ROBOTICS NEEDS NON-CLASSICAL PLANNING

BY

Scott Kiesel

BS of Computer Science, Plymouth State University, 2006

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

September, 2016

This dissertation has been examined and approved in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science by:

Dissertation Director, Wheeler Ruml,
Associate Professor of Computer Science,
University of New Hampshire

Radim Bartŏs,
Associate Professor and Chair of Computer Science,
University of New Hampshire

Philip J. Hatcher,
Professor of Computer Science,
University of New Hampshire

Christoper Amato,
Assistant Professor of Computer Science,
University of New Hampshire

Maxim Likhachev,
Research Associate Professor Robotics Institute and
National Robotics Engineering Center,
Carnegie Mellon University

On August 5th, 2016

DEDICATION

To my family and friends who always provided support and demonstrated more patience than I probably deserved.

ACKNOWLEDGMENTS

Certainly this work would not have been possible without the interactions I shared with a long list of unique characters over the past years. Every time I had the chance to talk about what I was working on provided an opportunity to see my work through someone else's eyes. Thank you for always listening.

I could not have made it this far without the understanding of my family. They see how much I enjoy what I do and gave me the time and support to finish this part my life in my own way.

The journey has been long and uncertain from my point of view, but I'm convinced that my advisor Wheeler always saw the endgame. When research progress seemed to halt, he always had a unique perspective to get things moving once more. After the number of years I've known him, he is still able to surprise me with how much he knows about everything and anything.

After so much time working in one area, you develop a certain sense of humor that few can understand let alone tolerate. Even fewer share that same sense of humor. These friends were able to make me laugh and are most likely responsible for my continued sanity. Thank you eaburns, seabass, stk5, flyngpngn, ylfchild and goof.

This work has been funded primarily by NSF and DARPA. I gratefully acknowledge their financial support over the years. I would particularly like to thank NSF for the support it has given me and many other researchers to attend conferences, doctoral consortia and summer schools.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

ROBOTICS NEEDS NON-CLASSICAL PLANNING

by

Scott Kiesel

University of New Hampshire, September, 2016

Classical planning has developed a powerful set of abstractions and assumptions that enables the study of the underlying characteristics of real world problems. While these abstractions and assumptions are beneficial in academic research, they prove to be a barrier against the direct application of classical planning to real world problems and systems. Similarly, non-classical planning approaches have been developed, constructing the necessary bridges between classical planning's assumptions and the hard truths of operating in the real world. These techniques remove many of the assumptions that simply do not hold while operating in the real world. They remove assumptions such as: a fully known initial world state, fully known future world states and unbounded, uninterrupted planning time.

This dissertation makes two contributions. First we show how uncertainty in the world model can be addressed through a non-classical planning algorithm called hindsight optimization. We consider two realistic sources of uncertainty: temporal uncertainty and open worlds. The second contribution is applying abstractions and techniques from the heuristic search community to motion planning. We demonstrate the power of abstraction in a complicated task and motion planning problem with temporal constraints. We then show how combining high level discrete reasoning, characteristic of heuristic search, can aid lower level sampling-based motion planning resulting in faster solving times and better solution quality.

CHAPTER 0

Introduction

Classical planning has developed a powerful set of abstractions and assumptions that enables the study of the underlying characteristics of real world problems. While these abstractions and assumptions are beneficial in academic research, they prove to be a barrier against the direct application of classical planning to real world problems and systems. As such, non-classical planning needs to be developed, constructing the necessary bridges between classical planning's assumptions and the hard truths of operating in the real world. These techniques will remove many of the assumptions that simply do not hold while operating in the real world.

The thesis of this dissertation is the same as its title: robotics needs non-classical planning. The robotics community can benefit from many ideas about high level reasoning from the planning community. Similarly, the planning community can benefit from addressing challenges posed by the applications and domains considered by the robotics community.

To that effect this dissertation makes two contributions. First we show how uncertainty in the world model can be addressed through a non-classical planning algorithm called hindsight optimization (Chong, Givan, & Chang, 2000). We consider two realistic sources of uncertainty: temporal uncertainty and open worlds. The second contribution is applying abstractions and techniques from the heuristic search community to motion planning. We demonstrate the power of abstraction in a complicated task and motion planning problem with temporal constraints. We then show how combining high level discrete reasoning, characteristic of heuristic search, can aid lower level sampling-based motion planning resulting in faster solving times and better solution quality.

0.1 Handling Uncertainty

In this portion of the dissertation we investigate two interesting classes of problems. The first focuses more on the temporal uncertainty of the underlying world model. It is not possible to know exactly how long an action will take to execute in many situations, especially when the actions are defined at a high level such as "move from location a to location b" or "pick up object c".

The second problem revolves around the idea of Open World Planning. This is a type of planning where the entire world state is not known a priori, but instead is slowly discovered over time. As the agent moves around the world and interacts with it through its sensors and actuators it is able to develop a more accurate picture of the world state.

0.1.1 Temporal Uncertainty

To show how versatile Hindsight Optimization is, we first direct our attention to Temporal Uncertainty. We begin by considering a simple robot assistant that could be given a variety of pick and place tasks around the house. The difficulty in this problem is however, that there is uncertainty in the objects' locations, the duration of each action to be executed, the success of an action's execution and also exogenous events that occur at uncertain time points.

In our implementation of Hindsight Optimization, we generate possible worlds that could exist given our current knowledge of the world. For example, we have some concrete knowledge of the possible locations of objects and temporal bounds on action durations or event happenings.

Given these possible worlds, we then use a very simple domain specific solver to try to maximize reward in each world. From that, we then rank the next action the robot could execute based on the expectation of reward that would follow. The action is executed, new sensor data arrives, new world samples are generated, planning happens again and finally a new next action is selected.

Using this simple approach we are able to demonstrate a reasoning agent with very interesting behavior. The agent is able to serialize goals based on expectation about temporal information, choose different actions based on their expectation to fail and even meet at rendezvous points with exogenous agents with uncertain arrival times.

Summary

This chapter provides an in-depth description of the proposed framework and an array of simulated results in the above described household assistant robot domain. Each result set targets and stresses a single aspect of the framework. A final set of results will also be provided to show how the framework is able to handle a realistic problems exhibiting all of the realistic temporal uncertainties in the domain at once.

This work was published as a 2014 ICAPS PlanRob paper (Kiesel & Ruml, 2014).

0.1.2 Open Worlds

In this chapter we will examine a few domains that exhibit open worlds, but the most significant is the Search and Rescue domain. In this domain, a robot is tasked with finding injured victims inside of a building. To make this more difficult, the robot does not know the layout of the building and it also does not know the locations of any victims.

The robot must balance exploration of the building and discovery of victims against an approaching temporal deadline to return to its home base. In this work we want to show that while very sophisticated techniques are the norm, simpler techniques also can perform quite well.

We use a form of Hindsight Optimization to plan for our robot to explore the building looking for victims. We call our system Optimization in Hindsight with Open Worlds (OH-wOW). In our implementation of Hindsight Optimization, we generate possible worlds that could exist given our current knowledge of the world. For example, we have some concrete knowledge of the building layout given the history of our sensor data (i.e. a partially formed map) and given a very rough idea of what a building might look like we could generate random topological building layouts. Inside that topological building layout we randomly distribute victims, or if an expected distribution was known, we would bias locations according to that.

Given these possible worlds, we then use a very simple domain specific solver to try to maximize reward in each world. From that, we then rank the next action the robot could execute based on the expectation of reward that would follow. The action is executed, new sensor data arrives, new world samples are generated, planning happens again and finally a new next action is selected.

Summary

This chapter provides an in-depth description of the OH-wOW framework and the range of problems this technique can be applied to. We also provide results from simulated testing environments on two benchmarks, a classic probabilistic planning "omelet domain" and a simulated search and rescue domain. We finally provide results from experiments run on a physical robot platform in the search and rescue domain.

This work was published as an 2013 ICAPS PlanRob paper (Kiesel, Burns, Ruml, Benton, & Kreimendahl, 2013) and also as a University of New Hampshire technical report (Kiesel, Burns, Ruml, Benton, & Kreimendahl, 2012).

0.2 Abstraction for Motion Planning

The second portion of the dissertation is on heuristics for motion planning. Motion planning finds itself as a module in almost every robotics system in some form. As such, they must find solutions quickly and find solutions of good quality. In order to accomplish this, it is important that motion planners are able to spend their time focusing on the important pieces of the search space expected to contain good solutions.

By speeding up the planning time of these types of planners, we can speed up the entire system they are embedded in. Also, if we are able to find solutions of better quality, we can also improve the overall execution performed by these systems.

0.2.1 Task and Motion Planning

In this chapter we consider the benefits of abstraction in a task and motion planning problem. Specifically, we examine the problem of vehicle routing and motion planning with temporal constraints. In this problem we are tasked with assigning v vehicles to visit w waypoints while avoiding static obstacles and minimizing cost induced by gaussian cost objects. Each waypoint also can be associated with a set of temporal constraints that can induce a partial ordering among waypoints and define a temporal window in which that waypoint must be visited. Waypoints also can have a radius that defines a circular area in which a vehicle must pass to consider that waypoint as visited.

To solve this very complicated problem we employ an abstraction over the possible space of routes a vehicle could traverse between waypoint pairs. In this abstraction, we assume there exists a continuum of routes between the fastest and cheapest routes between waypoints. Furthermore, we assume this continuum can be linearly interpolated across, between the two end points. By lifting the space of routes to a simple computation, we are able to dramatically speed up the high level task allocation problem by assuming we know about all possible low level routes. We use this information to do a local search over task assignments to vehicles, then use a linear program to minimize cost among time points to visit each of the waypoints in each route, and then finally task a motion planner with navigating between each waypoint pairing taking the specified amount of time. Of course, because of the use of the abstraction, the route may not be feasible, so we incorporate a feedback loop to the higher layers of planning to inform them about the information discovered at the lower layers.

Figure 0-1 shows an example solution for the Vehicle Routing and Motion Planning problem. Here there are 4 vehicles and 43 tasks. The intensity of the red in the image represents the magnitude of cost for traversing that area of the map. The solid gray polygons represent the areas of the map that can not be traversed. Each of the 43 tasks has an associated "achievement" radius as mentioned earlier, and contained within that radius is an 'x' marking the point at which that waypoint was considered visited along the route.

Summary

This chapter provides an in-depth description of the Waypoint Allocation and Motion Planning problem as well as our system developed to address this task and motion planning domain. We also provide results from handcrafted instances designed to target various aspects of the vehicle routing and motion planning problem. For example, scaling the number of waypoints and vehicles, minimizing cost through motion planning and examining a traveling salesman problem are discussed.

This work was published in a 2012 ICAPS paper (Kiesel, Burns, Wilt, & Ruml, 2012).

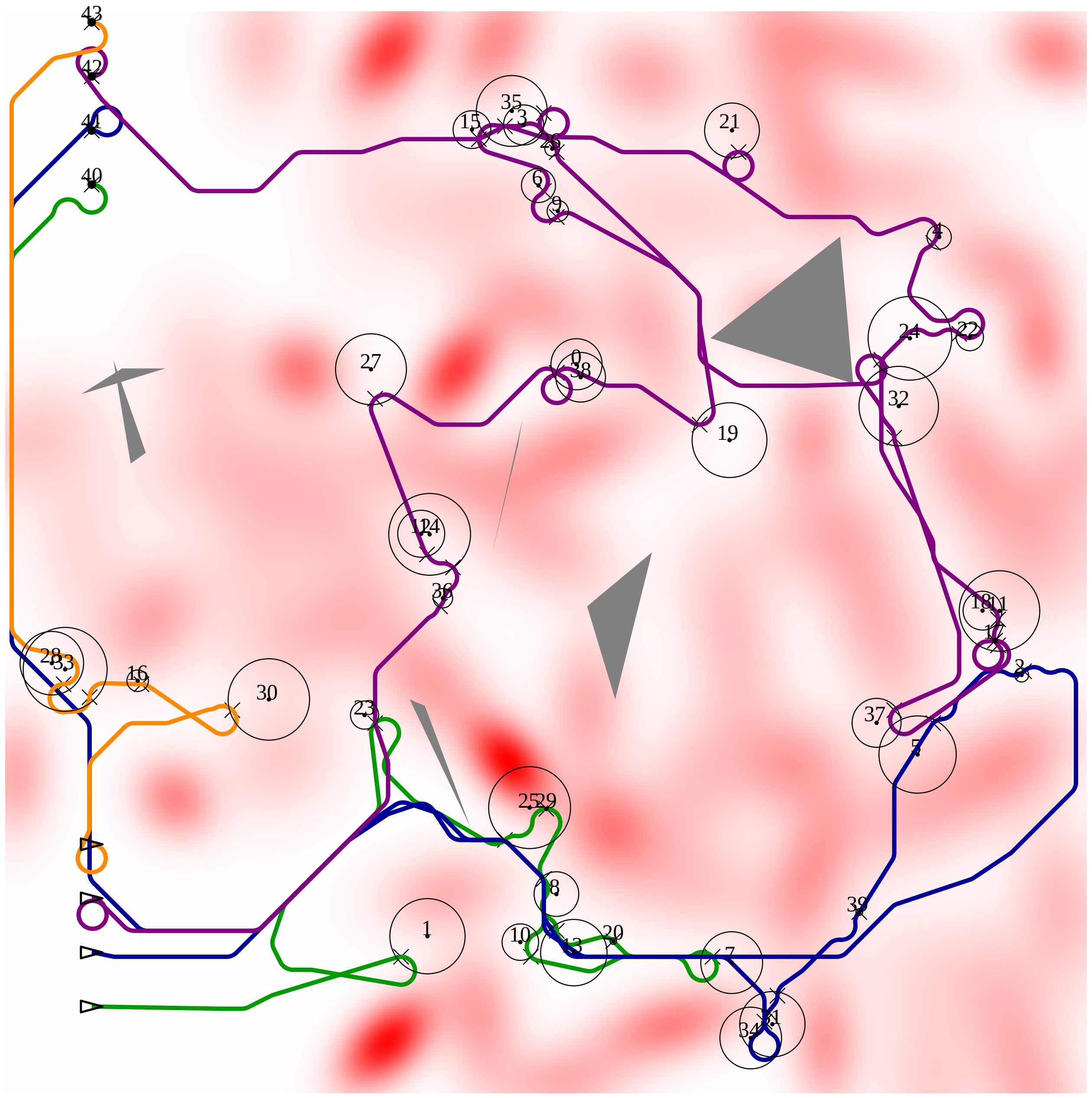
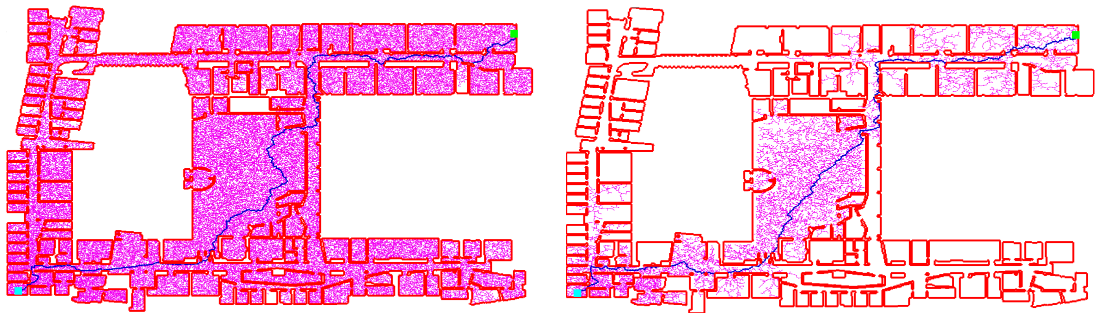


Figure 0-1: Example of a solution found by our planner in the Vehicle Routing with Motion Planning domain.



(a) (b)

Figure 0-2: Example of the power of intelligent biasing in sampling-based motion planning.

0.2.2 f-Biased Sampling

In this chapter we explore the use of Heuristic Search as an abstraction to bias sampling-based motion planner's exploration (Lavalle & Kuffner, 2000). By constructing a coarse abstraction over the workspace and using heuristic search in the abstraction we were able to identify large areas of the search space to focus sampling and other areas to (mostly) ignore. We are able to get much better performance compared to uniform random sampling and existing simple sampling biases such as goal biasing.

Summary

In this chapter we provide a description of the f-biasing technique along with results comparing it against similar sampling bias techniques. We provide these comparisons across a variety of domains.

This work was published as a 2012 SoCS extended abstract (Kiesel, Burns, & Ruml, 2012b) and also as a University of New Hampshire technical report (Kiesel, Burns, & Ruml, 2012a).

0.2.3 Hybrid Motion Planning

Following from the previously mentioned work, we can see that we are able to further bias sampling in a more intelligent manner. By maintaining a similar abstraction over the workspace, or even possibly the configuration space, we can run discrete heuristic algorithms while using sampling-based planners to form "edges" in the abstract space. By focusing on the frontier of the abstract space, we are able to build out the low level search tree much more quickly leading it through twists and turns much better than sampling along entire paths in the abstract space.

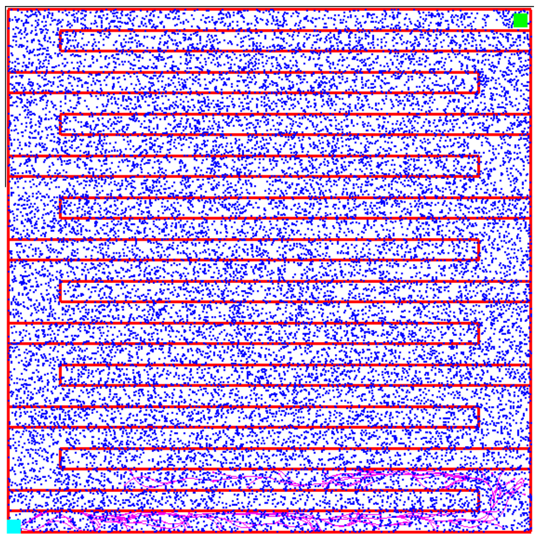
In Figure 0-3 we show a particularly devious example map. In this map the search tree will effectively have to climb a ladder of obstacles starting from the start state (blue square in the bottom left) to reach the goal (green square in the top right). In panels (a) and (b) we show the samples that have been generated to grow the tree from the start state to roughly the same progress up the first ladder step using f-biasing from the previous section and the newer idea where we maintain a notion of the current search tree in the abstract space. In panel (a), almost all of the abstract states look equally good so samples are generated all over the map. In panel (b), all states still look equally good in terms of their f values, but we are sure to keep the samples we generate near the currently built tree. As a result, many fewer samples are required to reach the same level of progress. To further illustrate the benefits of this idea, in panels (c) and (d), we show each algorithm after 20,000 samples have been generated. In panel (c), the search tree is nowhere close to the goal while in panel (d), the search tree has already arrived at the goal.

Seeing the parallels between heuristic search as a high level guidance and low level sampling-based algorithms we extend these ideas further by applying many of the algorithmic concepts used in anytime heuristic search to find solutions very quickly in the low level space and then improve them over time. We utilize some of the newer probabilistic roadmap (PRM) discretizations and tree sparsification techniques in our algorithm.

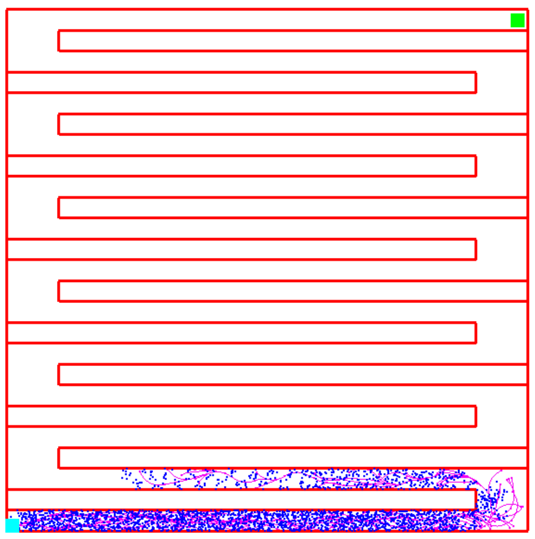
Summary

In this chapter we provide an in-depth description of our algorithms. We also provide comparisons across a variety of domains with other similar state of the art algorithms.

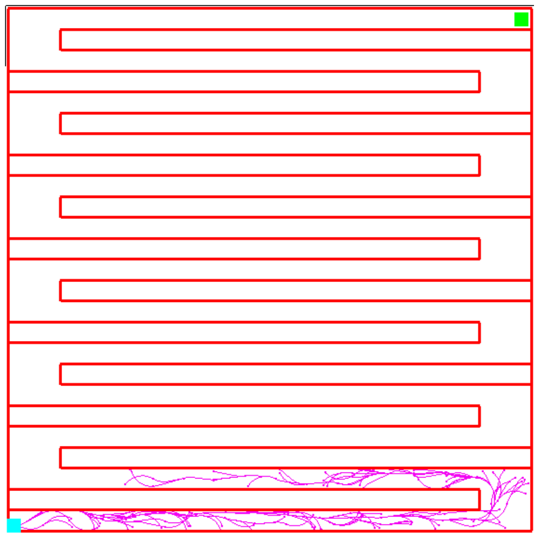
This work was accepted as a 2016 ICAPS PlanRob paper.



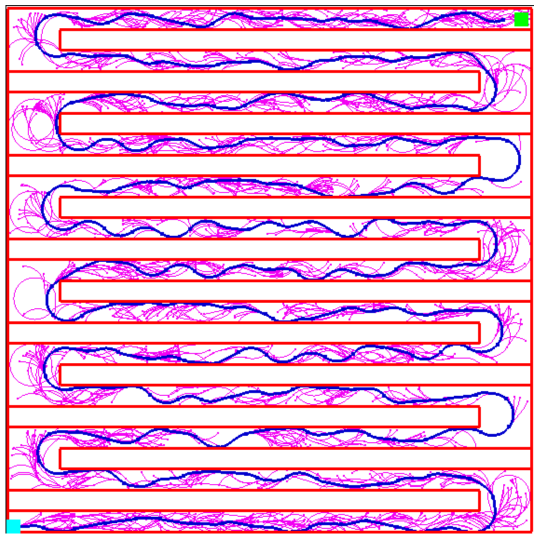
(a)



(b)



(c)



(d)

Figure 0-3: Example of keeping sample points near the currently expanded tree.

Part I

Handling Uncertainty

In this part of the dissertation, we consider uncertainty in the world model. Specifically, we examine a way of dealing with temporal uncertainty and uncertainty in the location of objects and events in Chapter 1. In classical planning, all temporal values are known in advance as well as the location of all objects. Then in Chapter 2, we explore a way of handling open worlds that robotics will face in a specific search and rescue scenario. In classical planning, the agent would know at the start of planning all there is to know about the world. It relies on a closed world assumption, that anything it does not know about is not important to reason about (or even doesn't exist).

Both of these types of uncertainty are not handled by classical planning techniques which deal directly in fully known deterministic domains. The uncertainty of operating in the real world is inescapable and requires non-classical algorithms to cope with it.

CHAPTER 1

Temporal Uncertainty

1.1 Introduction

In many real-world domains, such as robotics, a planning agent does not have complete knowledge of the world state or precise control over action outcomes of actors and processes. Incomplete world knowledge, stochastic actions and exogenous events are challenges a planning agent must tackle in many useful robotic applications.

For example, many planners make the assumption that an action, such as *pickup*, will have a deterministic outcome and duration. This is a fine assumption that makes planning much easier to reason about. However when the resulting plan is executed, actions can fail or take longer than anticipated. Consider the domain of a robot office assistant. When issuing the simple task of picking up a set of keys from your desk, the planner will quite quickly emit the plan: *pickup(keys)*. When this plan is executed, the pickup action might fail. If the action fails, then the execution certainly will not result in a goal state. The ability to reason about action outcome uncertainty becomes very important when a plan is intended to be executed.

Perhaps the goal state is slightly more interesting and the agent should pickup your keys and give them to you when you leave to go home. Another assumption that many planners make about actions is that their duration is a known constant. However, depending on the starting pose of the robot office assistant or the position of the keys, this simple *pickup* action can have a varying range of execution durations. If it could take anywhere between 1 minute and 10 minutes for your keys to be picked up, you might appreciate a planner that can take this range into consideration. The planner could start executing earlier, instead of causing you to be 10 minutes late going home. If a planner only assumes the best case for action durations, it is easy to see that the agent could be late to a rendezvous. On the other hand, if the planner only assumes the worst case for action durations, the agent may spend time waiting at a rendezvous point unnecessarily. It could instead be performing more productive tasks with this time. Similar behavior can occur when reasoning solely on the mean of the duration. The ability to handle action duration uncertainty becomes very important if a productive and punctual agent is desired.

An even more realistic situation for a robot office assistant to face is the task of retrieving your keys from a set of possible locations. It is not uncommon to forget exactly where you left your keys. Many planners however, require that the exact location of the keys be known when planning begins. If you need to manually search out your keys to simply set the initial state for your planner, you might as well skip using your planner because you have already found your keys. It is important for a planner to be able to handle location uncertainty if the exact state of the world is not known.

As hinted at in the previous example scenarios, the agent may not be the only entity in its world. Other agents may exist and these other agents are not necessarily under the control of the same planner as your robot office assistant. The world can be affected and changed outside of the control of the planner. Specifically, transitions in the world state may occur in ways entirely unrelated to agent action execution. Maybe you found your keys while trying to fully annotate an initial state for your planner. Instead, before you leave the office you would like your robot office assistant to give you a coffee for the trip home. The coffee is not essential for you to get home, but it makes the trip significantly more pleasant. As such, you are willing to wait for 10 minutes before you depart without your coffee. As a human, your timing is not always exact so you might leave sometime between 5pm and 5:30pm. If you would like to get your coffee frequently before you leave the office, it is important for a planner to be able to handle interactions with other agents and events exogenous to the planner.

Adding a single one of these three aspects of uncertainty, either incomplete world model or uncertain action durations or exogenous events, to a domain renders many planners inapplicable. Adding all three types of uncertainty further reduces the number of applicable planners. Those algorithms previously proposed to handle these uncertainties are complicated and can rely on computationally expensive data-structures such as a Simple Temporal Network With Uncertainty (STNU) (Morris, Muscettola, & Vidal, 2001). Many of them also require reasoning about all of the unknown factors at once requiring complex world representations.

In this chapter we introduce the Temporal-Uncertainty Hindsight Optimization Planner (Tu-Hop), a simple and straightforward approach that extends previous work on hindsight optimization. Instead of trying to manage the various uncertainties directly, we employ a basic sampling strategy and a deterministic specific planner. Tu-Hop begins with a set of beliefs about the initial world state. This belief state contains certain and uncertain information about locations, arrivals and departures of objects and agents and expected action outcomes and durations. Given the current belief, a set of deterministic world samples consistent with the belief state are generated and then solved by the domain-specific planner. Using the resulting solutions, the next action is chosen based on solutions maximizing overall expected reward. This action is executed, the belief about the world is updated based on the action's result and the process starts again.

We show that Tu-Hop is simple to understand and implement. We also show that Tu-Hop is very capable of solving problems containing uncertain action outcomes and durations, uncertain object and agent locations, as well as exogenous events.

1.2 Previous Work

There is wealth of literature on deterministic domain dependent and independent planners. We are able to leverage this previous work, as others have, to incorporate well researched deterministic planning ideas and concepts into a larger framework.

Yoon, Fern, and Givan (2007) incorporate a classical domain independent planner called FF (Hoffmann & Nebel, 2011) into their planning framework called FF-Replan to solve problems with uncertain action effects. FF-Replan uses FF to find a plan to carefully constructed deterministic version of the problem. It then executes actions according to the plan until the executed action has an unexpected effect or the goal is achieved. If an unexpected effect is observed before achieving the goal, FF is called once again to construct a new plan from the current state. There are also other planners, such as SDR (Shani & Brafman, 2011), that have been developed to handle and recover from deterministic planning with partial information.

1.2.1 Temporal Planning and Execution

EUROPA (Barreiro, Boyce, Do, Frank, Iatauro, Kichkaylo, Morris, Ong, Remolina, Smith, et al., 2012) is a class library and tool set for building planners within a temporal planning paradigm. It is a complicated architecture that handles time and resources and constructs plans offline. It is able to handle many temporal events using its modeling language NDDL. It relies on many handcoded internal modules and does not directly handle the uncertainties of object locations or action outcomes.

