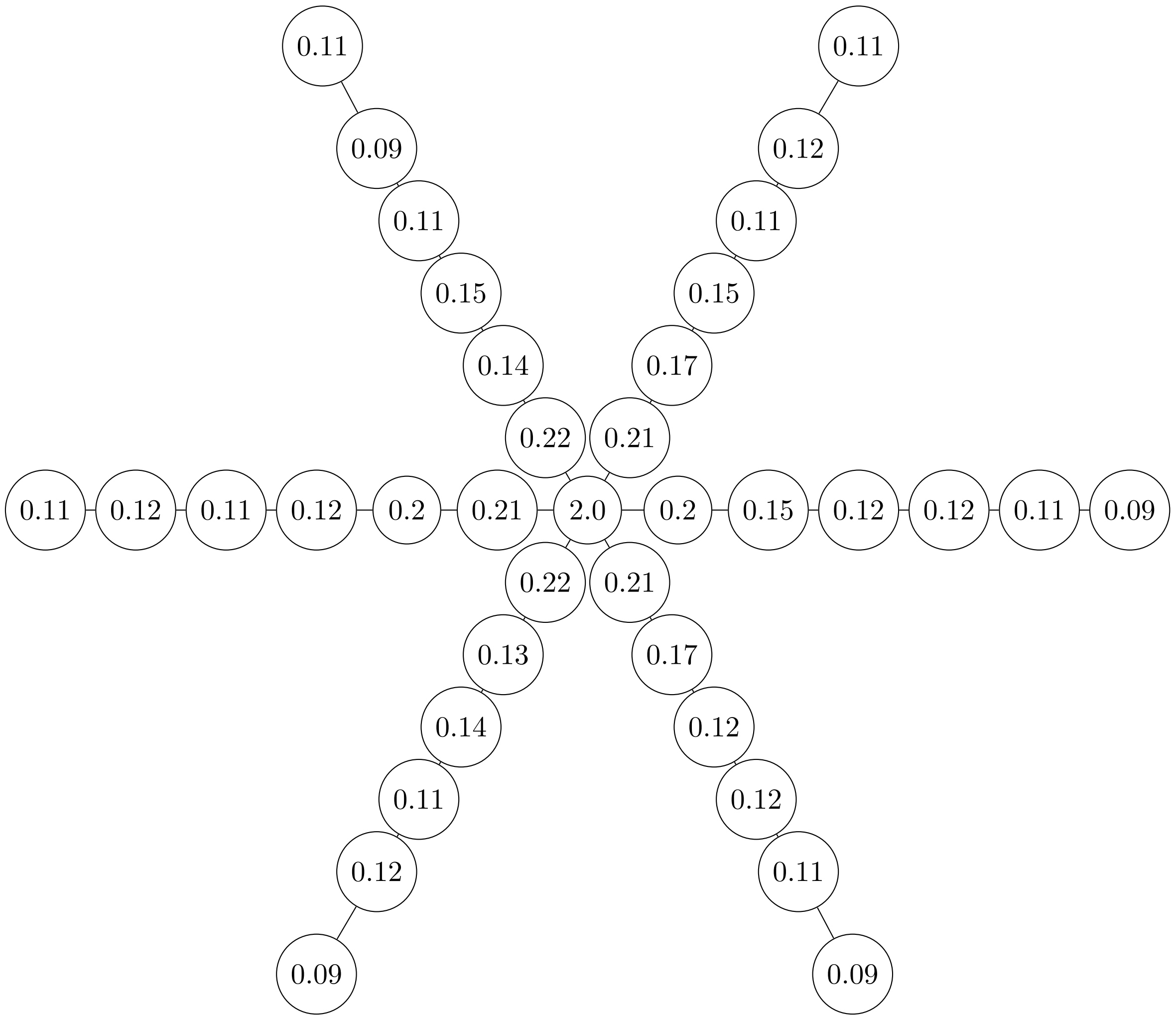


Figure A.12: The average frequency with which the Naive Thompson Sampling algorithm performed a task at each location in the star environment.