IxTeT (Ghallab & Laruelle, 1994; Laborie & Ghallab, 1995) is an complex offline planning and scheduling system that can handle time and resources by constructing partial order plans and resolving *threats* to the achievement of goals during planning. It strives to find a balance between the planning (what to do) and the scheduling (in what order to do it) addressing many domains in the intermediate spectrum between planning and scheduling. It however relies on absolute temporal bounds and does not take into account temporal uncertainty.

Procedural Reasoning System (PRS) (Ingrand, Chatila, Alami, & Robert, 1996) is a system for supervision and control of autonomous mobile robots. This system is explicitly able to monitor plan execution and provide feedback on action execution to an underlying planning system. However, PRS still relies on a high level planner to provide the high level actions for it to execute. Integration of Tu-Hop with PRS is a promising avenue for future research.

IxTeT-EXEC (Lemai & Ingrand, 2003) is complex system that allows for execution control, plan repair and replanning. IxTeT-EXEC is an extension of the IxTeT planner integrated with PRS (and several other layers). IxTeT-EXEC is able to handle temporal constraints (inherited from IxTeT) as well as action failures and unpredicted action outcomes as reported by PRS. However, this is a very complicated system that is non-trivial to implement and does not address the aspects of temporal uncertainty of interest in this chapter.

Simple Temporal Network with Uncertainty (STNU) (Morris et al., 2001) are an extension of Simple Temporal Networks (STN) (Dechter, Meiri, & Pearl, 1991). In many cases of interest in planning, an STNU would be used to determine dynamic controllability. If an STNU is dynamically controllable, then we can be assured that from the current state, the actions we plan to execute will not cause us to violate any future temporal constraints regardless of their outcome durations. An incremental approach, such as FastIDC (Nilsson, Kvarnström, & Doherty, 2014), can be used to compute dynamic controllability but requires $O(n^3)$ computation time, where n is number of nodes in the network. This computation must also occur each time an action or event is added to the network, for example each time a planner considers the application of an action during planning. In the case that an addition causes a violation of controllability, the entire network is copied before each addition so that backtracking can be done. This can be prohibitively expensive for an online planner that must plan and process new information as it arrives during execution.

1.2.2 Hindsight Optimization

Hindsight Optimization was originally developed for scheduling and networking problems (Chong et al., 2000; Mercier & van Hentenryck, 2007; Wu, Chong, & Givan, 2002) and has been used in a probabilistic planning setting (Yoon, Fern, Givan, & Kambhampati, 2008; Yoon, Ruml, Benton, & Do, 2010). In these previous applications, sampling was used to resolve uncertainty in the outcome of actions. Burns, Benton, Ruml, Yoon, and Do (2012) used hindsight optimization to solve a problem where exogenous goals are arriving, which requires the agent to plan ahead and anticipate these arrival events.

Similar to most sampling techniques, the samples of generated possible worlds used in this chapter are intentionally not exhaustive. They are intended to provide useful relative judgments on the expected value of actions. In hindsight optimization, given a current world state, we are faced with the choice between all applicable actions. We provide very high level pseudocode to walk through the general algorithm in Figure 1-1. The first step in hindsight optimization is to generate possible world samples (Line 3) that could be true according to the current world belief state. Then in order to estimate the value of an action, we apply that action in each of the sampled possible worlds (Line 8), find deterministic plans from each of the resulting states (Line 9), and average over the resulting reward yielded in each plan. The action with the highest average plan reward over the sampled worlds is chosen to be executed (Line 14).

More formally, we define the value of being in a state s_1 as the maximum expected reward over

```
hindsight_optimization($N$)
 1. while true
 2.   best_action = null
 3.   world_samples = create N samples from belief
 4.   max_reward = 0
 5.   foreach $a_i$ : actions
 6.     average_reward = 0
 7.     foreach $s_j$ : world_samples
 8.       state = apply $a_i$ in $s_j$
 9.       reward = compute optimal plan from state
10.       average_reward += reward / N
11.     if average_reward > max_reward
12.       max_reward = average_reward
13.       best_action = $a_i$
14.   execute best_action
```

Figure 1-1: Very high level pseudocode for the general hindsight optimization algorithm.

plans that extend from s_1. That is, the maximum reward over all possible future action sequences, $\langle a_1, a_2, ... \rangle$, of the total reward over all expected future states:

$$V^*(s_1) = \max_{A=\langle a_1,a_2,...\rangle} \underset{\langle s_2,...\rangle}{E} \left[\sum_{i=1} R(s_i, a_i) \right]$$

where $R(s, a)$ represents the reward of performing action a in state s. In this planning setting, these future states incorporate only a single sample of the possible values of all uncertain locations, events and outcomes. Given our expectations about these uncertainties, we would like to find the action sequence $A = \langle a_1, a_2, ..., \rangle$ that maximizes the expected sum of action rewards. To compute V^* exactly, we would need to compute the expectation for each of exponentially many plans over exponentially many possible resolutions of uncertainty.

In hindsight optimization, we approximate the value function by exchanging expectation and maximization, so that we are taking the expected value of "maximum-reward plans instead of the maximum over expected-reward plans". This means we are taking the weighted average over optimal plans under each action instead of taking the weighted average over all possible plans under each action:

$$\hat{V}(s_1) = \underset{\langle s_2,...\rangle}{E} \left[\max_{A=\langle a_1,a_2,...\rangle} \sum_{i=1} R(s_i, a_i) \right]$$

This approximation of $V^*(s)$ uses fixed values for each of the uncertainties in each maximization. As in other applications of hindsight optimization, the stochastic elements have been reduced to known values by sampling: for each possible value that an uncertainty could take on in the expectation, the problem is to maximize reward given a known world, i.e., standard, deterministic, reward-maximizing planning. As the underlying deterministic problem becomes more difficult to solve with a standard deterministic planner, Tu-Hop can employ a limited horizon planner. A limited horizon planner uses a time horizon which is simply a temporal value by which search depth is bounded. Setting this bound to infinity results in an informed, full solution to the maximization problem, decreasing the horizon results in greedier behavior, only considering more immediate reward. To offset this greedy myopic behavior, a heuristic evaluation function can be used at the leaf nodes to help gauge what reward could be achieved if the plan were to reach all the way to a goal state.

After an action is executed, hindsight optimization updates its current belief state based on the outcome of its action and how it has affected the world. This newly updated belief state will be used in the next planning step. We define the Q-value to be the cumulative expected reward of taking an action a_1 in state s_1:

$$Q(s_1, a_1) = R(s_1, a_1) + \underset{\langle s_2, \ldots \rangle}{E} \left[\max_{A=\langle a_2, \ldots \rangle} \sum_{i=2} R(s_i, a_i) \right]$$

From this, we estimate the best action choice in s_1 as $\max_a Q(s_1, a)$. Using this technique, we are said to be performing optimization with the benefit of "hindsight" knowledge about how future uncertainty will be resolved.

1.3 Approach

After this initial discussion of hindsight optimization we explain how to handle the three types of uncertainties previously discussed; uncertain action outcomes and durations, uncertain object and agent locations and exogenous events. Our planner, TU-HOP, is an online planner that interleaves search and execution, emitting single actions for the agents to execute at a time.

The pseudo-code in Figure 1-2 provides a high level summary of the TU-HOP planner. The planner first receives three parameters, the first is the current belief about the world state, the second is the number of samples to be used and the third is the horizon with which to bound the deterministic solver. First, we generate a set of N possible worlds that are consistent with the planner's current belief about the world (Lines 2–3). Next, for each action a in the domain, we consider the resulting state $s' = a(s)$ (Line 5). Then, each possible world w_i is initialized with the state s', generating a fully-known deterministic planning problem. Solving this problem provides an estimate of the reward from s'. The mean reward across the set of samples (Line 6) along with the reward of the action $R(s, a)$ is used as the Q-value for each action a in the original state s (Line 7). We then select the action with the maximum Q-value (Line 8), this action is then executed (Line 9). The result of this action is returned and the current belief of the world state is updated with this

```
TU-HOP($W, N, H$)
 1. while true
 2.   for $i$ from 1 to $N$ do
 3.     $w_i \leftarrow sample\_world(W)$
 4.   foreach action $a$
 5.     $s' \leftarrow a(s)$
 6.     $r \leftarrow (\sum_{i=1} solve(s', w_i, H))$
 7.     $Q(s, a) \leftarrow R(s, a) + r$
 8.   $a_{best} \leftarrow argmax_a\, Q(s, a)$
 9.   res = execute($a_{best}$)
10.   update($W, a_{best}, res$)
```

Figure 1-2: The TU-HOP planner.

information (Line 10). With this new belief about the state of the world, the planner returns to the beginning of the loop and executes another iteration.

1.4 Robot Office Assistant Domain

To experimentally evaluate this technique, we focus on a specific domain where a set of controllable and non-controllable agents are able to navigate around a topological map containing objects in a Robot Office Assistant Domain. Controllable agents are able to *pickup*, *putdown*, and *give* objects. The give action is the transfer of one object in an agent's possession to another agent. All of the information about the domain and belief of the world state will now be discussed.

The first major entity in the world belief is a representation of the topological navigation graph. Each node in the graph is named and connected to another node in the graph via an edge. Each edge has a traversal success rate, a minimum and maximum successful traversal time and a minimum and maximum failure time. The success rate represents the probability that traversing this edge will succeed. The minimum and maximum successful traversal times provide an expected interval of how long it will take to traverse the edge if the traversal is successful. Similarly, the minimum and maximum failure interval describes how long it will take for the attempted traversal to report failure.

The objects present in the world each have a unique name, two associated temporal intervals and a set of node locations. The first interval is the minimum and maximum expected arrival time for the object. This can be used to represent an object being dropped off by an agent outside the control of the planner. The second interval is the minimum and maximum duration the object is expected to remain usable in the world. This can be used to impose a deadline on when an object may need to arrive at its goal destination. The set of nodes following the intervals contains at least one node name. A single node represents complete certainty of the object's starting location. A set whose size is greater than one represents uncertainty regarding the object's starting location.

Agents are very similar to objects with two major differences. The first is that an agent can be marked as outside of the control of the planner. This is useful if an agent is only "stopping by" to receive an object from another agent that *is* under the planner's control. The second difference is that each agent also has a number of grippers available to hold objects.

Goals can be one of three different types. The first is a simple *goto* goal which tells the planner which agent needs to be moved to which location and what the reward for this goal is. The second type of goal is *move*, which tells the planner which object needs to be moved to which location and what the associated reward is for completing this goal. The last type of goal is *give* and tells the planner which object should be given to which agent and how much reward will be received for achieving this goal. Not all agents and objects need be involved with a goal.

Lastly, there are four actions in the domain; *pickup*, *putdown*, *give* and *no-op*. Please keep in mind that the *move* action is defined on an edge-by-edge basis in the graph. The motivation behind this is that some edges may be more difficult or simply take more time to traverse. Each action has a success rate, minimum and maximum successful execution duration and a minimum and maximum failure duration with the exception of the *no-op* action. The success rate represents the probability that executing this action will succeed. The minimum and maximum successful execution times provide an expected interval of how long it will take to complete the action if the execution is successful. Similarly, the minimum and maximum failure interval describes how long it will take for the attempted execution to report failure. The *no-op* action is always successful and has a deterministic execution duration equal to the planned duration. The duration for each *no-op* is determined during planning. It is set to be the time from the current state, until the next occurring event. An event is simply the arrival or departure of an agent or object, or another agent completing its current action. Setting the duration in such a manner can render a planner incomplete in certain domains. The types of domains where this becomes an issue are those where a *no-op* and an action need to occur before an event occurs. Consider the simple example of serving ice cream. A very simple problem would be to make sure ice cream is ready to eat in 1 hour. The actions are *no-op* and *serve*. *Serve* only takes a few minutes to execute. By only reasoning about event time points like agents arriving, we would be forced to either apply a *no-op* and be late serving the ice cream which is not acceptable, or *serve* the ice cream and then apply a *no-op* for the remaining duration in which the ice cream would melt. Ideally, the *no-op* would be applied for 55 minutes and then serve would be applied. We do not need to reason about the intermediary values for a *no-op* in this simple Robot Office Assistant Domain while still maintaining completeness. We can do this because in this domain all the time points of interest occur at these event boundaries.

1.5 A Closer Look

All of the domain instance information is managed and updated in the belief state of the Tu-Hop planner throughout its execution. Before emitting any action to be executed, Tu-Hop enters its first planning iteration.

It begins by generating a set of possible deterministic world samples consistent with the current belief state. This means that in each sampled world any and all uncertainty is removed. This is achieved for the location uncertainty by picking, at random, one of the locations in each location set for the agents and objects. The action outcome uncertainty is resolved by using the success rate, success interval and failure interval for each action and constructing a deterministic mapping of times to outcomes and durations. We increase the complexity of the deterministic worlds by

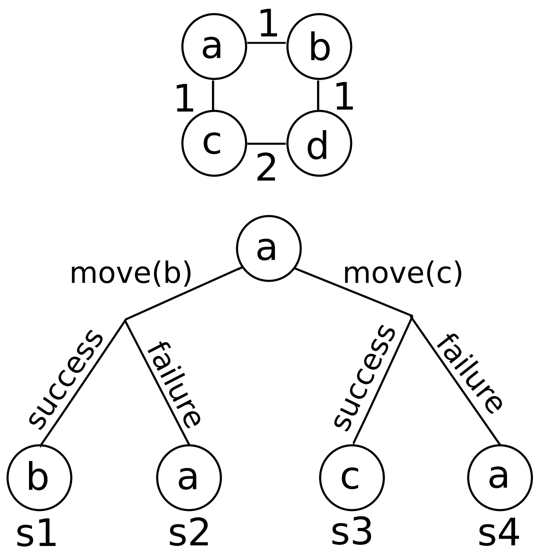


Figure 1-3: A simple example of the first step of hindsight optimization.

doing this, but conceptually it makes sense that an action may fail at one time and succeed the next. For example, if we mapped actions directly to outcomes regardless of time, then a *pickup* may never succeed and no solution could be found in that particular mapping. However, by mapping the outcomes by time we maintain the notion of actions sometimes succeeding and sometimes possibly failing. We can efficiently implemented this by lazily querying the action in the deterministic world sample when needed for its outcome and duration given the current time. If the time has no mapping, the outcome is randomly computed given the success and failure values, then stored in a map. If the time has a mapping already, that mapping is returned. These time values are rounded to a hundredth of a second before doing the lookup. The arrival and departure times of any object or agent are also resolved by taking a random sample from the arrival interval and the duration interval to construct an exact arrival and departure time.

Following the hindsight optimization framework, in each world sample, Tu-Hop examines each available action from the current state. In the most simple *goto* (navigation) case with no objects, Tu-Hop will evaluate what will result after moving to each node adjacent to its current node. In figure 1-3 the agent is currently in location (a) and is considering moving to location (b) or (c). Each move action can either succeed or fail as depicted. Each of these outcomes, $\{s_1, s_2, s_3, s_4\}$, is generated and then reward is maximized individually in each outcome. The reward for execution of the action is then a weighted mean of the reward achieved in the success and failure case weighted by the likelihood of the action succeeding and the likelihood of the action failing. For example, if the achievable reward under s_1 is 1, the achievable reward under s_2 is 0.5, and the the success rate for that action is 0.9 (0.1 failure rate), then the reward achievable for executing *move(b)* is 0.95 by Tu-Hop's reckoning.

This procedure is executed for each generated deterministic sample. Then the reward is averaged over all the samples yielding an estimate of achievable reward. The action with the highest expected reward across all samples is then selected for execution.

The action selected is then executed and the result of the action is used to update the belief state of the planner. This would include increasing the current time, removing a location from an object's possible location set, decreasing the size of an agent's arrival interval, and so on.

1.6 Experimental Results

We now evaluate Tu-Hop by stressing each of the three types of uncertainties (location, action outcome and temporal uncertainty). All experiments were planned and executed in simulation on a Lenovo W520 with an Intel Core i7 and 8GB of RAM. For each problem instance we ran 25 trials. In each trial we initialized the random number generator with a seed in the set $\langle 1 - 25 \rangle$. So in total we used 25 seeds. We also considered planner configurations which vary the number of world samples generated at each planning step. All plots presented show a line representing the mean y-value across the instances and vertical lines representing the 95% confidence interval at that point on the line.

1.6.1 Location Uncertainty

The first set of experiments begin by issuing the goal of moving an object from its start location to a goal location. The easiest instance starts with the object's location known exactly. We increase the difficulty of the instances by adding uncertainty about the objects location to possibly two locations, then possibly three locations and so on. As the uncertainty about the object's start location increases, the agent should be forced to search out the true start location. The results from this experiment using planner configurations of a horizon of 90 seconds and 1, 5 and 10 samples are shown in figure 1-4.

In all of these instances where the object is up to 5 possible locations, the planner is able to find the object and deliver it to its destination. We can also see by increasing the number of samples taken, the agent will visit the possible locations in a more reasonable order: first visiting the closest possible location, then moving to the next closest, and the next, until the object's true location is found. This is shown in figure 1-4 (b) and (c) as the lines representing 5 and 10 samples required fewer actions to achieve the goal and also an overall shorter goal achievement time. Figure 1-4 (a) shows the time required by the planner at each iteration. Even on the hardest instance with 10 samples the planner takes much less than a second on average before emitting an action for execution.

We also ran a small experiment to show the underlying planner's ability to scale with the number of object relocation goals. We start with a single goal of moving one object to a location and slowly increase the number of goals and objects in the world. The results from this experiment using planner configurations of a horizon of 90 seconds and 1, 5 and 10 samples are shown in figure 1-5.

In all instances the planner was able to move all the objects to their goal locations. We can see that as expected in figure 1-5 (a), using more samples does increase the overall planning time at each planning step. However, even in the hardest considered instance with 5 objects using 10 samples the planner only took on average 0.6 seconds before returning the next action. In figure 1-5 (b) and (c), a strange trend is shown where using more samples results in fewer actions in the final execution and also an earlier goal achievement time. This could be an artifact of not reusing the deterministic samples when possible. This could cause the samples in the 1 sample case to have different edge traversal durations between planning steps and the intent to retrieve a specific object can change as a result when an future edge is predicted to now have higher cost. When using more samples, this type of noise is minimized and more predictable behavior is achieved.

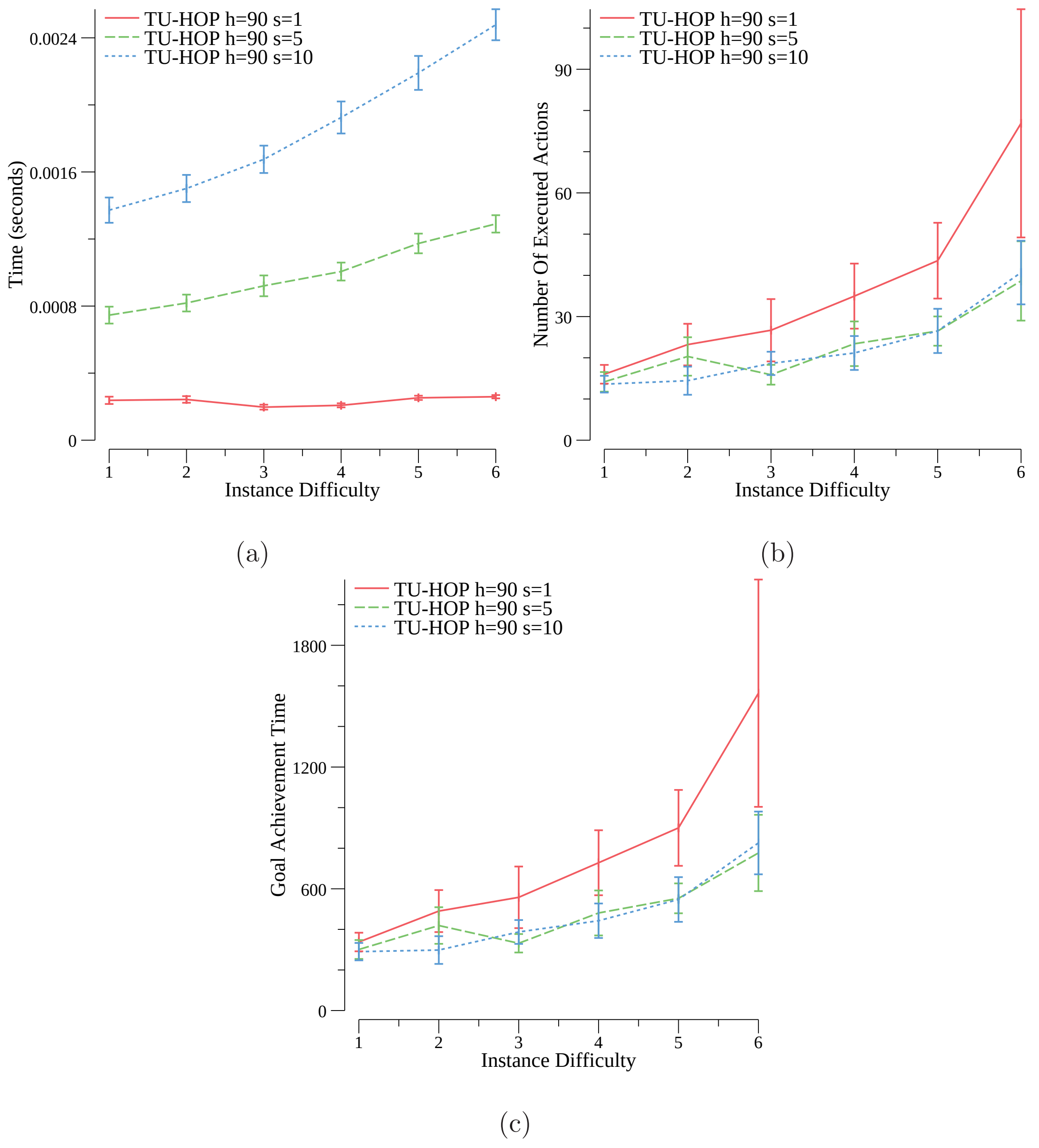
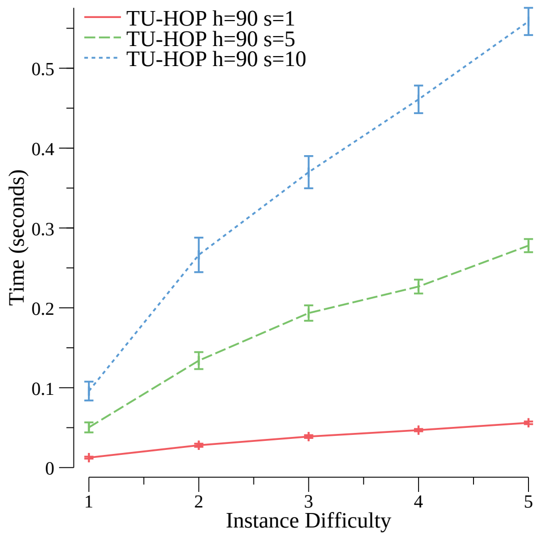
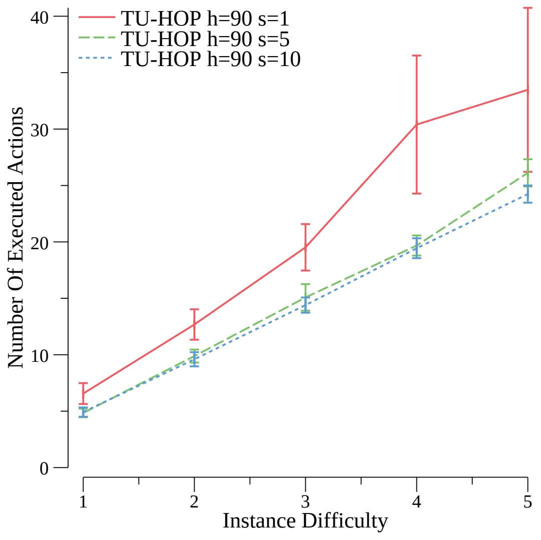


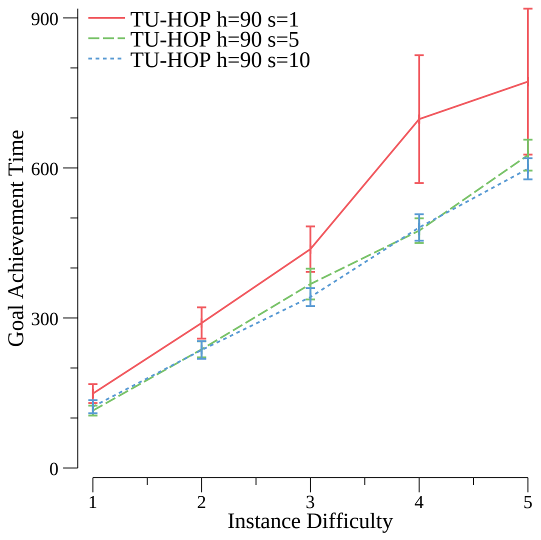
Figure 1-4: Increasing the amount of uncertainty in an object's location in the world between 1 and 5 locations using a horizon of 90 seconds and 1, 5 and 10 deterministic samples.



(a)



(b)



(c)

Figure 1-5: Scaling the number of objects in the world between 1 and 5 using a horizon of 90 seconds and 1, 5 and 10 deterministic samples.

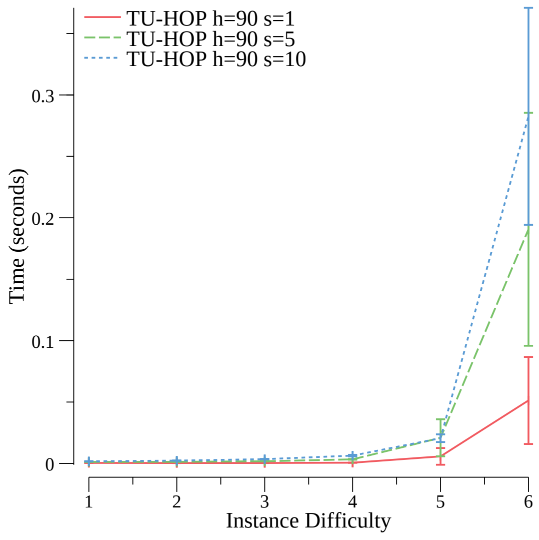
1.6.2 Action Outcome Uncertainty

In the second set of experiments we issue a similar goal of moving an object from its start location to a goal location. However in these instances the object's location is known exactly and the topological edge traversals required to achieve the goal will have increasing traversal failure rates. We start with a failure rate of 0% and increase it to 50%. By increasing the edge traversal failure rates, either plans will be generated expecting failed outcomes or the agent will be forced to re-plan and accommodate for the failure. The results from this experiment using planner configurations of a horizon of 90 seconds and 1, 5 and 10 samples are shown in figure 1-6.

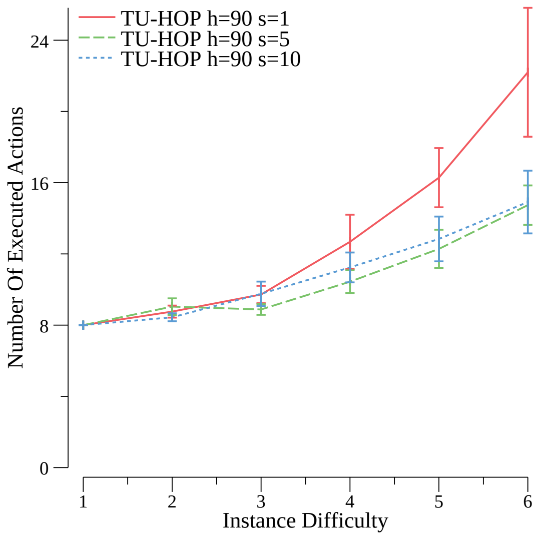
The planner was able to achieve all goals in all instances during this experiment. In figure 1-6 (a) we can see that the planning times for the 1, 5 and 10 sample cases are all quite similar until the edges become quite unreliable with only a 50% success rate. As the failure rate increases, the plan lengths will simply increase and planning with more samples magnifies this in its overall planning time. In figure 1-6 (b) and (c) we see a trend similar to the last experiment. This is most likely caused by the same issue. The noise between sampled worlds is minimized by generating more samples.

A simple third set of experiments extending the second set was also performed. The instance is created with a set of inexpensive topological edge traversals between the agent and the object's start locations with high action failure rates. A secondary set of expensive edge traversals with very low failure rates are also created. As the inexpensive route becomes more unreliable throughout the experiments, the agent should choose to take the more reliable expensive route. The results from this experiment are shown in figure 1-7.

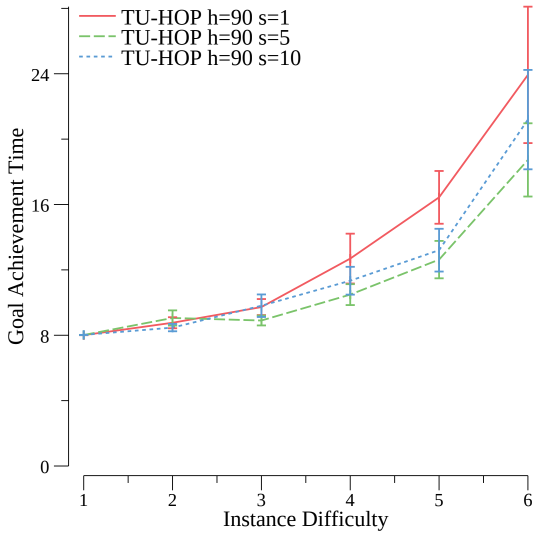
Again, in this experiment the planner was able to achieve all goals in all instances during this experiment. Figure 1-7 (a) shows a predictable trend where increasing the number of samples causes the planning step between action executions to increase. However, planning times with 10 samples on the most difficult instance are still well below 0.1 seconds on average. Figure 1-7 (b) and (c) show the ability to trade between cost and edge reliability. When using only a single sample the number of actions in the final execution is lower for the first three instances, but the overall goal achievement time for those instances is higher than when using 5 and 10 samples. Once the



(a)

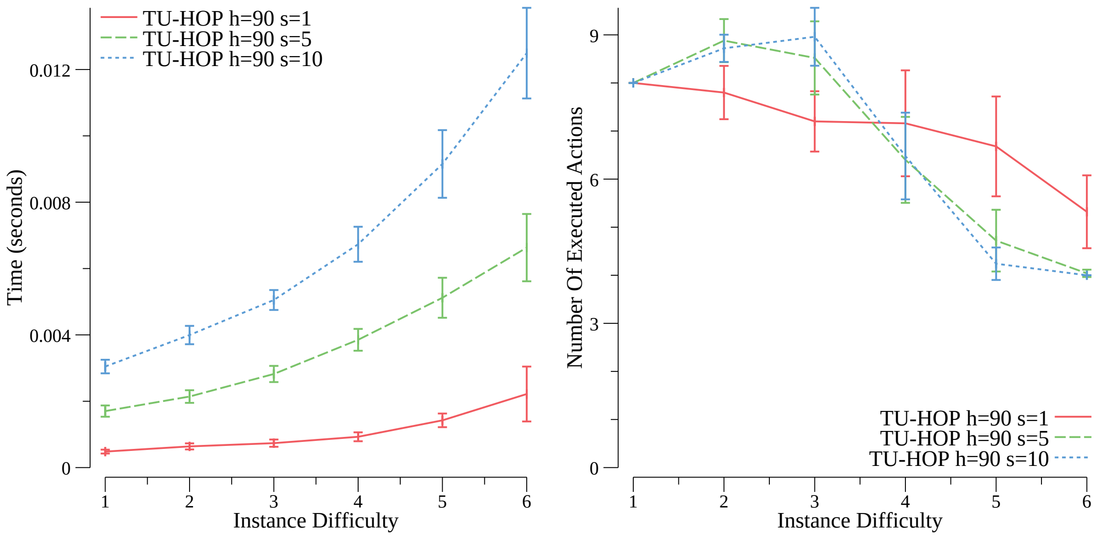


(b)

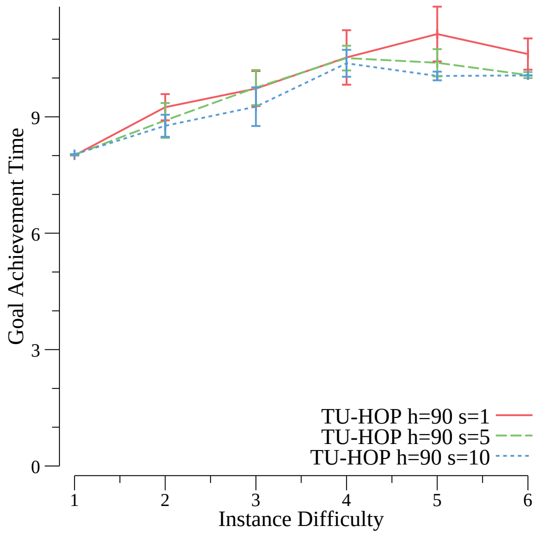


(c)

Figure 1-6: Decreasing the reliability the only edges available to achieved an issued goal using a horizon of 90 seconds and 1, 5 and 10 deterministic samples.



(a) (b)



(c)

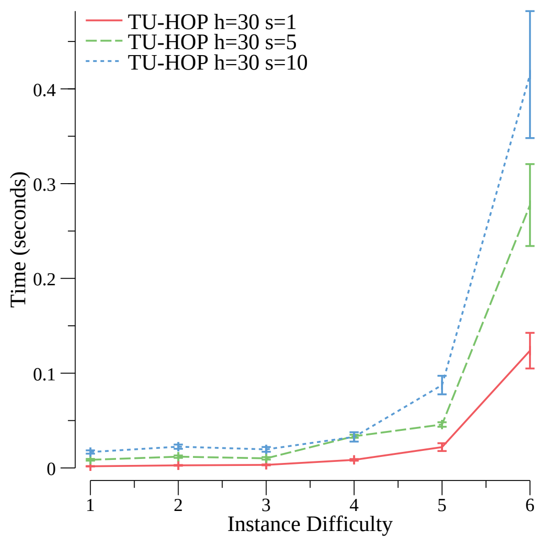
Figure 1-7: Decreasing the reliability of low cost edges forcing more reliable expensive edges to be utilized using a horizon of 90 seconds and 1, 5 and 10 deterministic samples.

reliability of the cheap edges decreases significantly in the last three instances, the single sample case starts to have longer execution lengths and continues to have later goal achievement times than the 5 and 10 sample cases.

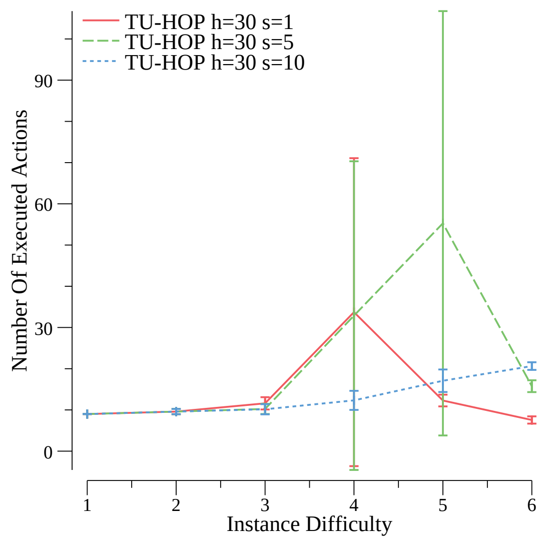
1.6.3 Temporal Uncertainty

In this fourth set of experiments we involve a second agent outside of the control of the planner. The goal issued in this set of experiments is to give an object to this second agent and also relocate a second object. Adding a second object goal forces the planner to choose an ordering for the two goals which becomes very important in this experiment. The second agent does not begin at any location in the domain model but is scheduled to arrive at one during a predefined interval and will only remain for a duration between some minimum and maximum value. We begin with this second agent starting with a small arrival interval and a long duration before departing. We increase the difficultly of these instances by making the arrival interval larger (more uncertain) and decreasing the duration the agent remains before departing. As the arrival uncertainty grows and the waiting duration shrinks, we're increasing the possible time the agent could be there, while decreasing the actual time he will remain. You can imagine a synonymous situation of moving an archer further away from the target (increasing their field of vision) while also decreasing the size of the target they are trying to hit. The results for this experiment are presented in figure 1-8.

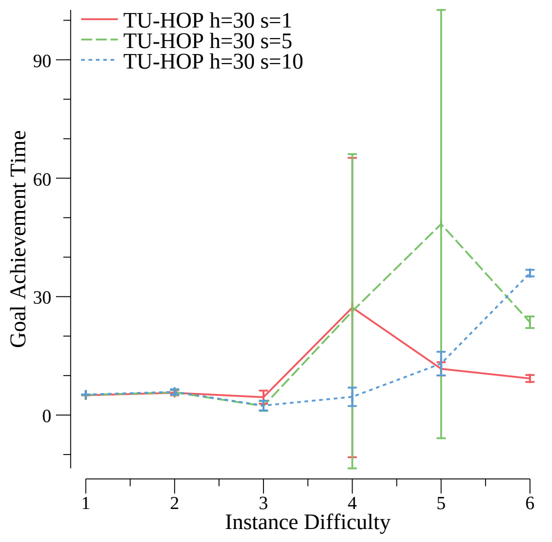
Figure 1-8 (a) shows that planning times between action executions are still less than 0.5 seconds which is certainly acceptable when the action executions times for a robot can be in the range of full seconds to a minute for some actions. In figure 1-8 (d) the reliability of both goals being achieved is illustrated. At first with the larger delivery window, the 1, 5 and 10 sample cases are able to achieve both goals reliably. However when the window is reduced, the planner clearly benefits from more samples as the 10 sample case is able to achieve all goals in all but the hardest instance. In the hardest instance none of the planner configurations are able to deliver the object to the second agent. Figure 1-8 (b) and (c) are slightly more difficult to interpret because if the planner incorrectly chooses to relocate the second object first and misses the delivery window, the planner will terminate with fewer actions and an earlier goal achievement time than an algorithm



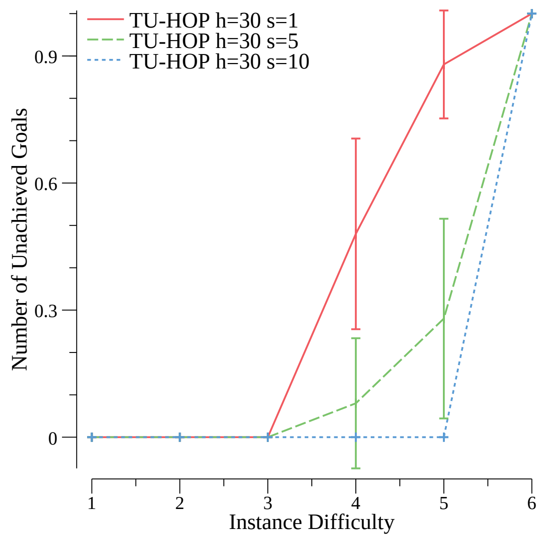
(a)



(b)



(c)



(d)

Figure 1-8: Increasing the uncertainty about when an agent will arrive while also decreasing the duration the agent will wait after arriving using a horizon of 30 seconds and 1, 5 and 10 deterministic samples.

that achieves both goals.

1.7 Discussion

The main two attractive features of hindsight optimization are its simplicity and generality. There are no obvious impediments to combining the current work with other efforts that use hindsight optimization to address other forms of uncertainty such as arrival of additional goals, partial observability, or open worlds.

Compared to a planner using an STNU, Tu-Hop's handling of intervals is imprecise, thus the combinations of circumstances that are anticipated is incomplete. This limitation gets more serious as the number of combinations of stochastic events that need to be considered increases. However, in many applications, it is not necessary to reason about such long chains of events in order to act successfully.

Tu-Hop demonstrates one way of very tightly coupling planning and acting, namely to ensure reactivity by planing after every state transition and never explicitly committing to actions beyond the one that is currently executing.

Unlike many other task planners, Tu-Hop does explicitly consider action failure when selecting actions.

It does not output a complete plan that can be shared with other collaborating agents. However, it should be possible to merge together the actions selected in each rollout to form a branching contingent plan that could be shared. Such sharing could then be represented in the planner by increasing the cost of actions that do not correspond to those in the shared plan. This directly models the coordination costs that the group would sustain if the plan were to be changed.

Hindsight optimization is often used with a limited horizon planner. When this is done, it places some responsibility on the heuristic evaluation function used at the leaf nodes of the search to correctly identify promising states. An alternative is to use hierarchical planning, in which a complete plan exists at some level of abstraction, and detailed planning is then done on those parts that are ready for execution. Such an approach has been proposed by Kaelbling and Lozano-Pérez (2011).

Hindsight optimization is an unsound reasoning technique. UCT is a popular sampling-based technique that is sound, in the sense that it is guaranteed to select the optimal action given an infinite number of samples. Eyerich, Keller, and Helmert (2010) compare hindsight optimization with UCT on the Canadian Traveler's Problem. While they find that UCT does indeed converge better in the limit of many samples, hindsight optimization performed better when the methods were given only a moderate number of samples.

1.8 Conclusion

Uncertainty is an unavoidable aspect of real-world robotic applications. We have shown how hindsight optimization yields a simple and general approach to planning with location, action outcome and temporal uncertainty. While the technique is approximate, it is easy to implement and our results suggest that it can be successful in practice. This work was published in a 2014 ICAPS PlanRob paper (Kiesel & Ruml, 2014).

CHAPTER 2

Open Worlds

2.1 Introduction

In this chapter we transition to another type of uncertainty. We move away from temporal uncertainty to focus on open world uncertainty. We will begin with a real world example to motivate this area of research.

Imagine a rescue robot entering a partially-destroyed building to search for survivors of an earthquake. The agent does not know the initial layout of the building, what new obstructions may exist, the locations of potential victims, or even how many victims there are. In open-world planning problems like this, the agent is not given a complete description of the initial state of the world, but it can perform sensing actions to determine the existence of relevant objects and the values of important fluents. To be useful, the planner must be fast enough to not materially delay the actions of the robot. It must be able to plan to discover and take into account newly sensed information, and ideally it would be expressive enough to handle soft goals, durative actions, temporal constraints, and actions with uncertain outcomes.

In this chapter, we continue with the simple on-line hindsight optimization planning approach that we used in the previous chapter, but now extend it to handle the requirements of open-world domains. We call this new approach *Optimization in Hindsight with Open Worlds* (OH-wOW). Rather than using traditional techniques that compute a policy or contingent plan in advance, we estimate on-line at each step which action is best in light of our current knowledge of the world. The OH-wOW approach is domain agnostic and does not commit to a particular representation for open-world knowledge or goals. Instead, it can leverage any closed-world planner appropriate for the underlying domain. Our central assumption is that the agent possesses some knowledge, likely probabilistic, about the domain. In our view, performing well in an open-world depends on having expectations about that world, e.g., building dimensions are typically tens or hundreds of meters rather than centimeters or kilometers, or that people are usually found in certain densities per square meter, or are more often found in certain areas, such as offices. This type of default or prior information can be overridden by direct experience, but ought to play a role in planning until it is discovered to be inaccurate. We use these expectations to generate possible states of the world consistent with the agent's current knowledge, use a closed-world planner to estimate the future reward achievable in those worlds after taking each currently-applicable action, and then select the action with the lowest expected cost.

After describing OH-wOW in detail, we contrast it with previous work. We then report on the method's empirical performance, both in simulated domains and when deployed on a physical mobile robot fully integrated with the Robot Operating System (ROS), Simultaneous Localization And Mapping (SLAM) and standard navigation. Our experience indicates that the method is surprisingly general and practical, achieving results as good as those of previous systems but with lower planning times and fewer ad hoc assumptions. This work showcases the power of Monte Carlo techniques and adds open-world planning to the list of non-classical planning settings in which simple planners can be leveraged to provide state-of-the-art performance.

2.2 A Hindsight Optimization Approach

Optimization in hindsight was originally developed for scheduling and networking problems (Chong et al., 2000; Mercier & van Hentenryck, 2007; Wu et al., 2002) and has recently been applied to probabilistic planning (Yoon et al., 2008, 2010). In these previous settings, sampling is used to resolve uncertainty in the outcome of actions. In our context of open-world planning, each sample forms a concrete hypothesis about the world—which objects might exist and which fluents might hold. While these will likely be revealed to the agent as it performs actions that have, a priori, uncertain outcomes, the sampling process for open-world planning is more involved than choosing an outcome in a PPDDL (Younes & Littman, 2004) or RDDL (Sanner, 2011) action description. For example, a rescue robot will generate possible world states with conceivable floor plans for the building, each with sets of victims distributed in various plausible locations. Each of these sampled worlds may potentially determine the outcome of multiple sensing actions during the course of the corresponding planning episode. Demonstrating the practicality of this approach is the central contribution of this work.

While these samples of possible worlds are intentionally not exhaustive, they are intended to provide useful relative judgements on the expected value of actions. In order to estimate the value of an action, we apply that action in each of the sampled possible worlds, find closed-world plans from the resulting states, and average over the resulting plan costs. The action with the lowest average plan cost over the sampled worlds is chosen to be executed.

More formally, we define the value of being in a state s_1 as the minimum expected cost over plans that extend from s_1. That is, the minimum cost over all possible future action sequences, $\langle a_1, a_2, ...\rangle$, of the total cost over all expected future states:

$$V^*(s_1) = \min_{A=\langle a_1,a_2,...\rangle} \underset{\langle s_2,...\rangle}{E} \left[\sum_{i=1} C(s_i, a_i) \right]$$

where $C(s,a)$ represents the cost of performing action a in state s. In open-world planning, these future states incorporate the sensed knowledge of the agent and the expectation is over the distribution of sensing outcomes. The agent will expect different outcomes based on its beliefs about the world. Given our expectations about sensing outcomes, we would like to find the action sequence $A = \langle a_1, a_2, ...\rangle$ that minimizes the expected sum of action costs. To compute V^* exactly, we would need to compute the expectation for each of exponentially many plans.

In optimization in hindsight, we approximate the value function by exchanging expectation and minimization, so that we are taking the expected value of minimum-cost plans instead of the minimum over expected-cost plans:

$$\hat{V}(s_1) = \underset{\langle s_2,s_3,...\rangle}{E} \left[\min_{A=\langle a_1,...,a_{|A|}\rangle} \sum_{i=1}^{|A|} C(s_i, a_i) \right]$$

This approximation of $V^*(s)$ uses fixed sensing outcomes in each minimization. As in other applications of optimization in hindsight, the stochastic elements have been reduced to known outcomes by sampling. For each possible outcome in the expectation, the problem is to minimize cost given a known world, i.e., standard, closed-world, cost-minimizing, deterministic planning. In OH-wOW, fixed sensing outcomes are generated using concrete hypotheses about the state of the world. For each fully-known, deterministic world hypothesis, the agent can compute the result of different sensing outcomes when solving the minimization in the equation for V^*. For example, the result of querying a vision system to look for an injured person depends on whether or not there is an injured person in the sensed portion of the world—this is fully-known for each hypothesis. The agent is aware of what features are truly known and which are merely hypothesized, as a result the deterministic problem can require sensing actions before the agent interacts with hypothesized portions of the world. In this way, the system will still be required to plan to sense. A dummy precondition is added to all actions that involve a hypothesized variable. This precondition enforces that the value of that variable is sensed before actions requiring the value are executed. This ensures that the resulting plan executes sensing actions appropriately. More concretely, if the agent hypothesizes that there is an injured person in a room, then the deterministic planner will require a sensing action before that person can be reported. When a sensing action is carried out in the physical world, its result may differ from the hypothesis. This new information will be reflected in the samples taken at the next planning step.

We define the Q-value to be the cumulative expected cost of taking an action a_1 in state s_1:

$$Q(s_1, a_1) = C(s_1, a_1) + \underset{\langle s_2, s_3, \ldots \rangle}{E} \left[\min_{A = \langle a_2, \ldots, a_{|A|+1} \rangle} \sum_{i=2}^{|A|+1} C(s_i, a_i) \right]$$

From this, we estimate the best action choice in s_1 as $\min_a Q(s_1, a)$. Using this technique, we are said to be performing optimization with the benefit of "hindsight" knowledge about how future uncertainty will be resolved.

The pseudocode in Figure 2-1 summarizes the algorithm. At each time step, the algorithm is used to find the next action to execute from the current state s, which includes information about both the agent's current configuration and its current knowledge about the world. First, we generate a set of N possible worlds that are consistent with the agent's current knowledge (lines 11–12). Next, for each currently applicable action a, we consider the resulting state $s' = a(s)$ (line 14). Then, each possible world w_i is initialized with the state s', generating a fully-known

```
OH-wOW($s = \langle agent, world \rangle, N$)
11. for $i$ from 1 to $N$ do
12.     $w_i \leftarrow sample\_world(world)$
13. foreach action $a$ applicable in $s$
14.     $s' \leftarrow a(s)$
15.     $c \leftarrow (\sum_{i=1}^{N} solve(s', w_i))/N$
16.     $Q(s,a) \leftarrow C(s,a) + c$
17. Return $argmin_a\, Q(s,a)$
```

Figure 2-1: The OH-wOW algorithm.

closed-world deterministic planning problem. Recall that, to incorporate sensing, the determinized problem requires the agent to sense before interacting with hypothesized features of a sampled world. Solving this problem provides an optimistic estimate of the cost from s'. The mean cost across the set of samples (line 15) along with the cost of the action $C(s,a)$ is used as the Q-value for each applicable action a in the original state s (line 16). Finally, we return the action with the minimum Q-value (line 17), the agent executes the action, possibly observing new facts and objects in the world, yielding a new current state, and the cycle begins anew.

2.3 Related Work

Open world planning is a broad problem that has been attacked from many angles. One issue is how to formally represent knowledge and goals related to open-ended sets; Etzioni and Weld (1994) and Babaian and Schmolze (2006) have addressed this. We do not address this issue in this chapter, except to point out that the underlying planners used in our approach are closed-world and do not require a particularly expressive (and expensive) representation language. We do require that the agent tracks what is currently known about the world and that the world generator respects this knowledge when sampling possible worlds.

In conformant planning (Cimatti, Roveri, & Bertoli, 2004, inter alia), one requires plans that are guaranteed to work without sensing. For most robotics domains, this is overly restrictive and renders problems unsolvable. Contingent planning (Meuleau & Smith, 2003, inter alia) allows for sensing, but computes a plan before beginning execution. In addition to handling open-worlds, we aim to scale to domains in which the number of contingencies may be very large (e.g., the number of possible floor plans), making synthesis of branching plans prohibitively expensive.

In the POMDP literature, computing actions on-line is recognized to provide increased scalability (Ross, Pineau, Paquet, & Chaib-draa, 2008). However, many POMDP algorithms attempt to compute future belief states of the agent, which can be expensive and cumbersome. Optimization in hindsight represents an extreme approach, disregarding future belief uncertainty and assuming that the agent can achieve the cost accrued by the plans for the fully-observed sampled worlds. Our work is perhaps most closely related to work on sampling techniques for POMDPs, where a particle filter approximates the belief space during sampling (Silver & Veness, 2010). Open world planning goes beyond traditional factored POMDP representations (Boutilier, Dean, & Hanks, 2011) because the structure of the world state requires representing a logically infinite domain of discourse; the universe of objects that exist and the possible relationships between them remain unknown to the agent (Doshi, 2009).

There has been sustained interest from roboticists in open-world planning. One way of handling open-world planning in practice is to force the robot to move in one direction simply to explore without a concept of cost or reward. Such simple ad hoc approaches cannot exploit the agent's expectations about goals (e.g., people are likely in offices) or take sensed information into account (e.g., a hallway implies new rooms to explore). Talamadupula, Benton, Schermerhorn, Kambhampati, and Scheutz (2010) present an approach where the planner assumes objects exist in order to instantiate goals and motivate a search and rescue robot to collect reward by discovering and reporting victims. As new information arrives about the environment, the planner replans. This can be seen as a degenerate form of our hindsight approach, where the robot operates on a single optimistic "sample". While it is simpler, it cannot generalize to domains where uncertainty is a major component.

Joshi, Schermerhorn, Khardon, and Scheutz (2012) use offline symbolic dynamic programming with known goals but unknown numbers or locations of objects, which does allow for reusable policies on any instance of the domain. However, in their experiments the number of possible objects was severely limited to retain feasible computation times (they require 4 hours for their 3 room example), which makes the resulting policies suboptimal. They also do not handle temporal constraints, action costs, or goal rewards.

2.4 Evaluation

We evaluate OH-wOW by applying it in two domains: the classic omelette benchmark for planning under uncertainty, and urban search-and-rescue, which we investigate both in simulation and using a physical robot.

2.4.1 Omelettes

In the omelette benchmark, introduced by Levesque (1996), the agent is attempting to make a three-egg omelette with ingredients of unknown freshness. The agent has four available actions. The agent can *break* an egg into a bowl, *pour* the contents of a bowl into another bowl or the trash, *wash* a bowl, or *sniff* whether the eggs in a bowl are good. All actions are deterministic except for the sniff action. The goal is to have exactly three good eggs in a specific bowl with no trace of bad eggs. To make the domain more challenging, we extended it to have both regular white eggs, which are bad with a probability of 0.5, and local brown eggs, which are bad with a probability of 0.1. The agent is able to observe the color of the next available egg without requiring a sensing action.

We compared OH-wOW to a perfectly omniscient oracle and also to a hand-coded controller. The controller puts eggs into the goal bowl, sniffing after each addition and cleaning out bad eggs until it finds a good one. Then it does the same routine using an extra bowl, pouring good eggs into the goal bowl from the extra bowl until the goal is reached. We generated three sets of 100 random instances, each set with a different probability of the next egg being brown. OH-wOW used a domain-dependent deterministic planner based on uniform-cost search.

Figure 2-2 shows the distribution of the resulting plan costs using box and whisker plots. Each box surrounds the middle 50% of the data, with a horizontal line indicating the median and whiskers

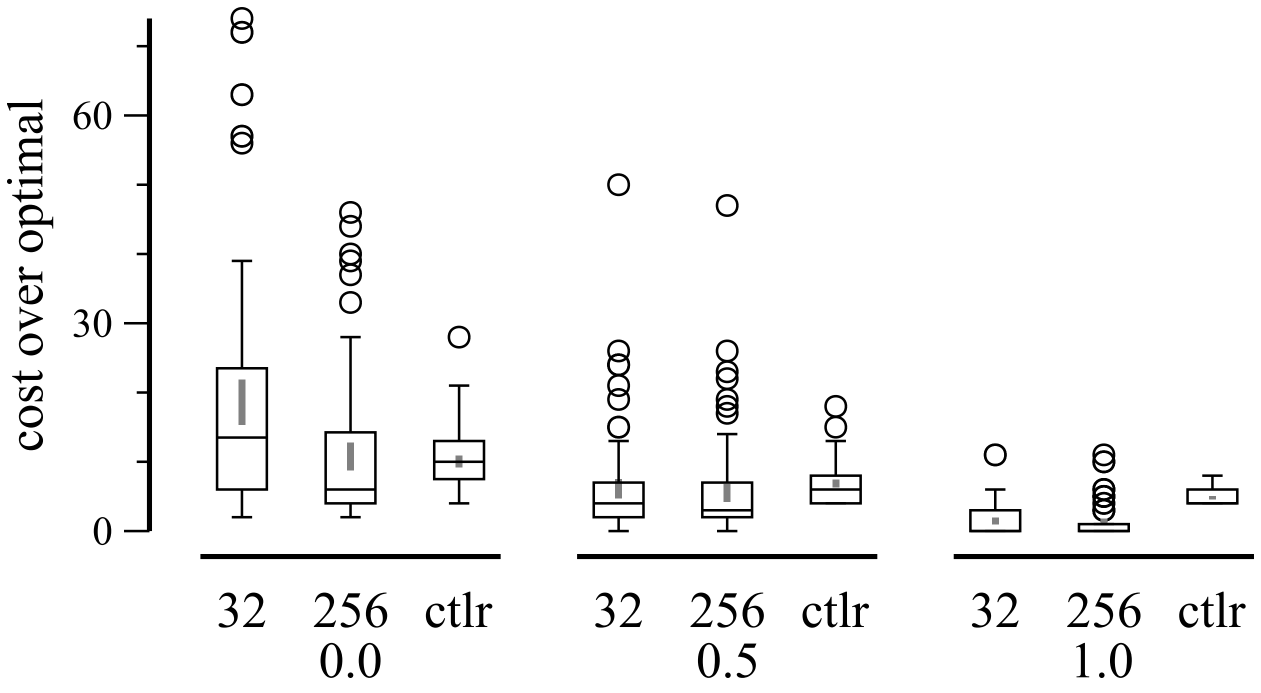


Figure 2-2: Plan cost in the three-egg omelette domain.