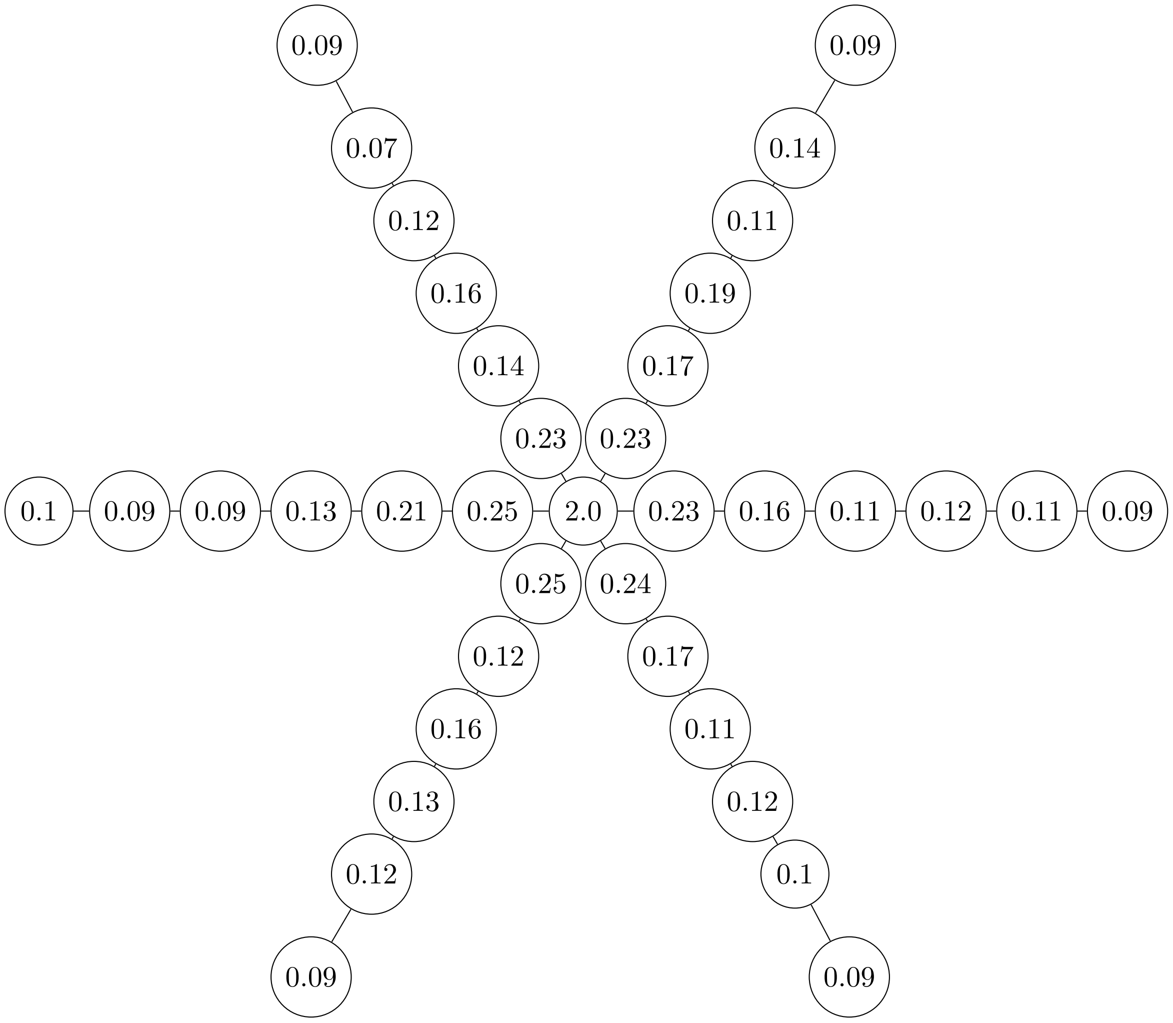


Figure A.13: The average frequency with which the Naive UCB1 algorithm performed a task at each location in the star environment.