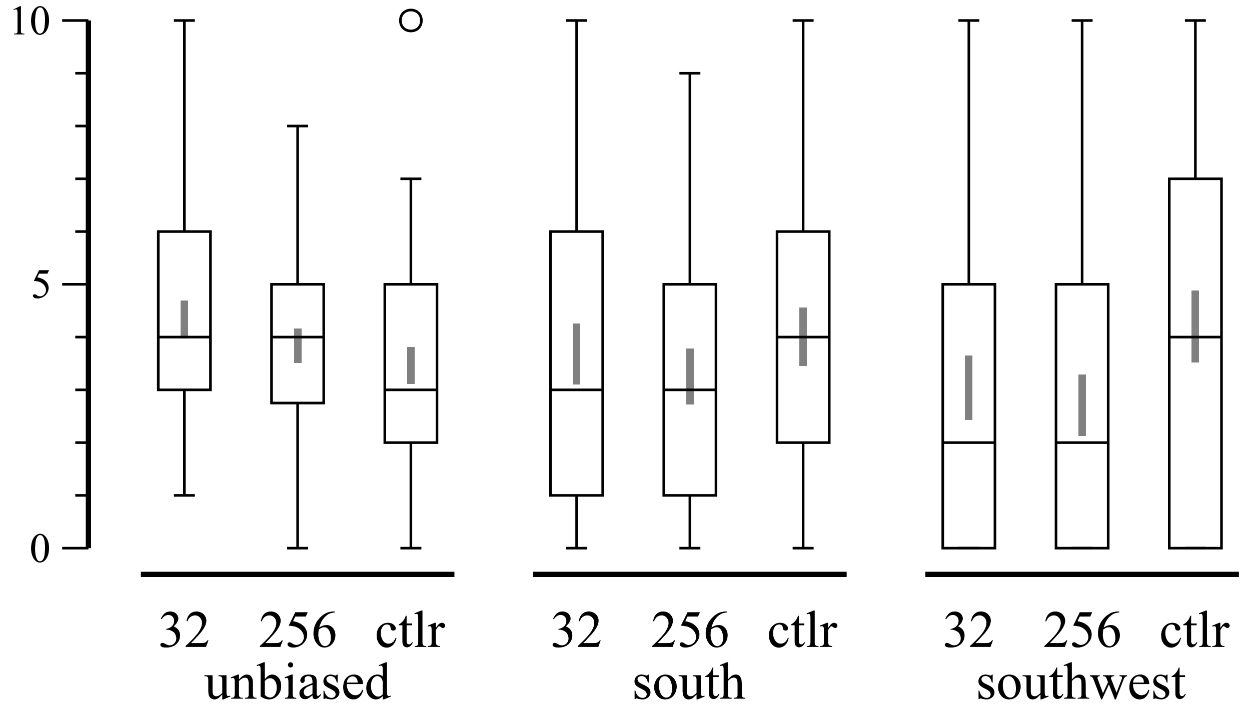


Figure 2-3: Plan cost in the search and rescue domain.

indicating the range (values beyond 1.5× the inter-quartile range are shown as circles). The gray vertical stripes inside each box show 95% confidence intervals on the mean. The plot shows the increase in cost over the optimal solution found by the oracle, of the hindsight planner with 32 and 256 samples, and the hand-coded controller (ctlr). The boxes are grouped by the probability of an egg being brown (0.0, 0.5, and 1.0). We can see that, when all eggs were white, the hindsight planner with 256 samples had a median cost that was less than the hand-coded controller (significant with $p < 0.05$ via the Wilcoxon signed-rank test). As the probability of a brown egg increased, the hindsight planner performed better, nearly dominating the controller when all eggs were brown. This is likely because the hindsight planner could recognize that brown eggs tend to be good, and put multiple into a bowl before bothering to smell, saving redundant sniff actions.

The average total planning time on a 3.1 GHz Core2 PC for OH-wOW to reach the goal using 256 samples on a problem without brown eggs was 12.9 seconds (standard deviation 8.0 seconds). Each plan was an average of 24.9 actions long (standard deviation 13.7 actions) and each action in the plan took an average of 0.52 seconds to select (standard deviation 0.31 seconds) before executing it. This compares favorably with the 185 seconds of offline planning reported for approximate RTDP (CPU unspecified) by Bonet and Geffner (2001). Levesque (2005) also generates full plans offline to solve the three-egg omelette in 1.4 seconds but requires 1,681 seconds if the omelette is scaled to four eggs. When using four eggs, OH-wOW's costs relative to optimal were similar to the three-egg case, and total computation time averaged only 76.7 seconds (standard deviation 43.6). Each plan was an average of 49 actions long (standard deviation 27.3 actions) and each action in the plan took an average of 1.57 seconds to select (standard deviation 0.99 seconds) before executing. The plans found by Levesque's planner also contain strictly more actions than our hand-coded controller (which in turn finds more costly plans than OH-wOW on the median), as Levesque's solution always uses the auxiliary bowl for staging and requires an additional pour action to move the first good egg into the goal bowl.

2.4.2 Search and Rescue

Now, we return to the motivating example of search and rescue robotics. The robot's objective is to maximize the number of injured people it reports while still returning to its starting location by a given hard deadline.

To generate possible worlds for OH-wOW, we need to generate building layouts consistent with the robot's current map and hypothesize the possible locations of injured people. We represent building layouts as rough topological maps. We assume that undiscovered nodes will lie on a uniform four-connected grid, and that a known node can be extended if it has an adjacent grid cell that can be reached without going through an obstacle or crossing an existing edge in the map. We iteratively choose an extendable node, generate a valid neighbor and connect them. We use a bias toward extending the most recently added node, and toward generating the neighbor that forms a straight line from the chosen node's parent. This was sufficient to yield plausible building layouts with hallways. Victims are generated independently with fixed probability per hypothesized node. The upper right panel of Figure 2-4 shows a very small example map with hypothesized extensions shown in gray.

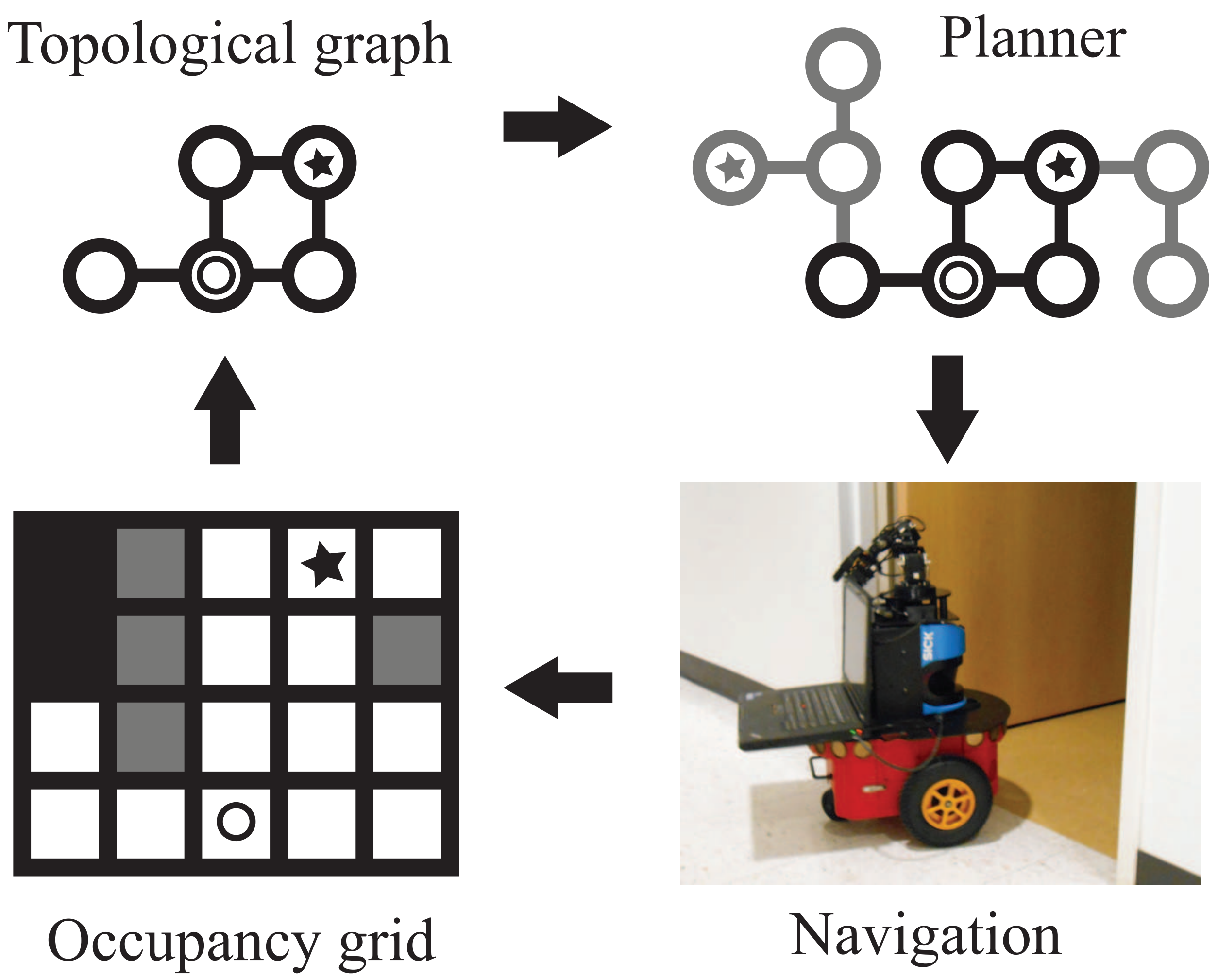


Figure 2-4: Architecture diagram.

The base planner used by OH-wOW precomputes all-pairs shortest-paths among nodes containing people and the start location. It then uses depth-first search, considering at each step to visit each unreported person or return home. The available actions depend on the remaining time. For efficiency, we avoid considering time as a separate state variable by incorporating it into the cost function (Phillips & Likhachev, 2011).

Simulation

To test the planner in simulation, we created 100 random worlds with 100 nodes each. We considered three victim distributions: *unbiased*, uniform probability of 0.1 per node; *south*, nodes south of the start location contains a person with probability 0.2 and nodes north of the start location 0; and *southwest*, southwest of the start 0.4 and 0 elsewhere. These distributions are representative of helpful domain knowledge that can be leveraged when generating possible worlds. Skewing the probability of a victim's existence to one side of the building could be used to represent the knowledge of a closed wing of the building or a scheduled company-wide event. We limit the total number of victims to 10. The cost of a plan is the number of unreported people remaining when the agent returns home and performs a dummy *finish* action. We compared OH-wOW to two different algorithms. The first is an oracle that knows the exact configuration of the building and location of all victims. The second is a hand-coded controller that performed a depth-first exploration of the building, reporting people that it encountered and returning to the start location when it had no more time to explore.

To gauge the complexity of these instances, we must consider the number of possible configurations of maps and victims. Considering only $n \times n$ grids, there are $2(n-1)n$ possible places for edges; our generator is limited to trees, so it must pick n^2-1. For $n = 10$, this is $\binom{180}{99} \approx 10^{52}$ maps. For each possible map, we must choose locations for victims; for 10 victims, there are $\binom{100}{10} \approx 10^{13}$ possible configurations on the map. Maintaining a belief over so many possible worlds would be challenging. Thankfully, it also seems unnecessary if we merely wish to estimate the expected value of actions.

Figure 2-3 shows results, grouped by the victim distributions. For the unbiased case, the

	victims found			
deadline	0	1	2	3
1 minute	4	6	0	0
5 minutes	0	7	3	0
10 minutes	0	3	4	3

Figure 2-5: Number of injured victims found and reported over 10 runs using a physical robot.

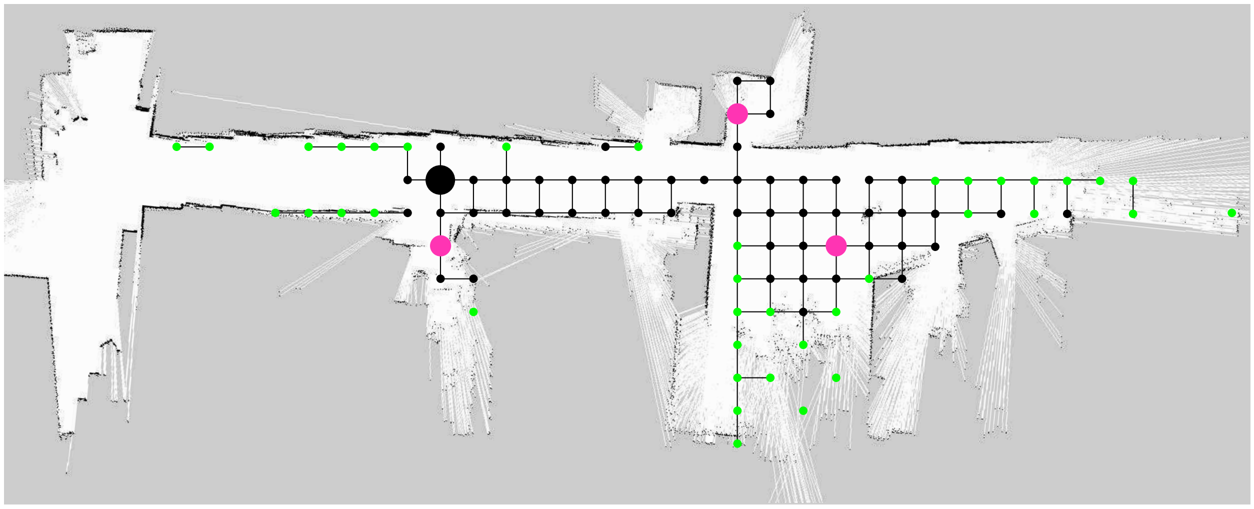


Figure 2-6: Example SLAM and topological map.

hand-coded controller gave the best performance, but OH-wOW was quite competitive. With a biased distribution, OH-wOW was superior as it was easily able to leverage prior knowledge about possible worlds. The average maximum per-action planning time for OH-wOW with 256 samples was 2.7 seconds (standard deviation 0.85 seconds). In order to compare with Joshi et al. (2012), we also ran smaller instances with at most three victims. The average maximum per-action time for 256 samples was 0.18 seconds (standard deviation 0.035 seconds), which is negligible compared to typical mobile robot latencies.

Physical Robot

We also integrated OH-wOW with the Robot Operating System (ROS, `www.ros.org`) on a 3.7 GHz quad-core i7 laptop onboard a Pioneer 3dx equipped with a SICK LIDAR shown in Figure 2-4.

We use the ROS Gmapping SLAM stack to generate a fine-grained occupancy grid, from which we extract a topological map with edge lengths of 1 meter to provide to OH-wOW. Figure 2-6 shows a final topological map overlayed on the corresponding SLAM map created during an experiment run of the search and rescue application. In the topological map, nodes are marked as either black, green, or pink. Pink nodes indicate an area of the building where a victim was found and reported. Green nodes are points in the map that can be extended when creating possible building layouts. The black nodes indicate that the layout of the building can not be extended from this area.

The ROS Navigation stack is used to execute movement actions, which are specified as the topological node to visit next. These topological nodes are then mapped to a two-dimensional point in the map built by SLAM before issuing the move action to the robot. In some cases, the rough topological graph places a node very near to an obstacle and the motion planner can not find a safe way to achieve the requested action. We supplemented the navigation component in these instances by issuing a set of perturbed points around the initial point before returning failure to the planner. This set was simply four points, one in each cardinal direction, one half of a discretization away.

We performed experiments in a hallway of approximately 20 meters with between 2 and 5 open doors to offices and 3 victims. We simulated detection of a victim using the range capability of the laser rangefinder. When the laser is able to collect data and populate a portion of the map corresponding to certain pre-selected locations (that were unknown to the planner), we pass that detection information along to the planner. In order to report a victim the robot must navigate to the containing topological node.

We used three different deadlines, one minute, five minutes and ten minutes. As shown in Table 2-5, the performance of the robot improves as it is given more time to search for victims. In all experiments the robot returned within the hard deadline we provided. At first, only given a short deadline of one minute, the robot is able to find one out of the three victims in six of the ten trials before returning home. When the deadline is increased to five minutes, the robot takes advantage of this and performs more exploration and is able to find two out of the three victims in two trials and one victim in the remaining eight. When this deadline is further increased to ten minutes, the robot is able to find all three victims in two trials, two victims in four trials, and one victim in the remaining four trials.

These results demonstrate that generating possible worlds consistent with experience is feasible in practice, even as the robot's knowledge is being updated during exploration. It also shows that under realistic conditions, OH-wOW correctly trades off soft goals under temporal constraints, but without the ad hoc goal handling of Talamadupula et al. (2010) or the hours of preprocessing required by Joshi et al. (2012).

2.5 Discussion

OH-wOW requires a generative model of plausible worlds. We assume such expectations can be developed either manually or through experience. When the world contradicts the agent's expectations, this can be interpreted as surprise, which might naturally lead to increased learning. The fundamental vulnerability of sampling-based planners is when unlikely worlds play a large role in determining action value; importance sampling may help here. For example, in the "Bombs in Toilets" domain (McDermott, 1987; Smith & Weld, 1998), OH-wOW may never sample a world in which a certain undunked package contains the bomb. The probability of this, however, is small ($7 \cdot 10^{-11}$ for 6 packages and 128 samples). In any case, optimal behavior is unattainable if one insists on fast response times in dynamic domains.

While faster than many POMDP algorithms, OH-wOW is much slower than a classical planner, as it must solve one classical planning problem for each sampled world. In our implementation, during each step, all planning problems were solved serially. These problems are entirely independent though and could trivially be solved in parallel to take advantage of multiple processor cores. OH-wOW is more general than standard off-line techniques as it can be used on-line, as shown in the experiments, and also off-line by simulating the domain to construct a branching plan. It is possible to improve the performance of OH-wOW by applying some of the enhancements of Yoon et al. (2010). One such technique is called *probabilistically helpful actions*. To find probabilistically helpful actions, the planner evaluates all samples from the current state of the agent instead of the one step lookahead states. Actions that lead to optimal plans starting from the current state are considered to be helpful while the others are not. The samples are solved as normal from the one step lookahead states, but the only actions that are considered are the ones that were deemed helpful. Another improvement presented by Yoon et al. (2010) is to save samples and plan prefixes that remain consistent with the outcome of a selected and then executed action. In domains with large amounts of determinism, this enhancement can greatly reduce the amount of planning required by saving work across deterministic transitions.

In this chapter, we assume that the world remains static as we explore it and that non-sensing actions are deterministic. OH-wOW however, is very general and immediately applies to dynamic worlds, stochastic actions, and on-line goal arrival; this remains an exciting area for future work.

2.6 Conclusion

Open world planning is essential for many real-world agents. We have shown how optimization in hindsight yields a simple and general approach to open-world planning with temporal constraints, decision-theoretic reasoning, and soft goals. While the technique is approximate, it is easy to implement and our results suggest that it can be successful in practice.

In this chapter, we were able to demonstrate the benefits of Hindsight Optimization in an Open World search and rescue domain. This further expands the realm of applicability of this non-classical planning technique from its traditional uses to handle an additional kind of uncertainty. We were able to plan and execute entirely online while still maintaining good performance. This work was published in an 2013 ICAPS PlanRob paper (Kiesel et al., 2013) and in a University of New Hampshire technical report (Kiesel et al., 2012).

In Part 1, we applied Hindsight Optimization to planning with temporal uncertainty and to planning in Open Worlds. We showed that leveraging this non-classical planning technique in the face of uncertainty can achieve good performance in our domains of interest.

Part II

Abstraction for Motion Planning

In Part I of the dissertation we examined ways to increase the expressiveness of hindsight optimization to handle aspects of uncertainty in robotics applications. In Part II we increase the efficiency for the quintessential robotics planning setting: motion planning.

As is common in AI, we exploit abstractions to assist in guiding motion planning. In Chapter 3, an abstraction is used to guide scheduling and motion planning in terms of vehicle allocation, subgoals, and expected temporal cost of a solution segment. The abstraction allows the motion planner to focus on exponentially smaller pieces of the state space. This allows us to solve large problems quickly.

In Chapters 4 and 5, abstraction are also employed to help guide motion planning, However, here we focus on sampling-based motion planning. The benefit of abstraction and heuristics is already established in another form of motion planning, lattice-based motion planning. Lattice-based motion planning can directly apply many of the ideas from heuristic search, while sampling-based motion planning requires some non-traditional modifications to bridge the gap. Our work focuses on sampling-based motion planning because of its smaller number of assumptions and probabilistic completeness guarantees. In some cases there are asymptotic optimality guarantees that are not qualified by discretization resolution.

By re-interpreting recent heuristic search techniques and ideas under the framework of sampling-based motion planning, we are able to find solutions faster than the previous state of the art. We are then able to leverage this speed along with cost reasoning to provide more robust performance across a wide variety of experiments. We show that using these non-classical planning ideas from heuristic search we can outperform the previous state of the art sampling-based motion planners.

CHAPTER 3

Task and Motion Planning

3.1 Introduction

There has been much interest recently in problems that combine high-level task planning with low-level motion planning. As techniques for high-level task planning and low-level motion planning each mature, there has been interest in how they might be integrated together to improve overall system performance. We will apply non-classical planning ideas to a robotics task and motion planning domain which is composed of several NP-Complete sub-problems. Without the use of non-classical planning techniques to help focus search effort, this problem could not be solved for any reasonably large instance size. This is because often, decisions at the high level, such as who will do what and in what order, depend on low-level considerations, such as the existence or cost of feasible motions for particular tasks. A straightforward approach would be to combine both the task and motion planning problems and then solve them all at once with a single search algorithm such as A* (Hart, Nilsson, & Raphael, 1968). Because of the exponential nature of such problems, however, this approach is intractable for even small instances. Alternatively, both the task and motion problems could each be solved independently by first finding a task-level plan and then solving the motion planning problem for each task. While this approach is usually feasible, it can lead to poor solutions, or even incompleteness. This is because the task planner has incomplete knowledge of the cost and dynamics that will be utilized by motion plans when achieving its tasks, and furthermore, the motion planner is focused on the individual tasks without considering the constraints from a full plan perspective.

This chapter makes two main contributions. First, we present a new problem that requires the combination of task and motion planning, called Waypoint Allocation and Motion Planning (WAMP). While the problem is easy to understand and compact to specify, it presents timely research challenges. Again, it consists of scheduling a fixed set of vehicles to achieve different waypoint locations according to given temporal constraints. At the high-level, it is a resource allocation problem in which waypoints must be assigned to vehicles. For each vehicle, an ordering of the waypoints must be found such that temporal constraints can be met. At the low level, it is a difficult motion planning problem where a physically feasible path that respects the vehicles' motion models must be found such that each waypoint is visited and, again, all temporal constraints are met. The solution cost depends on the low-level paths that are selected. As we describe below, many of the subproblems of WAMP are known to be NP-hard. We also prove that the target value search problem (Kuhn, Schmidt, Price, Zhou, & Do, 2008), which is related to WAMP's routing subproblem, is NP-complete.

The second contribution is a planner that we have developed to solve WAMP instances involving fixed-wing aircraft. We combine tabu search for waypoint allocation, linear programming for scheduling, and heuristic search for route planning. The planner separates the high-level scheduling and resource allocation from the low-level routing by using a *surrogate objective* that is optimized by the high-level solver as a proxy for the true objective of the problem. This greatly reduces the number of times the router needs to be called. The low-level planner has the ability to give feedback to the high-level sequencer to help improve the accuracy of the surrogate objective. We present experiments that demonstrate the infeasibility of using one single A* search to solve this problem. Then, we test the scalability of our planner and evaluate the performance of its major components. We also show that our planner is able to solve realistic problems within the required time limit. This work illustrates how real world robotics applications can feature the combination of multiple interacting planning problems, requiring the integration of diverse non-classical solution techniques.

3.2 Problem Formulation

WAMP is directly motivated by an industrial application. An instance of WAMP is given by a 6-tuple $\langle Size, V, W, R, C, K \rangle$ where $Size = \langle x_{max}, y_{max} \rangle$ is the problem's x and y dimensions, V is a set of vehicles, W is a set of waypoints, R is a set of relative temporal constraints between

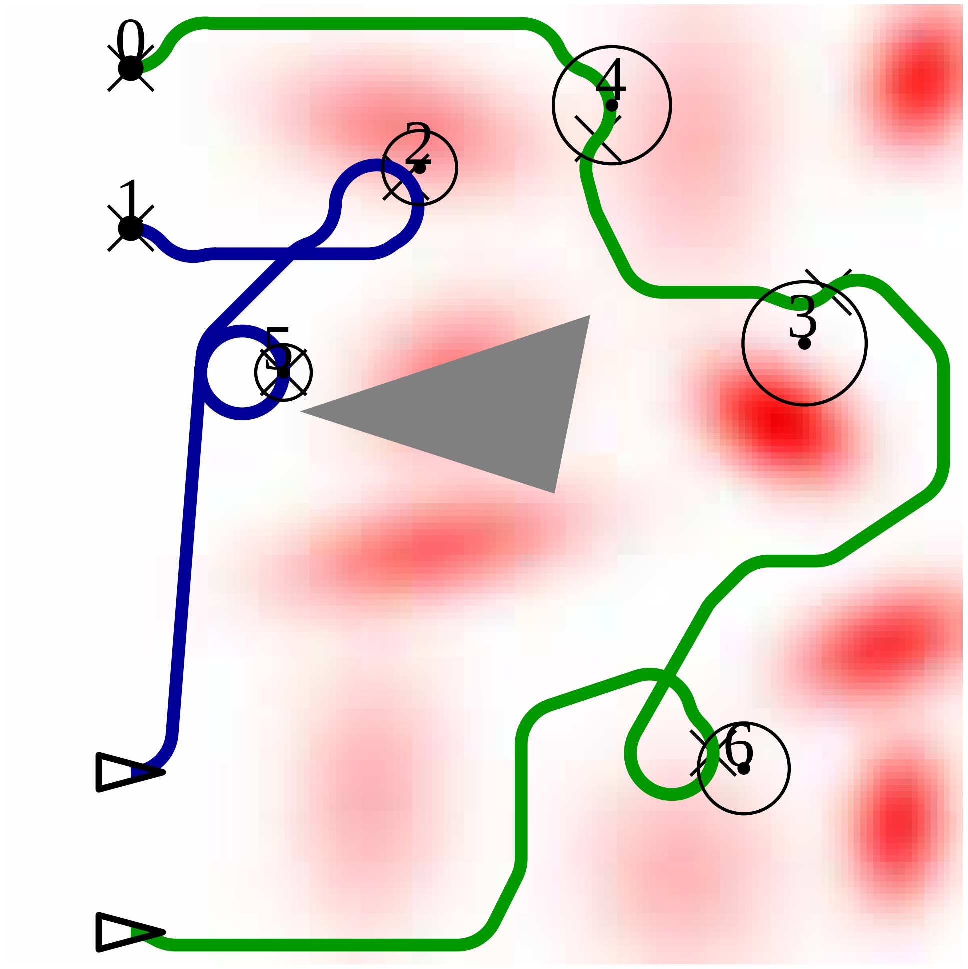


Figure 3-1: An example solution with 2 vehicles and 6 waypoints.

waypoints, C is a set of high cost regions and K is a set of non-traversable regions. The state space is restricted to two dimensions and vehicle collisions are not considered.

Vehicles In our instances, all vehicles are airplanes, so each element of the set V is a 5-tuple $\langle x_0, y_0, \theta_0, v, r\rangle$ where x_0, y_0 and θ_0 define the vehicle's initial position and heading, v is the vehicle's velocity and r is the minimum turn radius. In this chapter only fixed velocity vehicles are considered. In Figure 3-1 (a), the vehicles start poses are depicted by small black triangles.

Waypoints Each waypoint in the set W is represented as a circle and is defined by the 8-tuple $\langle x, y, r, t_s, t_e, \theta_0, \theta_1, A\rangle$, where x, y and r give the center point and radius, t_s and t_e give the start

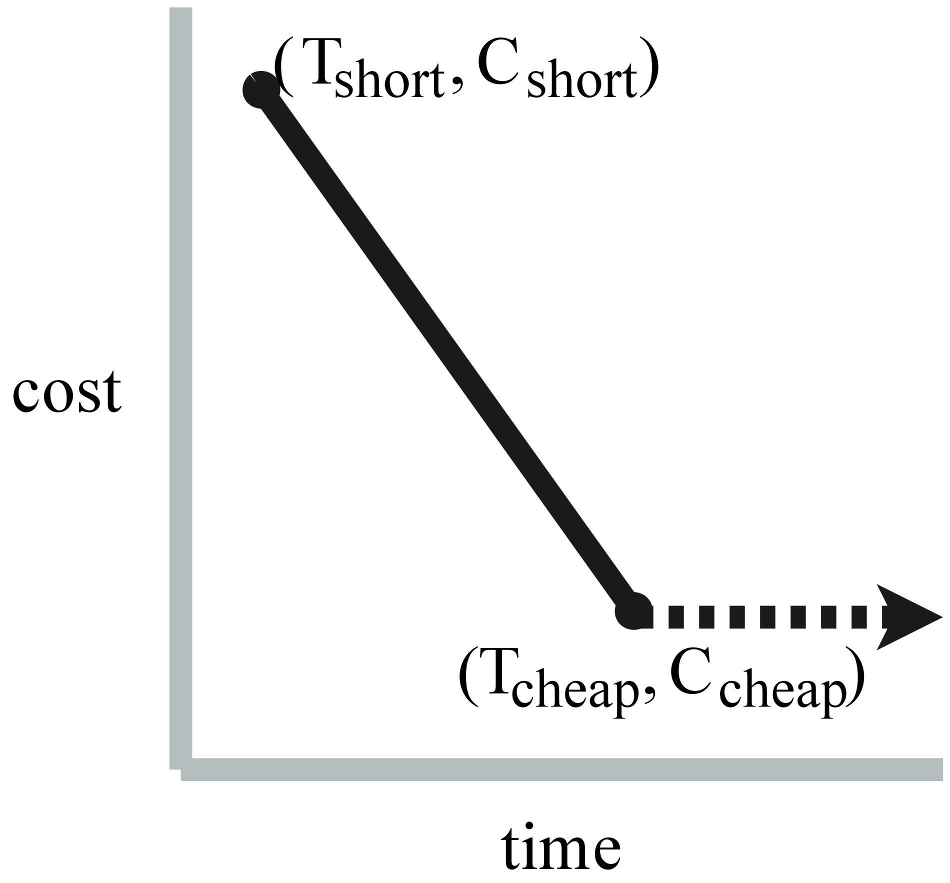


Figure 3-2: An example of time/cost trade-off.

n	Failure Rate
1	24%
2	64%
3	88%
4	98%
5	98%
6	100%

Figure 3-3: Sample A* scaling results

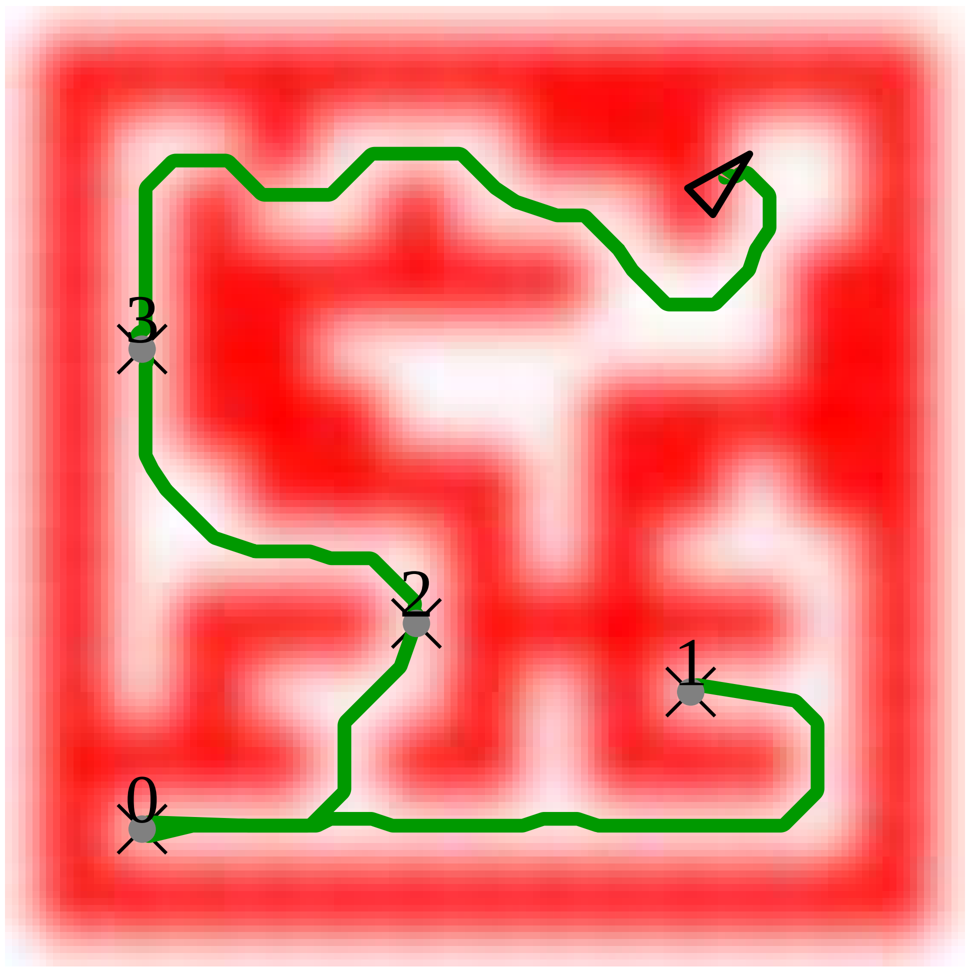


Figure 3-4: A router stressing maze example solution.

and end times of the window during which the waypoint must be achieved, θ_0 and θ_1 define a range of headings that the vehicle must be within when the waypoint is achieved and $A \subseteq V$ is a set of vehicles that are allowed to achieve that waypoint. The waypoints can be seen in Figure 3-1 (a) as numbers with circles. Each waypoint must be achieved by being within the circle at a legal time at a legal heading.

Relative Temporal Constraints In addition to each waypoint having an absolute time window, the set R defines a set of relative constraints. Each is a 4-tuple $\langle u, v, min, max \rangle$ where $u, v \in W$ are waypoints and min and max are the minimum and maximum allowable time difference between when u is achieved and v is achieved.

Costs C is a set of two dimensional Gaussians: $\langle x, y, h, \sigma_x, \sigma_y, c \rangle$, where x and y give the center location, h specifies the "height", or the cost that will be incurred at the center, the σ terms give the standard deviation in the x and y directions respectively, and c is the correlation. These are used to determine the cost of vehicle motion. There is also a minimum cost present everywhere representing fuel consumption and time. Every vehicle traverses a path, and the cost incurred by the vehicle is the time it spends in each location multiplied by the cost of being in that location. As the time discretization approaches zero, the limit is the line integral along the vehicle's path divided by the vehicle's speed. In Figure 3-1 (a), the cost of each cell is represented by the shade of red in the cell. Being in a white area incurs low cost, and being in a red area incurs high cost.

Keepouts Lastly, K is a set of "keepout" zones that cannot be traversed. Each zone is a triangular area defined by three points $\langle x_0, y_0, x_1, y_1, x_2, y_2 \rangle$. These shapes can be aligned to create more elaborate regions. In Figure 3-1 (a), the gray area is a keepout zone.

The cost of a solution is the sum of the cost incurred by all vehicles. However, after a vehicle achieves its final, waypoint it no longer accumulates cost. The objective of WAMP is to find a minimal cost solution, using the available vehicles, that achieves the given waypoints and meets all of the problem constraints.

3.2.1 Application Context

The planner must solve problems within ten seconds because it is part of an interactive decision-support aid with a human in the loop, who edits the resulting plans. The planner's solution might not be immediately acceptable because the Gaussian cost model is only an approximation of the real cost model and there may be other assets that are not modeled in the instance. We are also interested in pseudo-balanced vehicle makespans. Therefore our planner's objective was altered to take into account not only cost but an adjustable ratio between path cost and makespan. This tradeoff allowed us to specify how interested we were in minimizing time and cost as a weighted sum of the two final values:

$$obj = w_1 \cdot time + w_2 \cdot cost$$

3.2.2 Relations to Other Problems

We provide brief sketches showing how WAMP can be seen as a composition of several problems that are known to be NP-hard. We also provide an NP-completeness proof for one subproblem that, as far as we are aware, was not previously known to be NP-complete.

Vehicle Routing Problem with Time Windows VRPTW is a popular problem in the operations research community. While it now has a large number of variants, the classic VRPTW (Dantzig & Ramser, 1951) is: given a set of service requests with known fixed distances between request locations, find a schedule such that the required number of vehicles and total travel cost are minimized (in that order) and all requests are serviced within their given time windows. One variant that is closely related to our problem is the m-VRPTW problem (Lau, Sim, & Teo, 2003) where the number of vehicles is fixed at some value m and the goal is to find the minimum cost schedule with this fixed size fleet. The decision variant of this problem (determining if a feasible schedule exists) has been shown to be NP-complete (Savelsbergh, 1985; Lau et al., 2003).

The m-VRPTW problem can be reduced to a WAMP instance if each of the m vehicles has infinite capacity and the delivery destinations reside in the Euclidean plane. The reduction can then be achieved by setting the turning radius of each of the m vehicles to ϵ with a fixed velocity of 1. All vehicles share a start and end location, the location of the depot. Each delivery destination location and time window are direct mappings from the original problem. Using a large Gaussian to distribute cost uniformly over the map, such that the cost of each point on the map evaluates to 1, will result in a WAMP objective function that directly minimizes the overall distance traveled.

WAMP also shares similarities with the OR Orienteering Problem and its extensions (Vansteenwegena, Souffriaua, & Oudheusdena, 2011), specifically the Team Orienterering with Time Windows problem. Our task assignment solving techniques are similiar to those used in the OR research for the Orienteering Problem. The major difference from the traditional Orienteering Problem and WAMP is that the OR problem operates on a discrete graph with a discrete cost/reward function while WAMP operates in continuous space with a continuous cost function and obstacles.

Jobshop Scheduling Problem JSP is perhaps the most well-known scheduling problem. The JSP is an NP-complete problem (Garey & Johnson, 1991) concerning a given set of jobs, each composed of a set of activities that each have a given length. All activities must be assigned time on a given set of machines so that no two activities use the same machine at the same time and each activity must be serviced by a specified machine. The problem is to determine whether or not a feasible schedule exists within a given deadline.

We reduce the JSP to WAMP by associating vehicles with machines and waypoints with tasks. For each machine m, there is a special location l_m located sufficiently far from the special locations for all other vehicles that flying between any two special locations takes longer than the deadline. Each activity to be scheduled on machine m corresponds to three unique waypoints, the first one is placed at location l_m, the second is placed at a distance from l_m that corresponds to half of the length of the activity and the final one is placed at l_m. These waypoints have temporal constraints such that the first must be achieved first, the second one, that is not located at l_m, must be achieved exactly half the activity duration after the first and the final one must be achieved exactly half of the activity duration after the second. When a vehicle chooses to achieve the first waypoint for this activity, it cannot achieve any other waypoints besides the remaining two for this activity and the entire time to achieve all three must be equal to the activity duration. Finally, the activities are ordered within their respective tasks by constraining the last waypoint for an activity to precede the first waypoint for the activity that follows it within the task.

Traveling Salesman Problem TSP is a classic NP-complete problem (Garey & Johnson, 1991). The Euclidean variant of the TSP may be reduced to WAMP by creating an instance with uniform cost, a single vehicle, a turn radius that is infinitely small (the vehicle can turn and point itself directly at its next waypoint) and by placing the cities of the TSP at their respective x and y locations. The vehicle is able to traverse this set of waypoints within the given cost bound if and only if there is a solution to the TSP within the bound.

Target Value Search While WAMP is defined on a continuous space, our solution uses discrete search-based methods, thus it is useful to understand the complexity of related discrete problems such as Target Value Search. The TVSP (Kuhn et al., 2008; Schmidt, Kuhn, Price, de Kleer, & Zhou, 2009) is the problem of finding a path from start to goal whose length is equal to the target value. Schmidt et al. (2009) conjectured that the optimization variant of the TVSP (i.e., finding a path with cost as close as possible to the target value) is in EXPTIME. Kiesel et al. (2012) showed that the decision problem of determining whether or not a path with the exact target value exists is NP-complete. We include the proof here for completeness.

We specify a TVSP instance as a 4-tuple $\langle G, s, g, T \rangle$, where $G = \langle V, A \rangle$ is a finite graph with vertices V and a set of weighted, directed arcs $A \subseteq V \times V \times \mathbb{N}$, $s \in V$ and $g \in V$ are the start and goal nodes in the graph, and $T \in \mathbb{N}$ is the target value.

Theorem 1. *The target value search problem in a graph is NP-complete.*

Proof The problem is in NP since, given a solution, one can easily check the validity of the path and sum the edge weights in polynomial time. We show it is NP-hard by reducing from SubsetSum (Garey & Johnson, 1991). Given an instance of SubsetSum — a finite set $S \subseteq \mathbb{N}$ and a positive integer B for which we wish to know if there exists a subset of S that sums to B — we formulate a target value search problem as follows: $T = B$ is our target value. For each $s_i \in S$, we create a vertex v_i. The vertices are then linked together in a chain with two arcs between adjacent pairs of

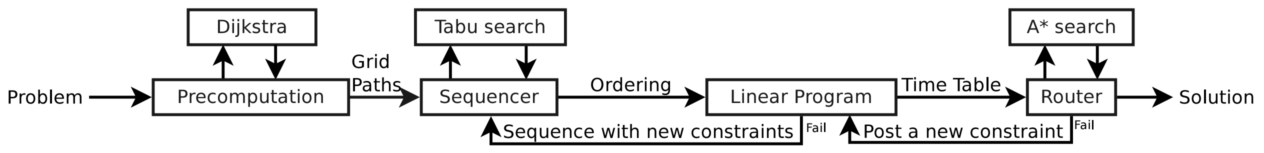


Figure 3-5: Overview of our system.

vertices, one arc has cost 0 and the other has a cost equal to the element of S corresponding to the first vertex of the arc: $(v_i, v_{i+1}, 0) \in A$ and $(v_i, v_{i+1}, s_i) \in A, 0 \leq i < |S|$. Finally, our start vertex is $s = s_0$ and our goal vertex is $g = s_{|A|-1}$.

There is a path in this graph that achieves the target value T if and only if there is a subset of S whose sum is equal to B, with the non-zero-cost arcs corresponding to the elements included in the subset. □

3.3 Our Approach

Our approach is guided by four features of the application context that we exploit to make the problem easier to solve: first, the cost function is relatively smooth, meaning that similar paths will often have similar costs. This allows us to approximate final path cost by evaluating the cost of a simple 8-way grid path. This implies that we can postpone detailed motion planning until we have a promising candidate solution. Second, there are many possible low-level paths, so we can make the assumption that any schedule will be routable given sufficient time per leg. This allows us to assume that a spectrum of paths exists between the fastest (most expensive) and the cheapest (relatively long) (see Figure 3-2 (b)). As we explain below, this spectrum is constructed optimistically and we therefore will incorporate feedback from motion planning as necessary to refine the estimates of achievable paths. Third, making a leg longer can usually decrease cost of the final route, as the vehicle has more time to navigate around high cost regions. Finally, it is easy to make a leg of a route longer, because if the route arrives at the destination waypoint too early, then extra time can easily be added by inserting loops into the route at low-cost locations. This means we can focus on trying to arrive early.

More specifically, our planner uses four stages: pre-computation, scheduling, building a timetable and routing (see Figure 3-5). First, we pre-compute information about times and costs between pairs of waypoints. This information will be used by the later stages to approximate the time and cost between pairs of waypoints. Next, the sequencer assigns waypoints to vehicles and then orders them such that there should be a feasible route for each vehicle that obeys the problem constraints. After an assignment and ordering are found, we use a linear program (LP) to find a *timetable* that specifies, for each waypoint, the time at which its assigned vehicle should arrive. The timetable is then passed to the router to find a flyable path for each vehicle that achieves the given times.

The information about routability used by the sequencer and LP is approximate, so there are two places where the procedure may fail. The first level of failure is when the router is unable to connect two waypoints and meet the deadline. This can happen because the initial estimates are approximate. If this happens, the router posts additional problem constraints, which are then used by the LP and sequencer to improve the accuracy of their estimates. The new constraint may make the the current ordering infeasible, at which point a new ordering is constructed. The following subsections describe each of these steps in greater detail.

3.3.1 Pre-computation

Both the sequencer and the timetable generation phases need to estimate the cost and duration of possible routes between each pair of waypoints, as depicted in Figure 3-2 (b). These estimates are represented by a linear interpolation between the quickest path and the cheapest path between each set of waypoints. The slope of this line represents an estimate of the rate at which adding additional time navigating on a leg can be converted into cost reduction, which we call the *improvement slope*. These shortest and the cheapest paths between the waypoints are computed in an 8-connected grid discretization of the problem where the discretization is the size of the smallest vehicle's turning radius. Each grid cell uses a single traversal cost estimate given by the mean of the true cost sampled at a fixed number of points distributed uniformly over the cell. In our implementation, the cost between two adjacent cell centers is the distance multiplied by half of the cost of each cell. To find the shortest and cheapest paths to each waypoint, we use Dijkstra's algorithm from each waypoint to all cells in the grid (once for each metric). Since the costs and distances are invertible, this gives an estimation of the shortest and cheapest paths to the given waypoint from anywhere in the problem. While the sequencer only needs waypoint to waypoint estimates, the router needs a heuristic value for every cell.

3.3.2 Sequencer

The sequencer finds an ordered assignment of waypoints to vehicles that is thought to be feasible given the problem constraints. For this step, we use a tabu search based on the technique for m-VRPTW described by Lau et al. (2003). WAMP however, has a handful of additional constraints such as allowable vehicle constraints and relative temporal constraints. The search is over ordered partial assignments of waypoints to vehicles. In each state, there is a set of waypoints that have yet to be assigned called the *holding list* and there is a set of ordered waypoints assigned to each vehicle. The neighborhood of a state is given by five operators: *relocate* a waypoint by moving it from one vehicle to a specific location in the ordering for another vehicle, *exchange* two waypoints in the ordering on a single vehicle, *unassign* a waypoint by moving it from a vehicle's ordering to the holding list, *assign* a waypoint to a specific location in the ordering for a vehicle and *exchange* a waypoint on the holding list with an assigned waypoint on a vehicle.

The search begins from the initial state where all waypoints are unassigned. The neighborhood of the initial state is evaluated to find the best neighbor using an ordering predicate described below. As neighbors are generated, they are tested for validity in two ways. First, constraints imposed by the ordering of each schedule are tested for feasibility using a simple temporal network (STN, Cervoni, Cesta, and Oddi (1994)). In order to account for the distance between waypoints, we use the pre-computed shortest path distances to constrain each pair of waypoints to be separated by at least the time required to traverse the shortest path between them. If the STN reports that the ordering constraints are inconsistent with the constraints of the problem, then the neighbor is discarded as it cannot lead to a valid solution.

The second test is to see if the neighbor is *tabu* by checking if any of the waypoints that moved while generating the neighbor are included in a *tabu list*. The tabu list contains waypoints that are temporarily disallowed from being moved. If a neighbor fails the tabu test, then it is considered as a candidate for the best neighbor only if there are no other feasible neighbors. The tabu list helps to prevent the search from getting stuck in local minima by causing it to explore new portions of the space.

Once the best feasible neighbor is found, then the waypoints that were moved to generate that neighbor are added to the tabu list. If the size of the tabu list becomes greater than a fixed size (7 in our experiments), then entries are removed in first-in-first-out order. Finally, the search iterates with the best neighbor as the new current state. The best state ever encountered by the search is maintained as an incumbent, giving the sequencer an anytime behavior. The sequencer is stopped when either a maximum time limit has been reached or, if a full schedule has been found, it is stopped when no new incumbent arrives for a quarter of a second.

Following Lau et al. (2003), the ordering function used by the sequencer to estimate the quality of a state is hierarchical. First, the ordering function prefers states in which more waypoints have been scheduled. This helps encourage the sequencer to find total assignments of all of the waypoints to vehicles. In order to allow the user to make a trade-off between inexpensive and short schedules, we break ties using our version of the WAMP objective.

These costs approximate the actual makespan and cost of the final flyable route for the given schedule. Since the sequencer finds an ordering over the waypoints and not a fully instantiated timetable, there is some question as to how time may be allocated among the different legs of each route if there is flexibility in the temporal constraints. For estimating the cost of a state during the tabu search, we can use one of three techniques. The first approximation is optimistic and assumes that each leg will always use a path with the cost and duration of the minimum cost grid path. We call this the *min* estimation technique.

The *greedy* technique assigns time to each leg greedily. Each leg has an associated improvement slope which we use to estimate the rate at which we can convert extra time into cost reduction. The greedy technique greedily allocates more time to legs for which additional navigation time is likely to reduce cost the most. As we describe below, this greedy strategy is optimal in certain situations.

The final estimation technique is based on linear programming and is fairly expensive when evaluated on each state generated by the tabu search. We describe it in the next section as it is the same technique used to generate the timetable of the final solution returned from the sequencer.

3.3.3 Generating a Timetable

Once a waypoint ordering has been found for each vehicle, we generate a timetable that assigns the time when each waypoint should be achieved. This timetable will be used by the router to find a flyable path for each vehicle that achieves each waypoint at its designated time. In order to decide where time should be allocated along each vehicle's route, we again use an estimation of the time/cost trade-off for each leg of the route. The objective of this part of the solver is to assign each leg a time such that the sum of the associated costs is minimized.

In order to meet the problem's time constraints, more time may need to be spent on a leg than would be taken by the cheapest path. If this happens, the cost of the leg will generally be greater than the cheapest path cost due to cost incurred while waiting for time to pass. Currently, our implementation uses an optimistic approximation in which additional time can be added for free.

The LP uses two base variables for each leg: reduction duration $dur_{red(i)}$ and free duration $dur_{free(i)}$. $dur_{red(i)}$ represents the additional time that is devoted to avoiding high cost areas, and is required to be larger than the minimum travel time between the two waypoints, and smaller than the travel time of the cheapest path between the two waypoints. $dur_{free(i)}$ represents time beyond the time required for the cheapest path. $dur_{red(i)} + dur_{free(i)} = dur_i$, where dur_i is the duration spent getting to waypoint i from the previous place, either the previous waypoint or the starting location. t_i is the time at which waypoint i was achieved, and it is equal to the sum dur_j for all waypoints that the vehicle services up to and including waypoint i. The objective function of the LP is minimizing $\sum_{waypoints} dur_{red(i)} \cdot red_i + 0 \cdot dur_{free(i)}$ where red_i is the improvement slope. The zero coefficient reflects our optimistic assumption that time beyond the cheapest solution can be added for free. Temporal constraints from the problem all restrict t_i using linear inequalities so these can be entered into the LP directly, restricting the legal values of the derived variables.

An alternative method for solving the timetable problem is to use the greedy estimation method used by the scheduler.

Theorem 2. *In the case where there are no relative constraints in the problem or when all relative constraints are subsumed by the absolute constraints on each waypoint, the solution produced by the greedy algorithm is optimal.*

Proof Suppose we have a potentially optimal solution that is not the greedy solution. The fact that this solution is not greedy means there exists a pair of legs S and S' such that S' offers a worse return on investment of time, and S' was allocated time that could possibly have gone to S. This possibility implies that it is possible to shift time from S' to S by simply moving all the waypoints between S and S' by some nonzero amount, leaving the duration of all other legs the same. This solution cannot be optimal, because we can improve it by moving some time from S' to S. This reduces the cost of the solution because S' offers a worse return on investment of time than S, and all other legs remained the same duration. $\square$

Theorem 3. *In the general case, the problem requires a non-greedy method, such as linear programming.*

Proof We exhibit an instance with three vehicles (and some relative constraints) that defies greedy scheduling. Vehicle v_1 must visit waypoint w_1, which is at least 2 minutes away. Vehicles v_2 and v_3 each start one minute from w_2 and w_3, respectively, and must visit them exactly 1 minute before v_1 visits w_1. Anytime after v_1 visits w_1, v_2 must visit $w_{2'}$ and v_3 must visit $w_{3'}$. $w_{2'}$ is at least 1 minute from w_2 and $w_{3'}$ is at least 1 minute from w_3. All waypoints must be visited before time 7. The traversal costs are such that giving v_1 more time for w_1 lowers cost by 6 per minute, giving v_2 or v_3 more time for w_2 or w_3 doesn't lower cost at all, and giving v_2 or v_3 more time for $w_{2'}$ or $w_{3'}$ lowers cost by 5 per minute. The greedy scheduler will put w_1 at $7 - \epsilon$, w_2 and w_3 at $6 - \epsilon$, and $w_{2'}$ and $w_{3'}$ at 7. This lowers cost for v_1 by $(5 - \epsilon) \cdot -6$ and for v_2 and v_3 by $\epsilon \cdot -5$, for a total of $-30 - 4\epsilon$. The optimal solution puts w_1 at 2, w_2 and w_3 at 1, and $w_{2'}$ and $w_{3'}$ at 7, which lowers cost for v_1 by 0 and for v_2 and v_3 by $5 \cdot -5$, for a total of -50. $\square$

3.3.4 Routing

The router constructs flyable paths that meet the timetable while minimizing cost. The router performs this task one vehicle at a time, one leg at a time. Each invocation has three phases: finding a grid path, smoothing the grid path, and adding additional travel if necessary to match the timetable.

Finding a Path

The first step in constructing each leg is to use a discretized version of the problem to find an 8-way grid path that connects the cells containing the leg's start and goal. All grid cells whose center point is within the radius of the leg's goal waypoint count as goals. If the waypoint's radius does not contain a grid cell center, the grid cell that contains the waypoint's center point is used as the goal. Any cell touched by a keepout zone is marked as impassable in the grid search. Technically this approximation makes the planner incomplete, however, this was not an issue in practice.

Grid paths are found using A* search, modified to account for time constraints. The modified A* search prunes any state whose travel time so far t_{cur} and estimated remaining travel time t_{rem} (from the pre-computed shortest 8-way grid path times) are greater than the deadline d_i imposed by the timetable, $t_{cur} + t_{rem} > d_i$. The cost of each grid cell is determined in the pre-computation phase. The heuristic used during search is based on the pre-computed costs. The pre-computed cheapest path is used if its length is less than that required to meet the deadline.

Smoothing

If used directly, the 8-way grid path is usually dynamically infeasible and might not intersect the waypoint's radius or take into account heading constraints from the waypoint or the previous leg (or start position).

The grid path is smoothed by substituting arcs at sharp turns, resulting in a smooth path that is traversable by the current vehicle. This smooth path may not achieve the waypoint correctly (incorrect heading and/or incorrect position) and may not line up correctly with the exit trajectory from the previous leg. This is resolved by constructing dynamically feasible Dubins paths (Dubins, 1957; LaValle, 2006) to match up the ends of the smooth path with the previous leg and the goal waypoint. This is done by constructing a Dubins path that connects a point on the smooth path to either the goal waypoint or the previous leg.

The connection point choice has very visible impact on the resulting path. Choosing a point too close may result in large turns to correct heading discrepancies. Choosing a point too far away can remove too much of the cheaply routed path. We iteratively try several lengths, keeping the best path according to a weighted combination of cost and distance.

Constructing the connection from the smooth path to the goal waypoint has one more free variable, the heading at the waypoint. The waypoint may have an associated heading constraint so any values chosen must be within the specified range. The same iterative technique is used to evaluate connection points along the smooth path.

The heading at which a non-goal waypoint is achieved affects both the cost of the segment entering the waypoint as well as the cost of the segment exiting the waypoint. We would like to achieve the waypoint at a heading that is expected to have a cheap ingress as well as a cheap egress. To account for both of these costs, we consider a small set of pairs of Dubins curves where one curve in each pair is entering the waypoint and the other is exiting. The set is constructed using all combinations of a discrete set of starting points along the smoothed path entering the waypoint, ending points along the grid path exiting the waypoint, and headings at the waypoint. Of this set, we choose the curve that enters the waypoint from the pair that minimizes the weighted sum of cost and makespan.

Extending a Route

The smoothing process can result in paths longer or shorter than the grid path solution. When a path whose increased length results in missing the deadline, the A* search is continued to find a faster path. If no such path can be found, the router will fail back to the timetable stage with a new constraint bounding the problematic leg.

If smoothing results in a path that arrives at the waypoint before the deadline, the path is lengthened. If the time required to arrive at the deadline is at least the circumference of a tight loop of the vehicle, loops are added to the leg in the area where they will increase the cost least.