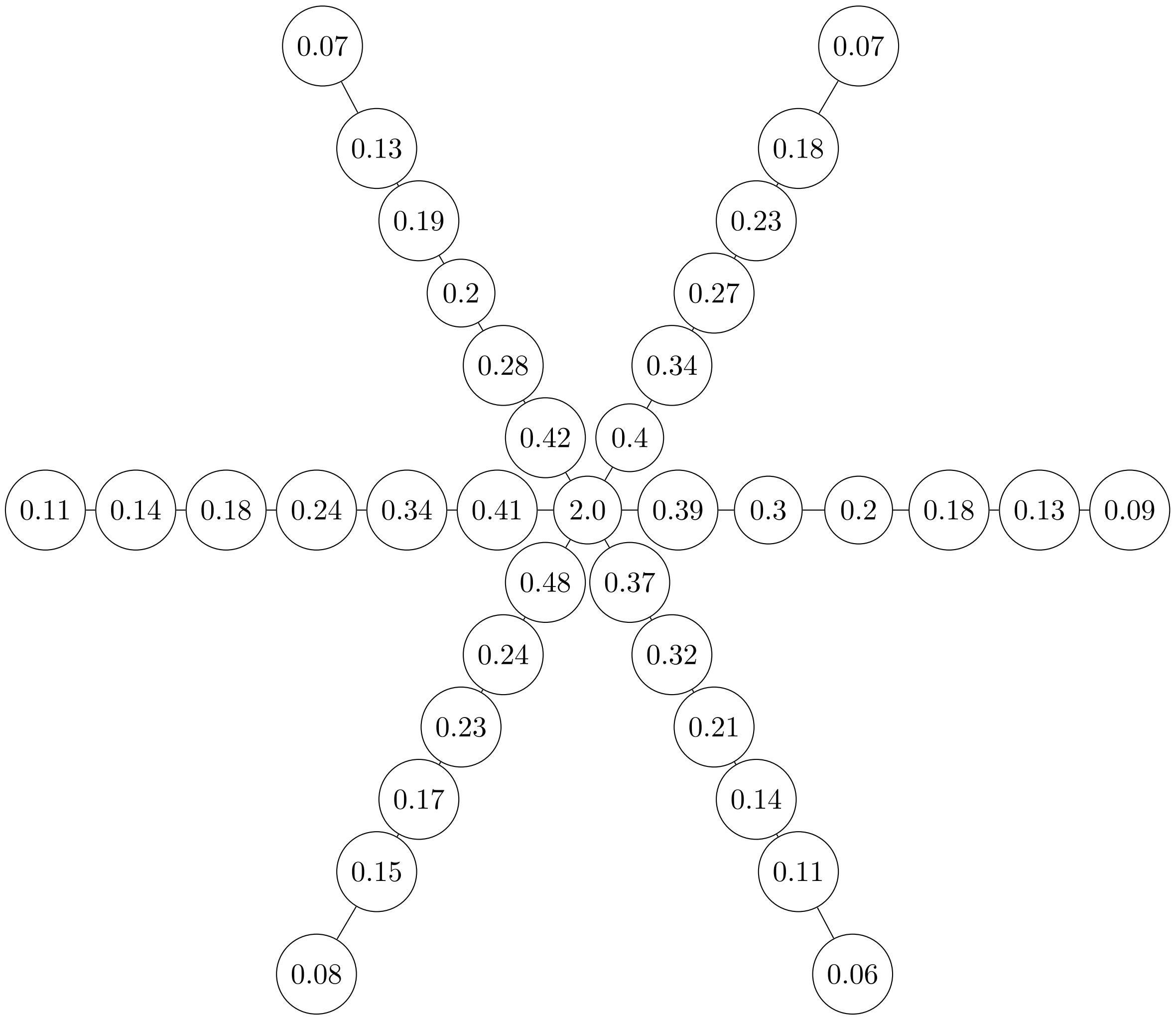


Figure A.14: The average frequency with which the Greedy algorithm performed a task at each location in the star environment.