3.3.5 When Routing Fails

The timetable is generated using only an approximation of the routability between waypoints, so it may happen that the router is unable to meet the given deadlines. This can occur when the shortest 8-way grid path between two waypoints is shorter than the shortest flyable path. When the router fails to successfully route a leg, it passes the true minimum distance and cost of the failed leg back to the LP and sequencer. Using this new distance constraint, a new timetable is found and routing restarts. Additionally, if the updated LP has become infeasible, then the ordering produced by the sequencer is invalid and the sequencer is restarted to find a different ordering.

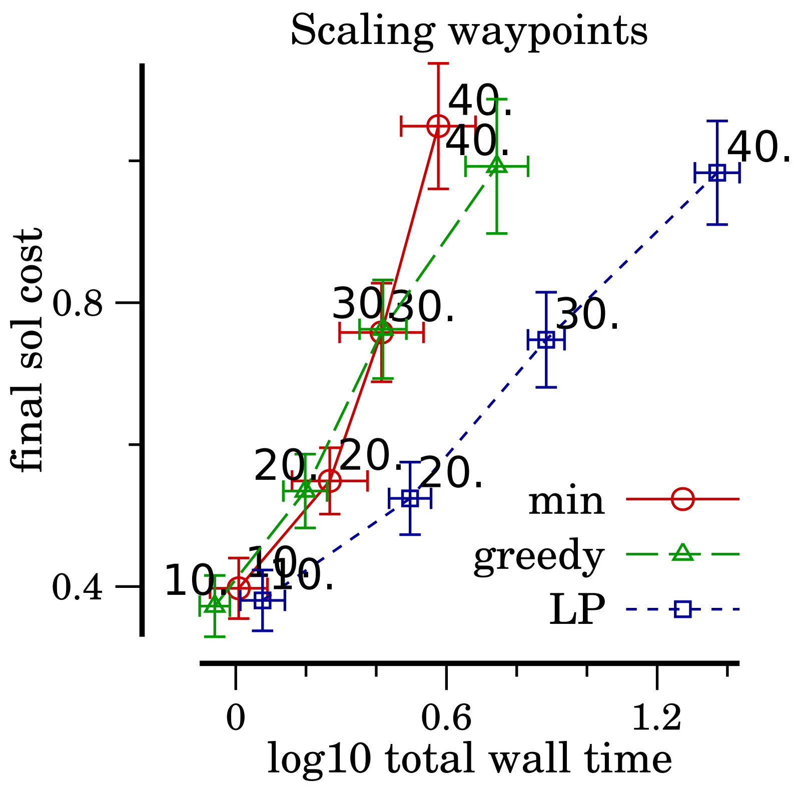
To avoid re-planning the same legs again in an updated timetable, the router caches the route for each successful leg. When a new timetable requires a leg that has already been routed with the same time constraint, this leg is re-used from the cache.

3.4 Evaluation

We now present the results from experiments we performed to evaluate the planner.

3.4.1 A Single Unified A* Search

Our first experiment verified that solving WAMP by running an A* search on the combined task and motion planning problem would quickly become infeasible. The state space included the airplane's position, heading, and time. The available operators were turn left or right 45° and go straight. For a heuristic, we calculated a minimum spanning tree of the 8-way grid path costs between waypoints on a discretized version of the problem, and added the distance of the vehicle to the nearest waypoint. The A* solver was written in Java, and we define failure as filling a 7GB object heap. Figure 3-3 (c) shows the failure percentages (right column) as the number of waypoints scales from 1 to 6 (left column). Each of these problems used a single vehicle and had no temporal constraints. The A* solver fared extremely poorly on this problem, and was unable to successfully solve a full set of these very small instances even with a single waypoint.



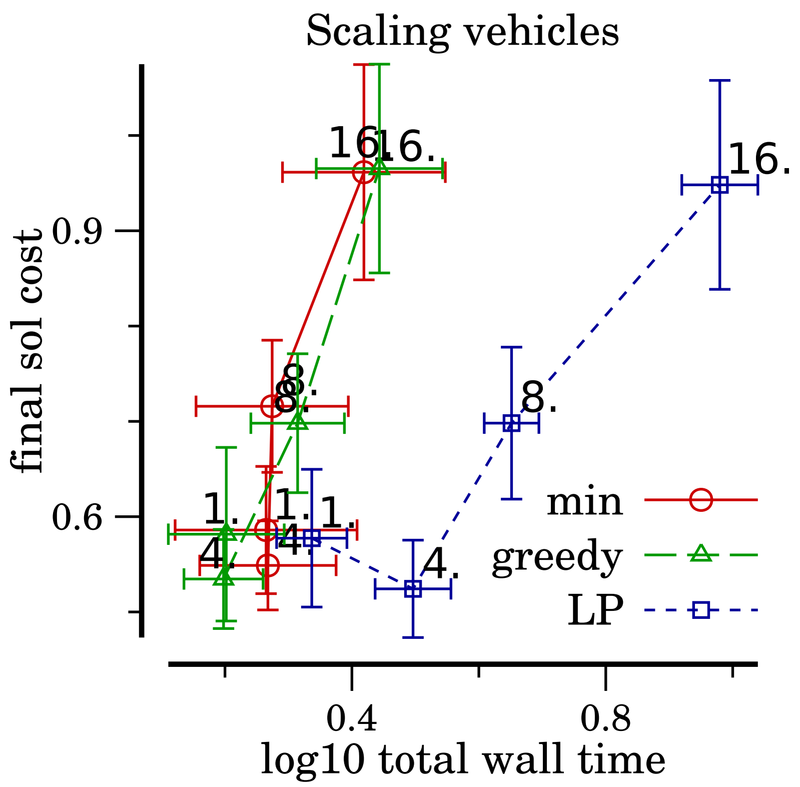


Figure 3-6: Scaling the number of waypoints and vehicles.

3.4.2 Scaling Behavior

We now turn to evaluating the approach discussed in this chapter, which was implemented in C++ and run on a 3.16 GHz machine with 8GB of RAM. Our first evaluation measured solution time and cost when scaling both the number of vehicles and the number of waypoints. Both of these parameters have a large effect on the difficulty of problem. The plots in Figure 3-6 show the results of these experiments. The top plot shows the scaling behavior of the min, greedy and LP surrogates as a function of the number of waypoints. Each glyph represents the mean time and cost over a set of instances with a number of waypoints given by the label (10, 20, 30 or 40) and 4 vehicles. The error bars give the 95% confidence interval on the means. A line connects each mean in order of increasing number of waypoints. As can be seen, the problems require more time and accrue more cost as the number of waypoints increases. When using both the min and greedy surrogates, we are able to solve the instances within our 10 second time frame even with up to 40 waypoints. We were surprised by the good performance of the min approximation. The LP approximation requires more time and is only able to solve up to 30 waypoint instances within the 10 second time frame.

The bottom plot of Figure 3-6 shows the scaling behavior as the number of vehicles increases. These instances had 20 waypoints. As the number of vehicles increases, the planning time increases. Again, the min and greedy surrogates give the best performance. Both the greedy and min techniques easily solve all problems within the 10 second time frame. The LP technique requires more than 10 seconds for some 16 vehicle instances.

3.4.3 Evaluating the Scheduler

Next, we considered synthetic instances that stress each major component of the system separately. To evaluate the sequencer, we created a set of instances for which we could find optimal solutions to the scheduling problem using the Concorde TSP solver (Applegate, Bixby, Chvatal, & Cook, 2006). We converted sets of TSP instances with 40 and 100 cities into WAMP instances for a vehicle with turning radius ϵ in order to compare the solutions found by our sequencer to the optimal TSP solutions. For comparison, we also implemented a simple nearest-neighbor TSP solver which chooses to visit the nearest unvisited city next.

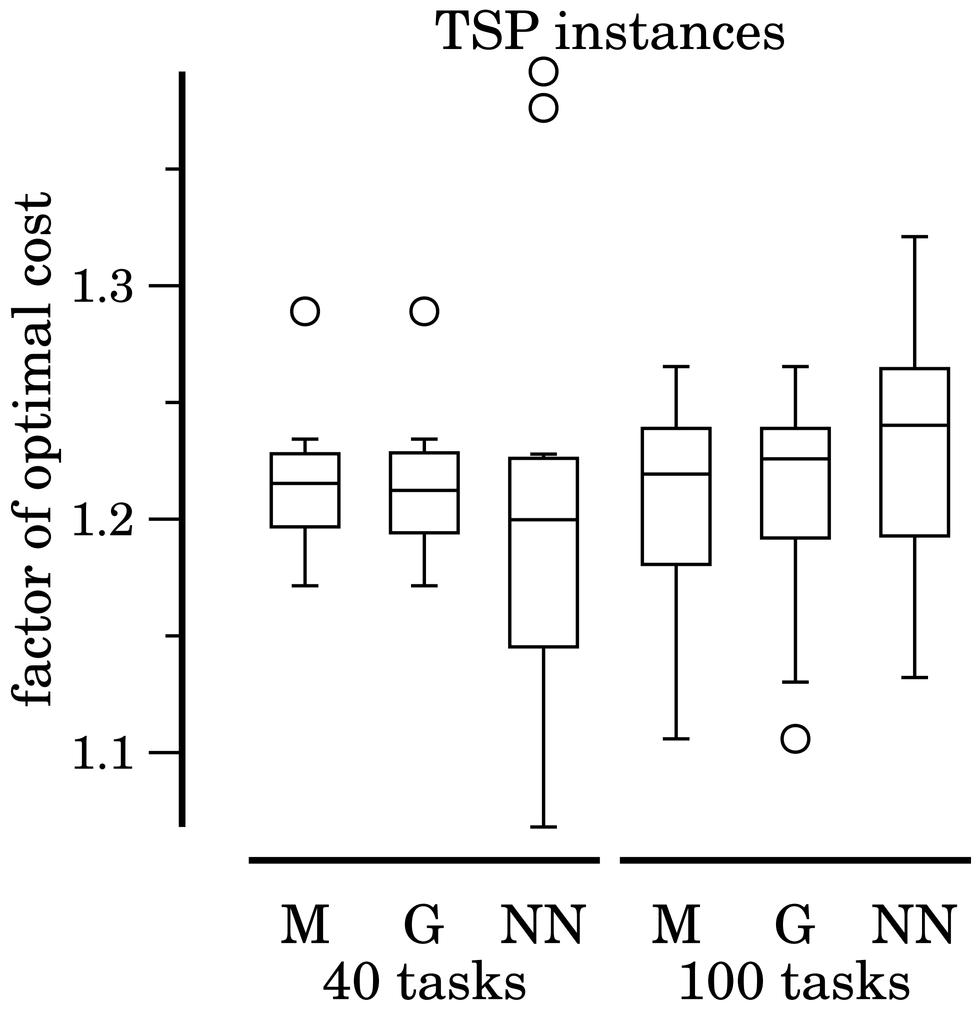


Figure 3-7: Optimality of TSP instances.

The results of this experiment are shown on the plot in Figure 3-7. We compared the min (M) and greedy (G) approximation techniques and the nearest neighbor TSP solver (NN). The y axis shows the factor of the optimal cost, so 1 is optimal and 1.2, for example, is 20% over optimal. Each box surrounds the middle half of the results, the horizontal line represents the median value, the 'whiskers' extend to the min and max. Circles beyond the whiskers show outliers. This plot does not include any results for the LP-based approximation as it was unable to solve any of the 100 city problems within a 120 second time limit. We can see from this plot that our ordering search tends to find solutions that are 20% above the optimal cost. For the more difficult 100 city instances, both the min and greedy approximations tend to outperform the nearest neighbor solver. Additionally, the 100 city instances seem to skew a bit more toward low-cost solutions than the easier 40 waypoint instances. We interpret these as positive results because they show that our sequencer is able to find reasonable solutions to these TSP instances.

3.4.4 Evaluating the Router

To evaluate the router, we created instances that required traversal of a maze of high-cost regions. Figure 3-4 (d) shows the path found for one such instance. While we did not have any simple way to quantify these results, it is visibly clear that the router was able to find its way through the mazes while avoiding high-cost regions.

3.4.5 Application

Finally, we evaluated on a set of instances that were similar to those used by our industrial partner. These instances were 200x200 miles, with 3 vehicles, and 41 waypoints. Our industrial partner's current system, which we do not have access to for reasons of intellectual property and security classification, solves instances like these in approximately 7 seconds. On this set of instances, our solver had a mean solution time of 2.5 seconds. We have designed our implementation such that we expect near linear time speedup on a multi-core machine; so these results could be improved even further. Due to confidentiality reasons we were unable to directly compare solutions on quality, however we generally received positive feedback.

3.5 Related Work

Lavalle (1998) introduced rapidly-exploring Random Trees (RRTs) which are a popular technique for finding dynamically feasible motion plans, however they do not minimize path cost. The RRT* algorithm (Karaman & Frazzoli, 2010) minimizes cost, but does not handle temporal constraints.

Bhatia, Kavraki, and Vardi (2010) combine sample-based motion planning with temporal goals by employing a geometry based multi-layered synergistic approach. Unlike the temporal constraints of WAMP, their goals are given by linear temporal logic formulas.

Dornhege, Gissler, Teschner, and Nebe (2009) describe how to combine low-level motion planning with high-level task planning via semantic attachment to a PDDL planner. In their approach, the lower level planner is used to check action applicability and compute effects whenever certain high level actions are used. In our approach, we use pre-computed minimum travel times to allow quick feasibility checking during high level planning, reserving the low level planner for computing the true cost of a solution.

Kaelbling and Lozano-Pérez (2011) present a more flexible technique for combining both task and motion planning called "hierarchical planning in the now." The technique generates a hierarchy dynamically. When refining a transition at one level in the hierarchy, a planner is used where the goal specification is given by the preconditions of the destination node of the transition. This technique does not handle temporal constraints or a cost metric other than makespan.

Frank, Stachniss, Abdo, and Burgard (2011) make use of surrogate objective for motion planning for a robotic arm in the face of deformable objects. Their technique uses a surrogate objective to avoid using a computationally intensive finite element methods simulation to compute the cost of object deformations.

There has also been previous work in routing for aerial vehicles. McVey, Clements, Massey, and Parkes (1999) present the Worldwide Aeronautical Route Planner (WARP) that plans fuel-efficient airplane routes around the globe. Štěpán Kopřiva, Šišlák, Pavlíček, and Pěchouček (2010) present Iterative Accelerated A* (IAA*) which is a technique developed for flight-path planning that increases the distance covered by each action primitive when the vehicle is far from surrounding obstacles. However, neither of these techniques consider temporal constraints.

3.6 Possible Extensions

Our current surrogate objective optimistically assumes that additional time can be added to a route free of charge. Better approximations for this issue in the future could be explored. One possible improvement is to estimate that additional time adds cost at a fixed rate. As long as the change in slope at that point is a positive change in slope, it can still be captured in a linear program.

An additional improvement to our current system would be to allow the router to pass more information back to the sequencer and linear programming layers. Currently, the router only sends accurate time/cost information back to these layers when it determines that a leg is unroutable. One may imagine a more complex system, however, where information flows back to the sequencer and linear programming layer for every successfully routed leg too.

3.7 Conclusion

We introduced the problem of Waypoint Allocation and Motion Planning, which requires integration of high-level task planning with low-level motion planning. WAMP models a real-world application, moving beyond the classic planning problems and raising interesting issues that have not received much attention, including how to partition effort in a multi-level planner and how to trade plan cost for execution time in a time-constrained context. WAMP contains many subproblems that are well-known to be NP-complete and we proved that the target value search problem is NP-complete. We described an approach for the WAMP problem that takes advantage of the characteristics of the problem in order to separate the solving into three distinct stages: scheduling, building timetables, and routing. Our approach makes use of a surrogate objective in the high-level layers in order to avoid calls to the more expensive low-level route planner. Using this hybrid approach we are able to meet the demanding requirements imposed by the application.

In this chapter, we used abstraction to solve a complicated task and motion planning problem, WAMP. As shown in the experiments, trying to solve the problem directly in the grounded space using an algorithm like A* was not scalable. By applying these non-classical planning techniques to estimate and identify promising portions of the state space to explore, we were able to get scalable performance. This work was published in a 2012 ICAPS paper (Kiesel et al., 2012).

In Chapter 4, we will continue to build on this idea of applying non-classical planning ideas using abstractions to the area of sampling-based motion planning for robots.

CHAPTER 4

f-Biased Sampling

4.1 Introduction

In this next chapter, we continue the trend of guiding low level search through high level reasoning. We shift our focus from task and motion planing, to simply the core motion planning problem.

We begin by recalling Dijkstra's algorithm (Dijkstra, 1959), the well-known search technique for finding paths in a discrete state space graph. Dijkstra's algorithm explores a graph by visiting its nodes in ascending order according to the cost necessary to reach them and it is guaranteed to find a cheapest path from an initial node to any node in the graph. Unfortunately, the search is unfocused and will explore portions of the graph that lead away from the goal as well as those that lead toward it. To alleviate this problem, the A* algorithm (Hart et al., 1968) uses a cost-to-go estimate called a heuristic. When a heuristic estimate is available, A* always visits fewer nodes than Dijkstra's algorithm, as it avoids portions of the graph that only participate in high cost solutions.

Rapidly-exploring random trees (RRTs) (Lavalle, 1998) are a popular technique for motion planning in continuous spaces. The RRT algorithm builds a tree of paths by sampling configurations. The point in the tree nearest to each new sample is steered toward the sample, creating a new path segment and a new node in the tree. RRTs are complete in the limit of infinite samples, however they do not optimize for low cost solutions. Karaman and Frazzoli's RRT* algorithm (Karaman & Frazzoli, 2011) re-wires the tree when lower cost paths can be found to existing nodes near each sample point. RRT* is both complete and asymptotically optimal. However, much like Dijkstra's algorithm for discrete graph search, RRT* will expend effort exploring portions of configuration space that lead exactly away from the goal as well as towards it.

The main contribution of this chapter is a new technique called f-biasing, named after the value f used by A* to order its search effort. Just as A* improves over Dijkstra's algorithm, f-biasing focuses exploration of RRT-based algorithms toward areas that are more likely to lead to the goal configuration, and to do so via low cost trajectories. To use f-biasing, we first solve a discretized and abstracted version of the motion planning problem. Then, using the cost estimates found in the abstracted problem, we bias the location of samples in the RRT so that they are more likely to be drawn from portions of configuration space that contain low cost solutions to the abstracted problem.

After discussing the method in detail, we prove that f-biasing maintains the completeness and convergence properties of RRT and RRT*. We then compare f-biased RRT and RRT* to their unbiased and goal-biased versions using three vehicles of increasing complexity: a simple straight-line vehicle, the Dubins car, and a hovercraft. f-biasing finds its first solutions more quickly in all domains except Dubins car with RRT*, where our current f-biasing implementation has more re-wiring overhead and thus is only as fast as the other methods. We also show anytime profiles that demonstrate that f-biasing both solves more problems and is able to improve its solution quality more quickly than other techniques. Finally, we show how f-biased RRT can provide a larger improvement over unbiased RRT than the RRT* algorithm. Broadly, we see this work as strengthening the connections between motion planning in robotics and combinatorial search in artificial intelligence that were pioneered by algorithms like RRT* and R* (Likhachev & Stentz, 2008).

4.2 Previous Work

We begin with a discussion of related work in both heuristic search and robotics.

4.2.1 Heuristic Search

A* (Hart et al., 1968) is an optimal search algorithm for discrete graphs (Dechter & Pearl, 1988). A* visits nodes in increasing order of estimated solution cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial node to node n and $h(n)$ is the heuristic value of n, estimating the cost from n to a goal node.

One technique for creating heuristics is by relaxing the constraints of the problem. Essentially, this technique adds extra edges between states that do not exist in the original problem. Likhachev and Ferguson (Likhachev & Ferguson, 2009) provide two examples of relaxation as applied to motion planning. The first example is their removal of obstacles from a motion planning problem to create a simpler relaxed problem that can be solved quickly. The second example is the ignoring of vehicle dynamics in order to relax motion constraints. Solutions to these relaxations are lower bounds on the cost-to-go in the original problem and are used to guide search.

Currently, some of the most powerful heuristics used by the search and AI planning communities are created using abstraction. An abstraction is a many-to-one mapping from the search space to a smaller abstract representation of the search space. For example, Remolina and Kuipers's (Remolina & Kuipers, 2004) topological maps are a form of abstraction created by mapping regions of space to single nodes in a map. Sturtevant and Geisberger (Sturtevant & Geisberger, 2010) also present an overview and a comparison of recent advances in the area of abstraction-based heuristics for grid pathfinding.

Pattern databases (PDBs) (Culberson & Schaeffer, 1998) are one of the most popular abstraction-based methods and are closest in spirit to f-biasing. A PDB contains the cost-to-goal for every state in an abstract representation of the search space, computed by performing Dijkstra's single-source shortest path algorithm in reverse from the abstract representation of the goal to every node in the abstract state space. During search, the abstract costs from the PDB are used as admissible heuristic estimates for search states: when a non-trivial heuristic estimate is needed for a node, the solution cost for the abstract representation of the node is used as the estimate.

4.2.2 Rapidly-exploring Random Trees

Rapidly-exploring random trees (RRTs) (Lavalle, 1998) grow a tree from the initial configuration toward random samples in configuration space. Each iteration of the RRT algorithm samples a random configuration, finds the node in the tree that is nearest to the sample, and then adds a new node to the tree by steering the nearest node toward the sample. In the limit of infinite samples, an RRT will densely cover the configuration space.

The RRT* algorithm (Karaman & Frazzoli, 2011) is a simple modification to the standard RRT algorithm that allows it to find cheaper motion plans. Whenever a new node is added to the tree, nearby nodes are updated if they can be reached by a cheaper path via the new node. The re-wiring performed by RRT* is closely analogous to a common technique used in heuristic search algorithms, such as A*, in which, whenever a cheaper path with a lower g value is found to a node, the cheaper path is used and the more expensive path is discarded. This can be seen as a form of dynamic programming. Unlike A*, however, RRT* makes no use of a heuristic estimator.

Other variants of the basic RRT algorithm have been proposed, such as bidirectional RRT (Kuffner & LaValle, 2000). In this chapter, we only evaluate f-biasing on the basic RRT algorithm and RRT*, however any sampling technique, such as f-biasing, could easily be applied to bidirectional RRTs.

RRT Sampling Schemes

Previous authors have also recognized that uniform exploration is not the most efficient choice for a single query motion planning algorithm. There are a variety of previous proposals for biasing sample selection in an attempt to decrease the time required to find the first solution, improve the handling of navigation near obstacles, and increase the exploration of the configuration space. Most of the techniques summarized here are discussed in greater detail by LaValle (LaValle, 2006).

Unbiased Random Sampling: Unbiased random sampling, the method that was originally proposed for generating an RRT, has the benefit of covering the configuration space without prejudice and is appropriate for domains where no prior knowledge or only inaccurate knowledge is available. The following biasing techniques attempt to exploit additional information to find better solutions faster.

Goal-biased Sampling: Goal-biased sampling (Lavalle & Kuffner, 2000) selects the goal configuration, or configurations near the goal, more often than uniform sampling in an attempt to grow the RRT more quickly toward the goal. There are two major flavors of goal biasing. First, the goal configuration itself can be selected as a sample with some fixed probability p, otherwise an unbiased sample is used. The second version of goal biasing selects configurations near the goal instead of only the goal itself. One common method for this is to use a Gaussian distribution (Lavalle & Kuffner, 2000; Song & Amato, 2001) around the goal configuration. These both can overcome minor local obstacles, however, if a the start configuration lies in a heavily obstructed part of the space far from the goal, it will be difficult for the planner to construct the the tree through the obstructions.

Heuristic-biased Sampling: Urmson and Simmons (Urmson & Simmons, 2003) introduced heuristic-biased sampling, which biases samples to be nearer to those nodes that the RRT reached via lower cost paths. This method has been shown to find cheaper motion plans, however, its biasing is based on the cost of paths found by the RRT regardless of whether or not these paths lead toward the goal. Like Dijkstra's algorithm, heuristic-biased sampling will explore portions of the space that lead away from the goal if it has reached them via cheaper paths than those leading toward the goal. Instead, we would like to sample from areas that we expect to be traversed by cheap trajectories that actually reach the goal.

Path-biased Sampling: The previous method that is most similar to ours is path-biased sampling (Vonásek, Faigl, Krajník, & Přeučil, 2009; Krammer, Granzer, & Kastner, 2011). While it was developed independently, path-biasing is similar because it can be seen as using the solution to an abstract or simplified representation of the motion planning problem such as a discrete grid or visibility graph (Nilsson, 1969). An RRT is then constructed by choosing samples along the solution path of the abstract problem with a probability p and uniformly otherwise. Using this technique, samples tend to occur along a possible low cost trajectory from the initial configuration to the goal.

Basic path-biasing fails if the path found in the simple problem doesn't take into account constraints of the complex motion planning problem. To hedge against this possibility, Krammer et al. (Krammer et al., 2011) propose a modified variant that draws samples from a Gaussian distribution around the abstract solution path. As we discuss next, f-biasing uses a more principled approach by selecting samples from areas of the configuration space with a probability based on the solution cost in the abstract space. Effectively, f-biasing takes into account all low-cost paths in the abstract space simultaneously instead of focusing on a single path. Furthermore, we will demonstrate that this is effective even for vehicles with complex dynamics, such as a hovercraft.

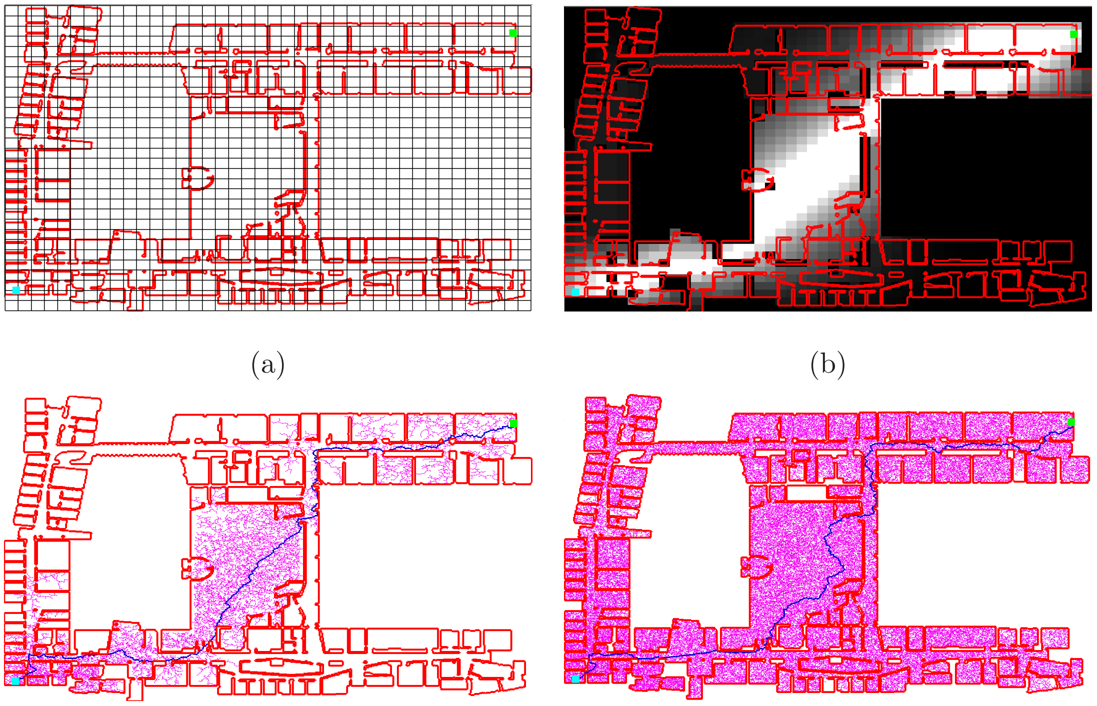
4.3 f-biased Sampling

We have discussed heuristic search and the benefits that it gains by using a heuristic to focus its effort on areas of a search space that harbor low cost solutions. Next, we saw that many of the most powerful state-of-the-art heuristics are created by using abstraction, and lastly, we described RRTs, which use sparse, uniform random sampling to explore the continuous and high-dimensional nature of motion planning problems. f-biased sampling combines these three ideas: heuristic search, abstraction, and sample-based motion planning. The first step in using an f-biased RRT is to create an abstract representation of the motion planning domain. Next, Dijkstra's algorithm is used to pre-compute the cost of the shortest path through each abstract node from the initial configuration to the goal in the abstract space, as in PDBs. Like a heuristic, these abstract solution costs give the ability to focus the RRT's growth toward configurations that map to abstract states with low costs. We now explain each of these steps in greater detail.

4.3.1 Abstraction

The abstraction is represented by a weighted directed graph that is small enough to be searched exhaustively with Dijkstra's algorithm. There are many possible techniques for generating an abstract representation of a problem. In our implementation, we use a simple uniform discretization of configuration space to create an n-dimensional grid, where n is less than or equal to the dimensionality of the configuration space. Each vertex in the abstract graph is a discrete configuration that represents all configurations in the continuous space that fall within its Voronoi hyper-rectangle. Adjacent vertices in the abstract graph are connected via an edge if neither vertex is obstructed by an obstacle. In our implementation, a vertex is obstructed if its discrete configuration is contained within an obstacle. The weight of each edge reflects an estimate of the cost of the navigating between the two discrete configurations that it connects.

Figure 4-1(a) shows a polygonal map of the second floor of Kingsbury Hall at the University


(a) (b)

(c) (d)

Figure 4-1: An example map showing abstraction (a), f values (b), an f-biased RRT (c) and regular RRT (d).

of New Hampshire along with a possible abstraction, represented as a coarse grid overlaid on the continuous domain. Each cell of the coarse grid is a vertex in the abstract graph and the graph has eight-connectivity. This is an extreme simplification, but as our experiments will show, it suffices to guide motion planning.

For each motion planning query, we map the initial and goal configurations to their abstract nodes in the abstract representation of the state space. We then compute the cost of the path from the initial node through each abstract node to the goal node. Because we use a discrete abstraction, this can be done in linear time in the size of the abstract space by using two calls to Dijkstra's single-source shortest path algorithm: one that computes the shortest path from the initial node to each node, $g(n)$ in A* terminology, and another that computes the shortest path from each node to the goal node, $h(n)$. The sum of these values gives the cheapest cost of a solution path passing through the given node, $f(n) = g(n) + h(n)$.

Figure 4-1(b) shows the f values for a motion planning problem using the abstraction from Fig. 4-1(a). The initial configuration is shown as a light blue square in the lower-left corner and the goal is shown as a green square in the upper-right corner. Each abstract node is shaded, with black representing high f cost. As we can see, even this simple abstraction suffices to uncover that it would be more desirable to focus RRT growth into the lighter areas of the map while spending less time considering the dark portions.

4.3.2 Growing an f-biased Tree

To create an f-biased RRT, we proceed as in the standard RRT or RRT* algorithm, however, more samples are taken from configurations that correspond to low cost abstract nodes. To accomplish this, each node is assigned a score so that nodes with low f values have high scores and nodes with high f values have low scores. These scores are then normalized to sum to 1 and the normalized scores give the probability with which an abstract node is selected for sampling. Once an abstract node is selected, a sample from the concrete configuration space is drawn uniformly from its preimage—the set of concrete configurations that map to the selected node in the abstract space.

The score for each abstract node is given by $s = f_{\min}^{\omega}/f^{\omega}$, where $f_{\min}$ is the minimum f value of all abstract nodes and ω is a configurable parameter representing the strength of the f-bias. Increasing ω increases the influence of the abstract nodes that are closer to $f_{\min}$, narrowing the corridor from which most of the samples are drawn. Decreasing ω decreases the influence of the abstract nodes that are closer to $f_{\min}$ and increases the amount of exploration. In our experiments, we used $\omega = 4$ as it was found to give good performance in a small set of preliminary experiments.

In some cases, an abstract node has an f value of infinity, for example, if it resides within an obstacle. We would still like to generate samples from these areas to maintain guaranteed completeness of the planner. When f is infinite, we define the score to be $s_{\min}/2$ where $s_{\min}$ is the minimum score of all nodes with finite f. The nth abstract node is selected for sampling with probability $p_n = s_n / \sum_{i=0}^{N-1} s_i$. Finally, a sample is generated uniformly from among all possible configurations in the preimage of the selected node. Figure 4-1(c) shows a completed f-biased RRT, along with its solution path. For comparison, Fig. 4-1(d) shows the first solution found by an unbiased RRT. Notice that, in the f-biased RRT, most of the exploration is focused in the lighter cells that reside along the diagonal between the initial configuration and the goal. The unbiased RRT required many more samples and explored the entire map.

4.4 Theoretical Analysis

Previous results on RRT and RRT* are robust enough to survive the bias introduced by our technique.

Lemma 4. *Under f-biasing, there exists a positive constant that bounds from below the probability of selecting each configuration.*

Proof: Under f-biasing, every abstract node has a positive probability of being selected to sample within. This lower bound is defined as $s_{\min}/2$ where $s_{\min}$ is the minimum score of all nodes with finite f. The probability of the selecting the lowest probability abstract node is defined as $p_n = s_n / \sum_{i=0}^{N-1} s_i$. Therefore every configuration in the preimage of an abstract node thus has positive probability of being sampled by construction. $\square$

Theorem 5. *Using f-biased sampling does not disrupt the probabilistic completeness of the RRT or RRT* algorithms.*

Proof: Lemma 4 is exactly the condition required by the completeness proof for RRT given by LaValle and Kuffner (LaValle & Kuffner, 2001), thus completeness is maintained. The completeness proof of Karaman and Frazzoli for RRT* (Karaman & Frazzoli, 2011) is inherited directly from RRT, thus RRT*'s completeness is also preserved. □

Theorem 6. *Using f-biased sampling does not disrupt the asymptotic optimality of the RRT* algorithm.*

Proof: The proof of RRT*'s asymptotic optimality (Karaman & Frazzoli, 2011) relies on the rewiring step to monotonically decrease path costs, which requires positive probability of adding any configuration to the vertex set. This property is ensured by Lemma 4. Said another way, RRT* merely rewires the same vertex set as constructed by RRT. Using f-biasing preserves non-zero probability of generating every possible RRT vertex set, hence it preserves asymptotic optimality. □

4.5 Experimental Results

Next, we evaluate the performance of f-biased RRTs experimentally on three different path planning domains.

4.5.1 Implementation Details

We attempted to obtain a copy of the RRT* implementation by Karaman and Frazzoli (Karaman & Frazzoli, 2011) for comparison, however, the source code was not available at the time of our request. Instead, we wrote our own implementation of RRT and RRT* in C++ using the same K-D tree implementation that was used by Karaman and Frazzoli (available from `http://code.google.com/p/kdtree/`). Our RRT* implementation also used their technique for reducing the size of the ball from which nodes are considered for re-wiring as more samples are generated. All techniques in our comparison used the same implementation and data structures; the only difference between the techniques was the decision of where in the configuration space the samples were generated. All experiments were performed on a 3.16 GHz Core2 duo PC with 8GB RAM running Linux.

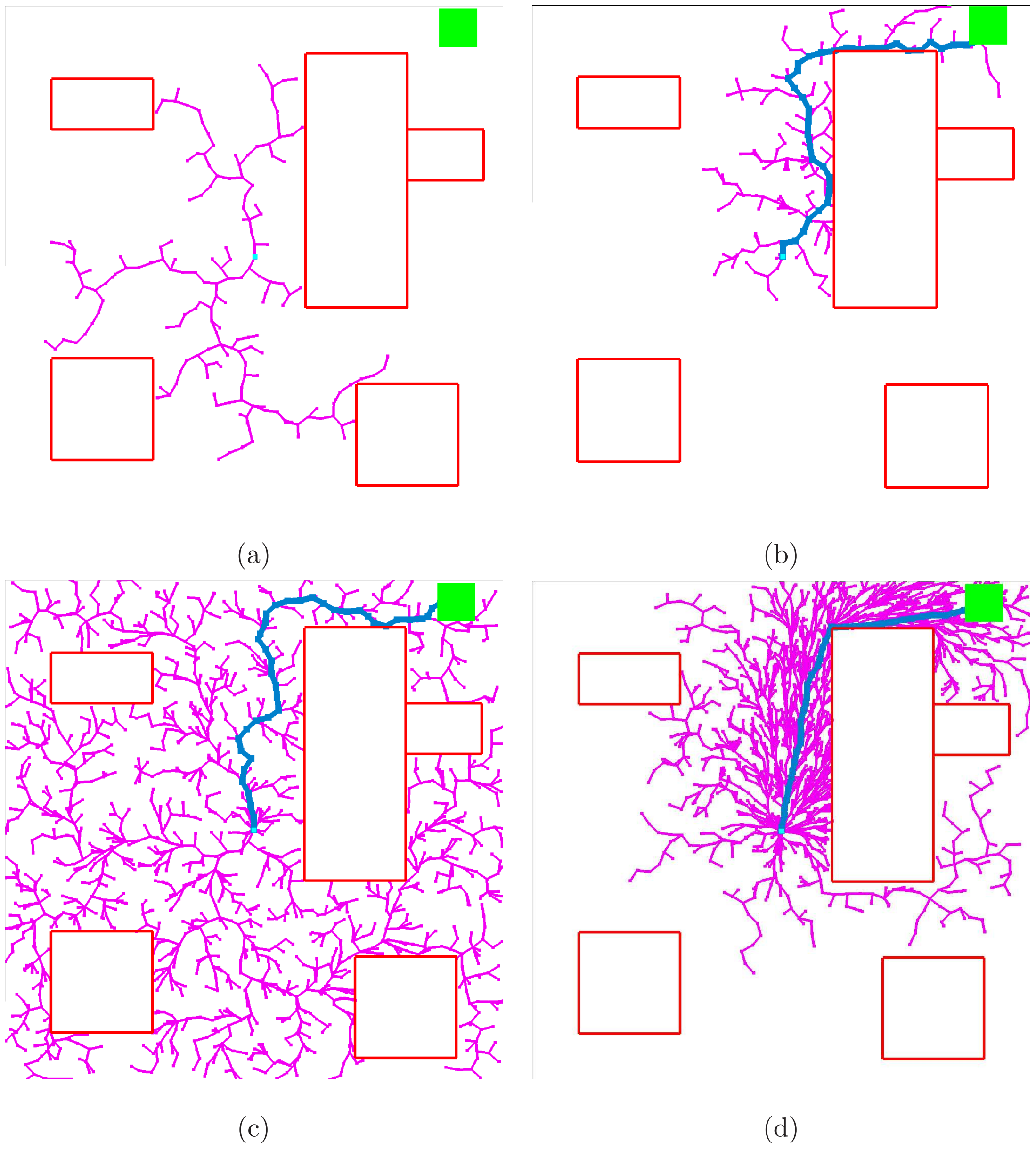
For *f*-biasing, the abstract node from which to sample was selected in constant time by inserting a reference to each abstract node multiple times into a large array to approximate its probability relative to the least probable node. An index into this array was then chosen uniformly at random. An alternative, less memory hungry, approach is to use binary search to select the node. To reduce the time spent by binary search, clustering can be used to group the abstract nodes into a small fixed number of equiprobable bins that can be searched very quickly.

In timing results with *f*-biasing, we do not include the time required to build the abstraction since it can be computed once for a given map and stored. Typically, the time required to build the abstraction was only a few seconds, most of which was spent testing if abstract nodes are blocked by obstacles in the configuration space. These tests can easily be performed in parallel, allowing the abstraction creation time to be greatly decreased with modern multicore hardware. Our timing results do include the time required to perform the Dijkstra shortest-path pre-computation step for each instance, because this must be performed for each individual motion query. Our implementation runs both Dijkstra searches in parallel as they are completely independent of one another. Regardless, this time was found to be quite insignificant.

4.5.2 Straight-line Vehicle

Our first set of experiments uses a very simple vehicle motion model from Karaman and Frazzoli (Karaman & Frazzoli, 2011) that we call the "straight-line vehicle." The straight-line vehicle moves straight and can instantly turn to any angle. The objective to minimize is the path length.

We begin by comparing unbiased RRT* with *f*-biased RRT* on a reproduction of the map used in Karaman and Frazzoli's Fig. 1 (Karaman & Frazzoli, 2011). They used this simple map to show the benefits of RRT* over basic RRTs. Likewise, Fig. 4-2 uses this map to show the benefit of *f*-biasing. Figures 4-2(a) and 4-2(b) show unbiased RRT* and *f*-biased RRT* respectively after 235 samples, when *f*-biasing finds its first solution. Figures 4-2(c) and 4-2(d) show the state of both algorithms after 2000 samples. We can see that combining the sampling of RRTs with guidance



(a) (b) (c) (d)

Figure 4-2: The example of Karaman and Frazzoli after 235 iterations with unbiased RRT* (a) and f-biased RRT* (b) and after 2000 iterations (c) and (d).

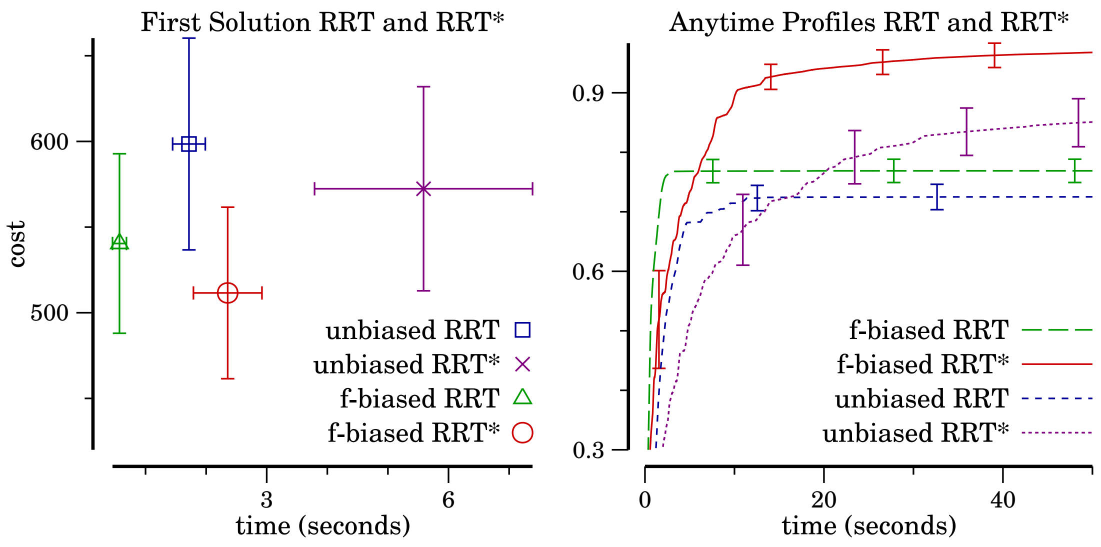


Figure 4-3: Straight-line vehicle: f-biasing and RRT* improvement over unbiased RRT.

from heuristic search caused f-biasing to find its first solution more quickly and enabled it to decrease the solution cost more quickly too.

The map in Fig. 4-2 is very simple, so our next results are on a set of 100 path planning problems given by uniformly selected initial and goal locations on the more realistic map of Fig. 4-1. The abstraction used for f-biasing was a uniform eight-connected grid of resolution 12x10.

First, we look at the improvement of f-biasing over standard RRT compared to the improvement of RRT* over RRT. The left plot in Fig. 4-3 shows the first solution time and cost for RRT and RRT* with and without f-biasing. The x axis shows the first solution time in seconds and the y axis shows the first solution cost. Each glyph represents the mean over the 99 instances that were solved by all techniques with RRT and the 100 instances solved by all techniques with RRT* within a 90 second time limit. Error bars show the 95% confidence intervals on the mean. We can see from this plot that f-biased RRT actually found its first solution significantly more quickly than all alternatives and in addition, its first solution costs tended to be slightly cheaper than that of unbiased RRT*. f-biased RRT* gave the best solution cost and took only slightly longer than unbiased RRT.

RRT and RRT* are naturally anytime algorithms; they provide a stream of solutions of decreasing cost as they are given more time. One common way to compare anytime algorithms is by comparing their *anytime profile*, i.e., solution cost over time. We ran each biasing technique twice with the same random seed for 90 seconds with RRT and RRT* on each of our 100 instances. The first run computed the solution cost achieved at each sample. Because there are many iterations, this cost computation required a non-negligible amount of CPU time, so the second run measured the time at which each sample was taken without the cost computation. This data was used to build anytime profiles.

The right plot in Fig. 4-3 shows the anytime profiles for RRT and RRT* with and without f-biasing. The data points were computed in a paired manner by finding the best solution found on each instance by the algorithms in the given plot and dividing this by the incumbent cost at each time value on the same instance. By initializing incumbent scores to infinity, this technique allows for comparison at times before all instances are solved. The lines show the mean over the instance set and the error bars show the 95% confidence interval on the mean. The plot shows that f-biased RRT and RRT* both find cheaper solutions faster than their unbiased counterparts.

Next, we compare f-biasing to both goal-biased and unbiased RRT and RRT*. The top row of Fig. 4-4 shows the time and cost of the first solution for f-biasing, goal-biasing with 1%, 10% and 25% of the samples being the goal configuration and unbiased RRT and RRT*. f-biasing found its first solutions significantly more quickly than the other techniques and the cost of its first solutions tended to be lower. The bottom row of Fig. 4-4 shows anytime profiles. From the left plot, we can see that when used in the RRT algorithm, f-biasing dominated the other techniques. In the right plot, we can see the same behavior for RRT* except that more time was required to approach the best cost solution. This is likely because of RRT*'s convergence to optimality: the best solution found by RRT* was much cheaper than the best found by RRT and more time was used to find it. Also, each iteration requires re-wiring.

4.5.3 Dubins Car

In this section, we evaluate the performance of f-biasing with the Dubins car (Dubins, 1957), which has an x and y location and heading θ that is constrained by a fixed turning radius. The abstraction used by f-biasing on this domain used a uniform grid of discrete x, y and θ combinations with

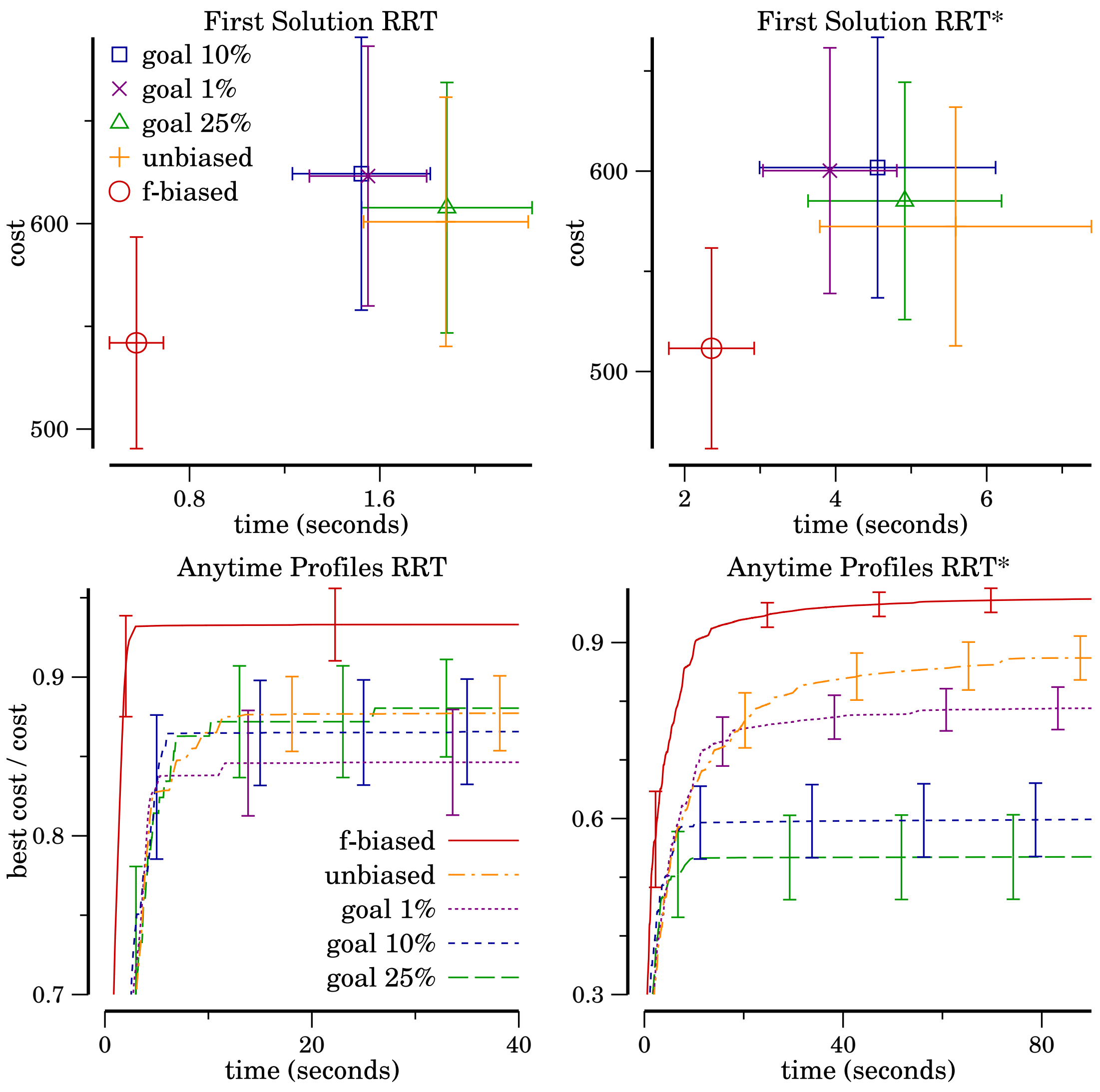


Figure 4-4: Straight-line vehicle: first solution times and anytime profiles for RRT (left) and RRT* (right).

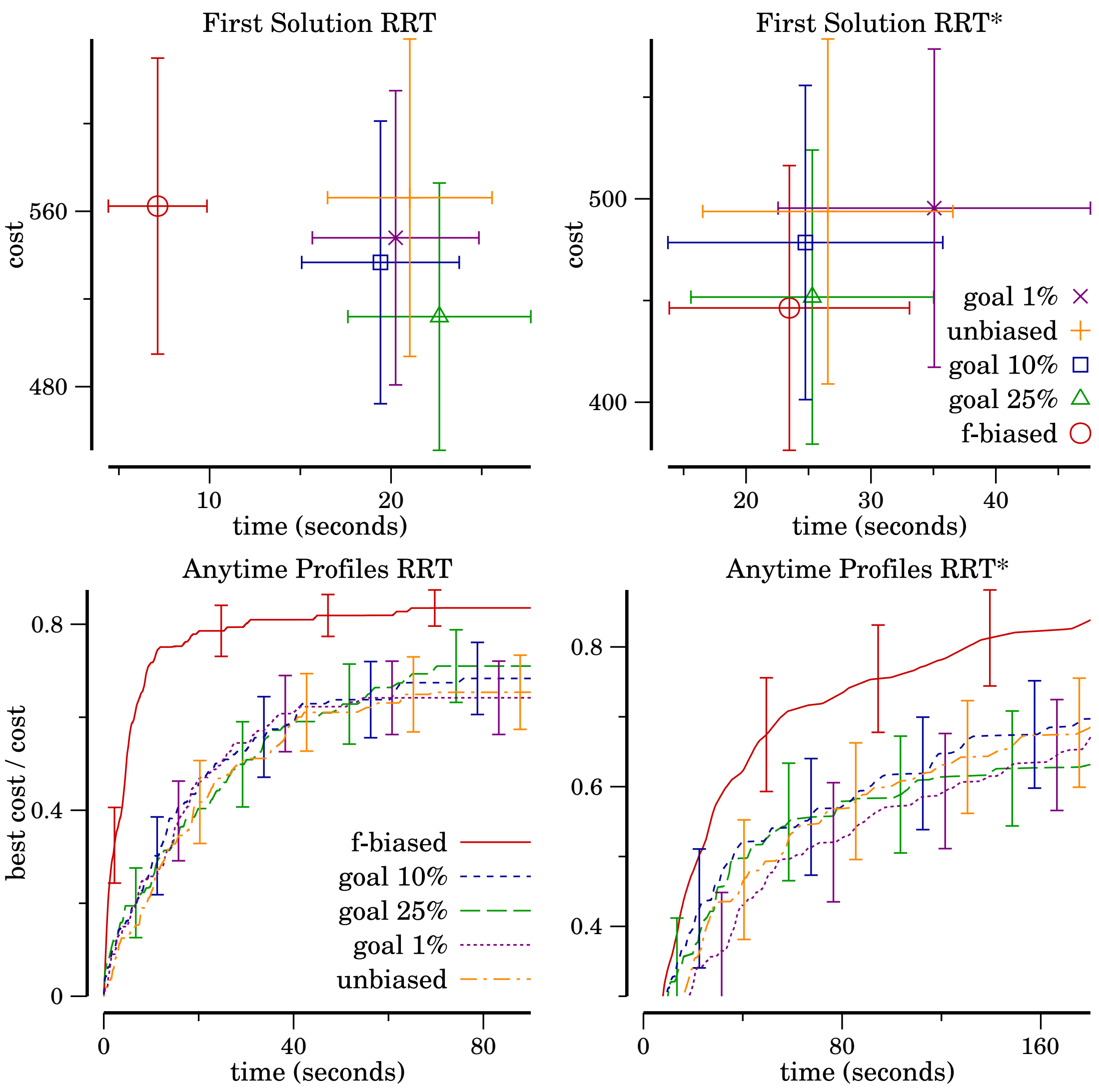


Figure 4-5: Dubins car first solution times and anytime profiles for RRT (left) and RRT* (right).

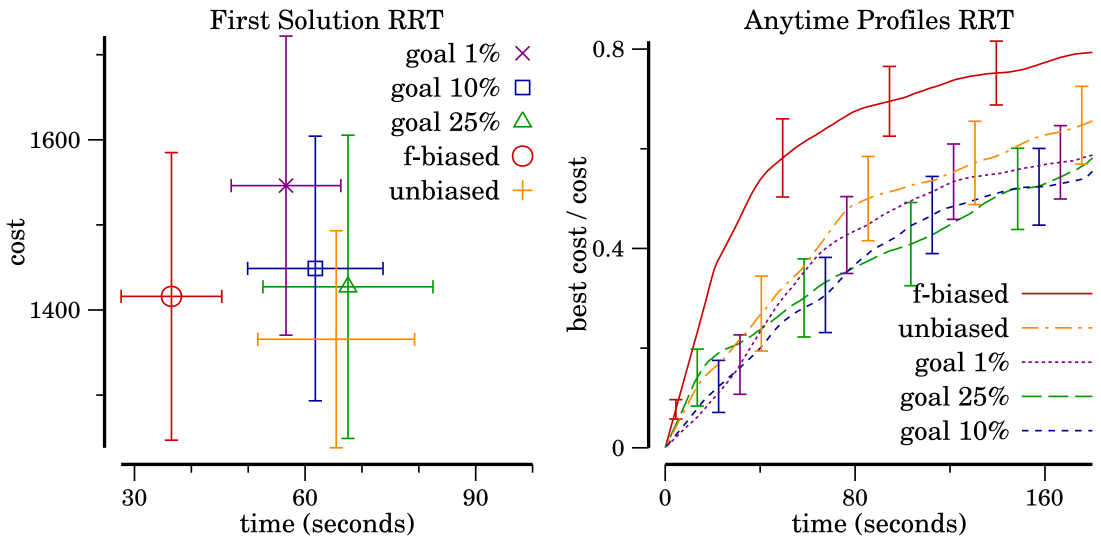


Figure 4-6: Hovercraft first solution times and anytime profile for RRT.

dimensions 75x65x4. In this set of experiments, we used the same instances that were used for the straight-line vehicle with a time limit of 90 seconds for RRT and 180 seconds for RRT*.

The top two plots in Fig. 4-5 show the time and cost of first solutions. For RRT, f-biasing found its first solutions significantly more quickly than the other techniques. For RRT*, however, none of the techniques found their first solution significantly faster than the others. f-biasing did not find its first solution faster in this setting because its biased samples created a very dense tree and so RRT* performed a lot more expensive re-wiring. The bottom two plots show anytime profiles. f-biasing had a better profile than all other techniques on both algorithms even though it performed fewer samples within the time limit for RRT*. This is because f-biasing both solved more instances and was able to find cheaper solutions with fewer samples than the other methods.

4.5.4 Hovercraft

The final domain that we present is path planning for a simple hovercraft. Each configuration consists of $\langle x, y, \theta, \delta x, \delta y, \delta\theta \rangle$. x, y and θ represent the craft's position and orientation. δx and δy represent the current translational rate in each respective direction and $\delta\theta$ represents the rotational velocity. This models a simple hovercraft with two fans: one propels the craft in the direction θ and the other applies rotational force in either direction. This domain has the largest dimensionality and presents the most difficult motion model of all domains considered in this chapter.