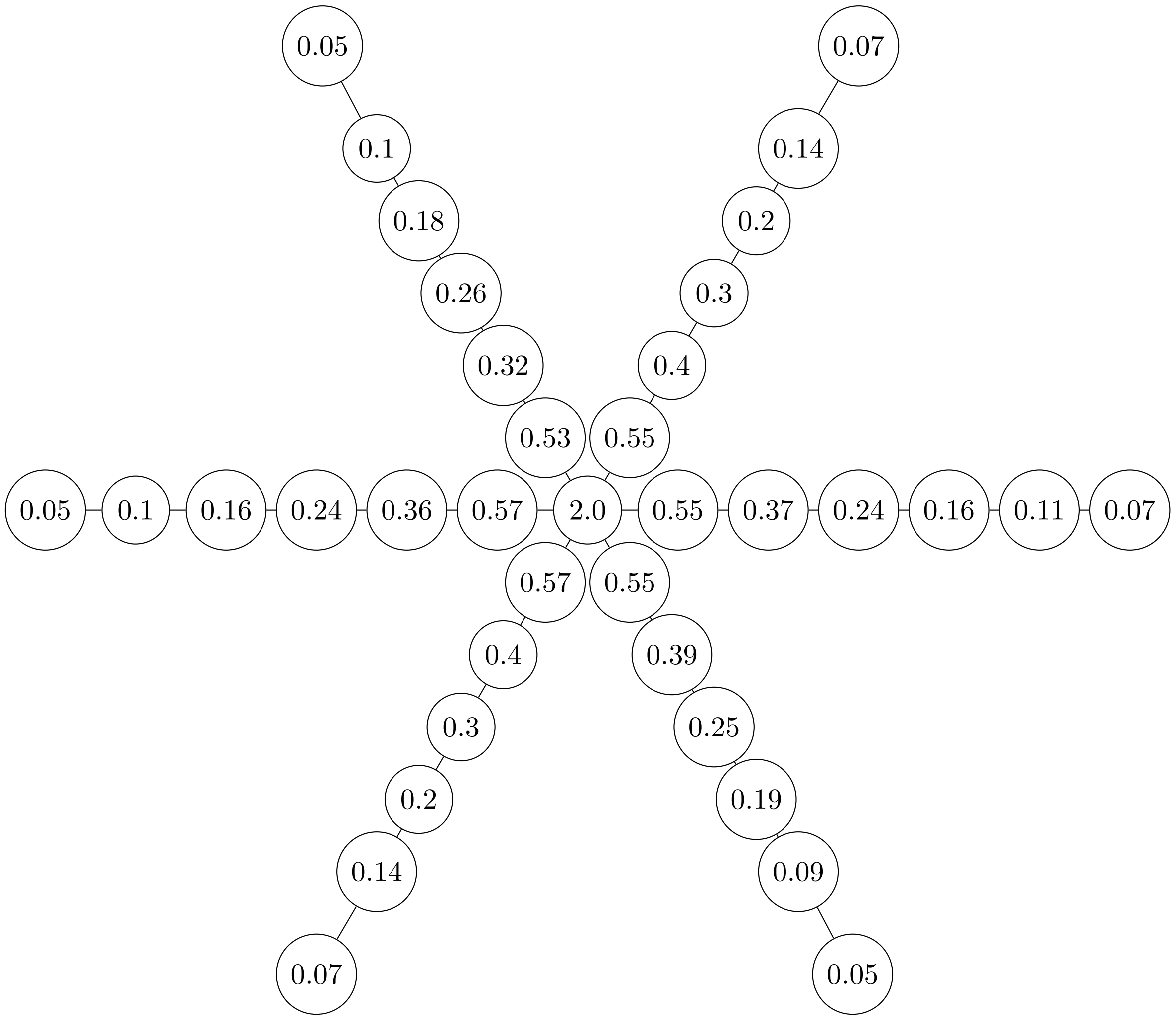


Figure A.15: The average frequency with which the Epsilon algorithm performed a task at each location in the star environment.

List of Figures