For the experiments in this domain, we used 100 random start and goal configurations on the map from LaValle and Kuffner (LaValle & Kuffner, 2001). The abstraction used for f-biasing was the same as used for the Dubins car with dimensions 26x26x4. f-biased RRT solved 90% of all instances within a 180 second time limit whereas goal-biased RRT with its best setting (1%) only solved 74% of the instances and unbiased RRT only solved 75%. The left plot in Fig. 4-6 shows the first solution costs and times for the 43 instances solved by all algorithms within the time limit. The first solution costs from f-biasing were not significantly different from that of the other techniques, however, it found these solutions significantly faster. The right plot shows the anytime profile, where we can see that f-biasing gave the best performance. Achieving good performance with such a basic abstraction for this complex domain suggests that f-biasing is robust to the choice of abstraction.

4.6 Discussion

As we point out in Section 4.5.3, f-biased RRT* is not able to generate samples as quickly as unbiased and goal-biased RRT* because it builds a denser tree and therefore requires more re-wiring at every sample. The sample speed of f-biasing can be increased in a couple of ways. First, Karaman and Frazzoli's (Karaman & Frazzoli, 2011) k-nearest technique can be used to fix the number of nodes tested for re-wiring at k, instead of checking all nodes within the ball. A second possibility is to choose the ball size used to test for re-wiring dynamically based on the sample density of the selected abstract node and its neighbors. Even without these optimizations, our results show that f-biasing performs favorably as it is able to find cheap solutions with fewer samples than alternative methods.

While the results presented in Fig. 4-6 show that f-biasing can give good performance even with a simplistic abstraction, it is worth noting that the choice of abstraction can be important. If the abstraction is too coarse, then it may not account for important obstacles in the planning problem. If this occurs, then the sampling can be biased toward regions of space that contain only infeasible plans due to the unaccounted obstacles. Given this, one might assume that a finer discretization of the abstract space will always perform better, as it is more informative, however, we have found that coarser discretizations actually tended to perform better in our experiments.

We have shown that f-biasing works well for constructing RRTs. We are also interested in trying to combine these ideas with other types of motion planning techniques. Probabilistic roadmaps (PRMs) (Kavraki, Švestka, Latombe, & Overmars, 1996) are a popular alternative to RRTs that work by constructing a roadmap of feasible paths between points that are sampled randomly from the configuration space. Once the roadmap has been constructed, motion planning queries can be performed by connecting the initial and goal configurations to any points on the roadmap and performing a fast discrete graph search.

As with RRTs, it is possible to bias the selection of locations used to create a PRM. One possibility for using the ideas presented in this chapter in conjunction with PRM construction would be to compute the *betweenness centrality* (Brandes, 2001) of nodes in an abstract graph. Betweenness centrality is a measure of the number of shortest paths upon which a node in a graph resides. Sampling from locations in the abstract graph with higher betweenness centrality may lead to more effective PRMs as the nodes in the roadmap may reside in areas of the space that are used in many shortest paths.

4.7 Conclusion

We have presented f-biasing for RRTs, a new technique that combines guidance from heuristic search with sparse sampling techniques from robotics. f-biasing effectively focuses the growth of an RRT on areas of configuration space that are traversed by low-cost paths in an abstract representation of the problem. This allows f-biased RRTs to find cheaper motion plans more quickly than other sampling techniques. Our experimental results demonstrate that this new technique outperforms unbiased and goal-biased RRT and RRT* on three different vehicle motion models: a straight-line vehicle, a Dubins car, and a hovercraft. This work strengthens the connections between motion planning in the robotics community and heuristic search in artificial intelligence.

In this chapter we employed abstraction ideas as well as traditional search node ranking using g, h, and f values to identify portions of the state space likely to contain solutions. This is a similar use of non-classical planning ideas as the previous chapter where we used a layer of high level reasoning to guide the low level construction of plans. By applying this sampling bias, we were able to obtain performance better than some of the standard sampling biases in robotic motion planning. This work was published in a 2012 SoCS extended abstract (Kiesel et al., 2012b) and a University of New Hampshire technical report (Kiesel et al., 2012a).

In Chapter 5, we further build on these ideas of abstraction to guide sampling based motion planning. We will borrow more ideas from the heuristic search community to modify the underlying motion planning algorithm, not just the sampling bias, to optimize the metric: time to first solution.

CHAPTER 5

Hybrid Motion Planning

5.1 Introduction

In this chapter we build on the success we saw in the previous chapter guiding low level motion planning using a high level abstraction. We continue to address the problem of single-query kinodynamic motion planning, a very common problem in robotics. Given a start state, description of the obstacles in the workspace, and a goal region, find a dynamically feasible continuous trajectory (a sequence of piece-wise constant controls) that takes the robot from the start state to the goal region without intersecting obstacles (Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki, & Thrun, 2005; LaValle, 2006). As in the previous chapter, we work within the framework of motion trees, popularized by sampling-based motion planning, in which the planner grows a tree of feasible motions from the start state, attempting to reach the goal state. This approach is appealing because it applies to any vehicle that can be forward simulated, allowing the planner to respect realistic constraints such as acceleration limits. Examples of algorithms taking this approach include RRT (LaValle & Kuffner, 2001), KPIECE (Sucan & Kavraki, 2009), and P-PRM (Le & Plaku, 2014).

Although the figure of merit on which these algorithms are usually compared is the time taken to find a (complete and feasible) solution, close examination of these algorithms reveals that their search strategies are not explicitly designed to optimize that measure. RRT uses sampling with a Voronoi bias to encourage rapidly covering the entire state space. KPIECE uses more sophisticated coverage estimates to achieve the same end. It is focused on regions of the state space with low motion tree coverage which helps to grow the tree outward, but is not focused on reaching the goal. Coverage promotes probabilistic completeness but not necessarily finding a solution quickly.

In artificial intelligence, a central principle for exploring large state spaces is to exploit heuristic information to focus problem-solving in promising regions. The A* heuristic graph search algorithm serves as the central paradigm. In motion planning, the P-PRM algorithm exploits heuristic cost-to-go information to guide growth of its motion tree, with the aim of finding solutions faster than unguided methods. While focusing on low cost regions directs sampling toward the goal, it ignores the effort that can be required for a motion planner to thread a trajectory through a cluttered area. In this way, cost-to-go estimates can encourage the search to focus on challenging portions of the state space, slowing the search. Fundamentally, optimizing solution cost is not the same as optimizing planning effort.

Recent work in heuristic graph search has recognized the separate roles of cost and effort estimates in guiding search, particularly when solutions are desired quickly (Thayer & Ruml, 2009, 2011). In this chapter, we show how to exploit that idea in the context of motion planning. We propose an algorithm, Bayesian Effort-Aided Search Trees (Beast), that biases tree growth through regions in the state space believed to be easy to traverse. If motion propagation does not go as anticipated, effort estimates are updated online based on the planner's experience, and used to redirect planning effort to more fruitful parts of the state space. We implemented this method in the Open Motion Planning Library (OMPL) (Sucan, Moll, & Kavraki, 2012) and evaluate it in five different simulated domains (Kinematic Car and Dynamic Car, Hovercraft, Blimp and Quadrotor). The results suggest that Beast successfully uses effort estimates to efficiently allocate planning effort: it finds solutions faster than RRT, KPIECE, and P-PRM and is the only method able to solve all benchmark instances within the time limit. We see this work as a further demonstration of how abstraction and other ideas from heuristic graph search and non-classical planning can be useful in robot motion planning.

5.2 Previous Work

There has been much previous work on biases for sampling-based motion planners. The two most prominent types in the recent literature have been to bias toward less explored portions of the state space or to bias exploration toward regions of the state space believed to contain low cost solutions. Both of these biases have shown strong results in finding solutions quickly. The two state of the art algorithms considered in this chapter are KPIECE and P-PRM (Le & Plaku, 2014).

5.2.1 KPIECE

Kinodynamic Planning by Interior-Exterior Cell Exploration, or KPIECE , is an algorithm that uses a multi-level projection of the state space to estimate coverage in the state space. It then uses these coverage estimates to reason about portions of the state space to explore next. Expansive Space Trees (EST) (Hsu, Latombe, & Motwani, 1999) and Path-Directed Subdivision Tree (PDST) (Ladd & Kavraki, 2005) also focus on less explored portions of the state space but have been shown to be outperformed by KPIECE . The general all-around good performance of KPIECE has led to its selection as the default motion planner in OMPL.

KPIECE is focused on quickly covering as much of the state space as possible. It always gives priority to less covered areas of the state space. When an area of low coverage is discovered, it attempts to extend the motion tree into that area. It uses a coarse resolution initially to find out roughly which area is less explored. Within this area, finer resolutions can then be employed to more accurately detect less explored areas.

While KPIECE targets exploring unvisited areas of the state space, this may not always be the fastest approach to finding the goal. Certainly targeting exploration toward the goal could help improve performance.

5.2.2 P-PRM

P-PRM (Le & Plaku, 2014) is based on ideas from an earlier planner called Synergistic Combination of Layers of Planning (SyCLOP) (Plaku, Kavraki, & Vardi, 2010). It shares the intuition that information from a discrete abstraction of the workspace can be used to identify low level paths that may lead to the goal. While SyCLOP was shown to be very successful, in recent work P-PRM has been shown to outperform SyCLOP in a variety of planning problems .

P-PRM uses the geometric component of the state space, position and orientation, to construct a Probabilistic Roadmap (PRM) (Kavraki et al., 1996). It generates random states in the geometric space, then connects each state to its nearest neighbors via an edge, forming a graph. The edges in the graph are collision checked and removed from the graph if a collision along them is found. The graph vertices represent regions of geometric space and the edges summarize the connectivity of the regions.

P-PRM runs a Dijkstra search out from the abstract region containing the concrete goal to compute h-values, or heuristic estimates of cost to the goal. It then uses these heuristic values, and the associated shortest paths from the goal to each abstract node, to bias sampling.

It searches by maintaining a queue of abstract states in the graph sorted by decreasing scores, where the score is defined as:

$$score = \frac{\alpha^{NrSel}}{\epsilon + h}$$

α is a parameter that must be greater than zero and less than or equal to one and $NrSel$ is the number of times the abstract state has been chosen. The intuition behind this parameter is to discourage over-relaxation in the abstraction by artificially inflating the cost of associated regions that may be impossible for the vehicle to traverse. No value was provided in the original paper. We tried several values and found 0.5 to give the best performance and matched the reported performance from the original paper (relative to KPIECE's performance). h is the precomputed heuristic value from the abstract graph from the current node and ϵ is used to avoid divide by zero problems at the goal region where $h = 0$.

At each search iteration the abstract state with the highest score is selected. An abstract state along the cheapest precomputed path rooted at the currently selected state is chosen. This state is then used to create a random concrete state within some pre-specified state radius. This is now the "target" state used; similarly to when plain RRT chooses a state uniformly at random. That means that the nearest state in the existing motion tree is chosen as the root for the new propagation which is steered (if possible) toward the random state. Any new abstract states touched by the propagation attempt are added to the queue if not previously enqueued.

P-PRM also samples uniformly at random from the entire state space a certain user-provided percentage of the time. The suggested value for this parameter is 0.85 which means that 15% of the time, P-PRM is just running the basic RRT algorithm.

The other 85% of the time, P-PRM tries to pursue the completion of low cost paths by following its heuristic estimates in the abstract space. It tries to avoid getting stuck during planning by penalizing the score of abstract states when they are examined using the α term given above.

5.2.3 Speedy Search

While RRT and KPIECE are often the reliable workhorses of motion planning, the success of heuristically-informed graph search algorithms such as A* (Hart et al., 1968) in artificial intelligence would suggest that brute-force expansion into all unexplored regions of the state space (in a manner similar to Dijkstra's algorithm) is not an optimal strategy. P-PRM has been shown to provide state of the art performance by exploiting heuristic cost-to-go guidance. Yet recent results in the heuristic graph search community show that exploring the state space based on cost often does not give the best speedup.

In the context of discrete graphs, Greedy search, which focuses on nodes with low heuristic cost-to-goal, is often surpassed by 'Speedy search', which focuses on nodes with a low estimated number of hops (or graphs edges) to the goal (Thayer & Ruml, 2009; Wilt & Ruml, 2014). In this chapter, we present one approach at adapting this idea to motion planning, in which the state and action spaces are continuous and there is no predefined graph structure.

5.3 Exploiting Effort Estimates

While there is not a direct translation of the "number of edges to the goal" concept, there is still a notion of search effort. In heuristic search, the fewer expansions needed to find the goal, typically the quicker a solution is found. In sampling-based motion planning, the unit of measure would be the number of samples, or propagation attempts used to build the motion tree. Each forward propagation of the system state requires collision checking, which is computationally expensive. The fewer propagation attempts made before finding the goal, typically the faster a solution is found (assuming reasonable iteration overhead).

5.3.1 Overview

Bayesian Effort-Aided Search Trees (Beast) is a novel method that tries to find solutions as quickly as possible by constructing solutions which it estimates require the least effort to build. It maintains online Bayesian estimates of the effort of connecting abstract regions of the state space and allocates its search effort to the region of the state space that is estimated to require the lowest effort to connect to the abstract goal region.

Beast exploits a discrete abstraction of the state space. In the experiments reported below, we use a PRM workspace abstraction very similar to the one used by P-PRM. We begin by identifying the geometric component of the state space (again, position and orientation). In the experiments reported below, the abstraction only exists in this subspace. We generate uniformly random states in the abstract space (1000 in the experiments below). As in P-PRM, these states induce a division of the state space into abstract regions (by associating any concrete state with its nearest abstract state). Neighboring abstract states (the 5 nearest in the experiments below) are connected by directed edges, forming a directed graph. If the abstract start and goal regions are not connected, additional samples are taken until they are.

For each edge e, Beast maintains an effort estimate, $ee(e)$, of how many attempts would be required on average to propagate a concrete state contained in the abstract region represented by the source vertex of the edge such that it resulted in a concrete state contained in the abstract region represented by the end vertex. These estimates are initialized by using a geometric collision check along the abstract edge. However, Beast explicitly acknowledges that this quick check in the geometric space is only a rough approximation of a robot's ability to steer from one region to the other.

We represent our uncertain belief about each edge by regarding a propagation attempt as sampling a Bernoulli variable. The uncertainty about each edge is maintained as a Beta distribution (with parameters α, β) over its propagation success probability. In a Beta distribution, α is the number of successful trials and β is the number of failed trials. The initial geometric collision check provides some evidence about this probability, and then each propagation attempt during planning provides additional evidence. In the experiments reported below, an edge with a detected collision is initialized to $\alpha = 1, \beta = 10$, and all other edges are initialized to $\alpha = 10, \beta = 1$. (Very similar performance was obtained by setting $\alpha = 1$ and $\beta = 1$ and this can avoid minor overhead of initialization collision checking.) Successful propagation attempts increase α by one and unsuccessful propagation attempts increase β by one. Based on our belief, we estimate the

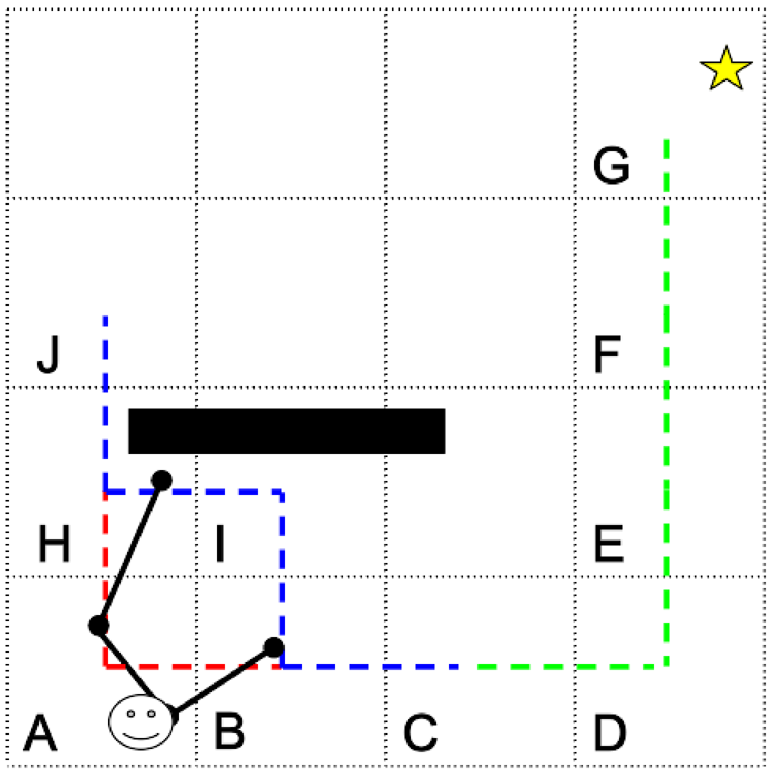


Figure 5-1: The different types of edges reasoned about by Beast.

number of propagation attempts that will be necessary in order to have a successful one as $\frac{\alpha+\beta}{\alpha}$.

Beast uses the abstract graph as a metareasoning tool to decide where it should spend its time growing the motion tree. We only consider abstract regions touched by the motion tree and each edge from the corresponding vertex in the abstract graph represents a possible propagation that the algorithm could attempt. In this way, the edges represent work the algorithm might do and thus they are at least as important as the vertices. This is similar to a traditional discrete graph search frontier. We compute, for each directed edge e, the expected total effort, $te(e)$, required to reach the abstract goal if we start propagating a state from its start region through its end region and onward to the goal. For "exterior" edges, whose start region has not yet had a successful propagation into its end region, this is straightforward: the estimated effort to cross that edge plus the estimated total effort-to-goal from the end vertex. "Exterior" edges are illustrated in Figure 5-1 in blue: $\langle \vec{BC}, \vec{BI}, \vec{HI}, \vec{HJ} \rangle$. Therefore, the $te(\vec{BC})$ is the estimated effort of $\vec{BC}$ plus the estimated efforts of the green edges: $\langle \vec{CD}, \vec{DE}, \vec{EF}, \vec{FG} \rangle$. More formally: if, for every vertex v in the abstract graph, we let $te(v)$ be the minimum over its outgoing edges e of $te(e)$, then:

$$te(e) = ee(e) + te(e.end)$$

"Interior" edges are more complex and are drawin in red: $\langle \vec{AB}, \vec{AH} \rangle$. Unless the goal region has been reached, any interior edge will lead to an exterior edge that has a lower total effort estimate, so such edges may not appear to be useful for propagation. However, recall that our state space abstraction might be very rough, and not all concrete states falling in the same abstract region are necessarily equivalent. We may well want to propagate along an interior edge in order to add additional states to the end region, in the hopes that this will increase the probability of being able to propagate onward from there. An example of where propagating along an "interior" edge ($\vec{AH}$) would benefit a following edge ($\vec{HJ}$) is illustrated in Figure 5-1. The current motion tree leads directly up to an obstacle and may be hard to propagate further, however there is additional unexplored space to the left of where the current tree exists. We model this by assuming that an additional state in the destination region will raise its α by $1/n$, where n is the number of states already in the region. (We want this bonus to inversely depend on the number of existing states, to reflect the decreasing marginal utility of each additional state.) So for an interior edge e with a destination vertex d that currently contains n states in its abstract region,

$$te(e) = ee(e) + \min_{e_2 \in d.out} \frac{e_2.\alpha + 1/n + e_2.\beta}{e_2.\alpha + 1/n} + te(e_2.dest).$$

5.3.2 Details

Pseudocode for Beast is presented in Figure 5-2. The algorithm is passed an abstraction of the workspace, a concrete start state and a concrete goal state. Beast first begins by initializing initial edge effort estimates and then propagating these estimates through the abstract graph outward from the region containing the concrete goal state (Line 3). For efficiency, the collision checking and beta distribution initialization can be done lazily.

We use the pseudocode in Figure 5-3 to estimate the number of propagation attempts needed if the planner were to start by propagating along a specific edge. For exterior edges, this effort value is straightforward (Line 22).

```
Beast(Abstraction, Start, Goal)
1.  AbstractStart = Abstraction.Map(Start)
2.  AbstractGoal = Abstraction.Map(Goal)
3.  Abstraction.PropagateEffortEstimates()
4.  Open.Push(AbstractStart.GetOutgoingEdges())
5.  While NotFoundGoal
6.      Edge = Open.Pop()
7.      StartState = Edge.Start.Sample()
8.      EndState = Edge.End.Sample()
9.      ResultState = Steer(StartState, EndState)
        // Or Propagate With Random Control
10.     Success = Edge.End.Contains(ResultState)
11.     If Success
12.         Edge.UpdateWithSuccessfulPropagation()
13.         If Edge.End == AbstractGoal
14.             Open.Push(GoalEdge)
                // Goal Region To Goal State
15.     Else
16.         Edge.UpdateWithFailedPropagation()
17.     Abstraction.PropagateEffortEstimates()
18.     Open.Push(Edge)
19.     If Success
20.         Open.Push(Edge.End.GetOutgoingEdges())
```

Figure 5-2: Pseudocode for the Beast algorithm.

```
GetEffort($Edge$)
21.  If Not Edge.interior
22.      Return ee(Edge) + te(Edge.End)
23.  Else
24.      Child_Edges = Edge.End.GetOutgoingEdges()
25.      Return ee(Edge) +
        $\min_{Child \in Child\_Edges}$ OptimisticBenefit(Child) +
                                   te(Child.End)


OptimisticBenefit(Edge)
26.  PositiveEffect = 1. / Edge.Start.NumStates
27.  Opt$\alpha$ = Edge.$\alpha$ + PositiveEffect
28.  Return (Opt$\alpha$ + Edge.$\beta$) / Opt$\alpha$
```

Figure 5-3: Pseudocode for calculating an edge effort value.

On Line 25 for interior edges, we examine each of the children of the current edge to see which child edge would require the least effort to arrive at the goal if it were provided another state in its start region. We take the minimum effort over the children and add in the estimated effort of propagating along the current edge.

If effort estimates were static, a single pass of Dijkstra's algorithm would suffice to compute the *te* values. In our case, edge effort estimates change over time so we use an incremental best-first search algorithm inspired by D* Lite (Koenig & Likhachev, 2002) to avoid re-planning from scratch. D* Lite updates the heuristic estimates for cost to go to the goal at each vertex in the graph, in our case we are using effort (*te*) to go instead. We modified the termination criteria of D* Lite's propagation loop slightly to guarantee the entire graph is consistent rather than terminating when the "agent's" current position becomes consistent. While propagating effort at each vertex we also store an effort estimate, *te*, at each edge which is calculated using *GetEffort*.

To reiterate, this value can be seen as an estimate of how many expected samples will be required to reach the goal if you were to choose to propagate along an edge and then choose the minimum effort edges thereafter. A queue called *Open* is then initialized with outgoing edges from the abstract region containing the concrete start state (Line 4). *Open* is sorted in increasing order of edge effort. The search always considers the least effort edge first.

The algorithm proceeds by popping the edge off *Open* with the lowest estimated effort (Line 6). This edge is then sampled at its start abstract region and its end abstract region in lines 7-8. In our implementation, the concrete state in the edge's start region that has been selected the fewest number of times is chosen as the *StartState*. A concrete state is chosen from the edge's abstract end region uniformly at random within some user provided radius centered around the region's generator. In our experiments we tried several values and found a radius of 6 provided good visual coverage over abstract states and also provided good overall algorithm performance.

An attempt is then made to grow the tree from *StartState* to *EndState* using a steering function (Line 9). In our implementation if no steering function was available in OMPL, we instead generated 10 random controls, applied each to *StartState* and the resulting motion that got closest to *EndState* was chosen. [1]

If the newly propagated motion at any point reached the target abstract region (the selected edge's end region), the edge is updated with a successful trial (Line 10-12). This simply adds one to the α value of the beta distribution associated with this edge.

If the target region is not reached, the β value is incremented (Line 16). With each trial to propagate along an edge we update our belief about the effort required to reach the goal by using the edge. This effectively changes the edge "cost" in the abstract graph and we use our incremental search to update the effort estimates throughout the graph based on this local update (Line 17).

If the edge was successfully propagated along, we also add its child edges (outgoing edges from the current edge's end region) to the *Open* list if not already there (Line 20). We re-add the current edge to the *Open* in all outcomes (Line 18).

There is also a special case (Line 13) added which enables us to use sparse abstractions. With sparse abstractions we can compute our effort values more efficiently during each iteration. However, when the goal abstract region is reached, with a sparse abstraction, it may cover a large portion of the state space. Growing the tree into a possibly large goal region may not be focused enough to find a state close to the goal state. To combat this we add a special *GoalEdge* to *Open* (Line 14). This is an edge that when expanded will return a *StartState* from the goal abstract region and an *EndState* focused around[2] the actual concrete goal state (wherever it may be within the abstract goal region).

5.4 Experiments

All experiments were run using control algorithms from the OMPL framework where available (KPIECE and RRT). P-PRM was implemented following closely along with the description and

[1]This functionality was implemented at the control sampler level in OMPL for each domain so any algorithm using "sampleTo" provided by the domain's control sampler received equal benefit. The only algorithm in this chapter that does not use "sampleTo" is KPIECE which uses its own algorithm specific strategy for choosing controls.

[2]This was achieved by leveraging OMPL's builtin "GoalSampleableRegion" class and its "sampleGoal" functionality.

pseudo code included in the paper. Experiments also used OMPL's implementation of a Kinematic Car, Dynamic Car, Blimp and Quadrotor vehicles, as detailed below. We implemented a Hovercraft vehicle in OMPL following Lynch (1999).

5.4.1 Kinematic Car

The mesh used for the Kinematic Car vehicle is shown in Figure 5-4 panel (a). The equations defining the Kinematic Car's motion and control inputs in OMPL are as follows:

$$\begin{aligned}\dot{x} &= u_0 \cdot cos(\theta),\\ \dot{y} &= u_0 \cdot sin(\theta),\\ \dot{\theta} &= \frac{u_0}{L} \cdot tan(u_1)\end{aligned}$$

where the control inputs (u_0, u_1) are the translational velocity and the steering angle, respectively, and L is the distance between the front and rear axle of the car which is set to 1.

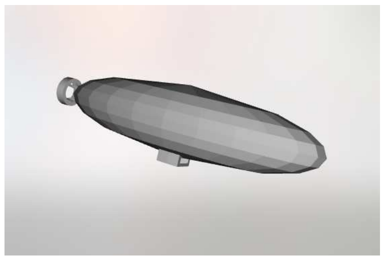
5.4.2 Dynamic Car

The mesh used for the Dynamic Car vehicle is shown in Figure 5-4 panel (a). The equations defining the Dynamic Car's motion and control inputs in OMPL are as follows:

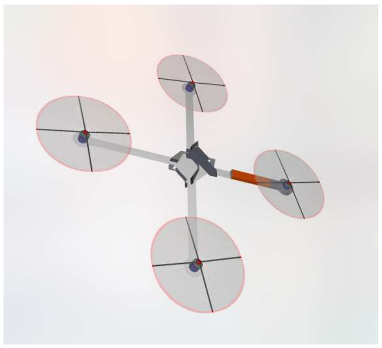
$$\begin{aligned}\dot{x} &= v \cdot cos(\theta),\\ \dot{y} &= v \cdot sin(\theta),\\ \dot{\theta} &= \frac{v \cdot m}{L} \cdot tan(\phi),\\ \dot{v} &= u_0,\\ \dot{\phi} &= u_1\end{aligned}$$

where v is the speed, ϕ the steering angle, the controls (u_0, u_1) control their rate of change, m is the mass of the car (set to 1), and L is the distance between the front and rear axle of the car (also set to 1)

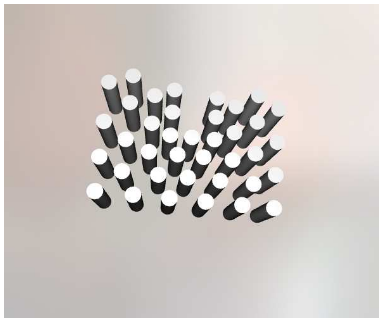
(a)



(b)



(c)



(d)

Figure 5-4: Aerial vehicles illustrations

5.4.3 Hovercraft

The mesh used for the Hovercraft vehicle is shown in Figure 5-4 panel (a). The equations defining the Hovercrafts's motion and control inputs from Lynch (1999) are as follows:

$$\begin{aligned}
\dot{x} &= \frac{F}{M} cos(\theta) - \frac{B_t}{M} x, \\
\dot{y} &= \frac{F}{M} sin(\theta) - \frac{B_t}{M} y, \\
\dot{\theta} &= \frac{\tau}{0.5\ cdotM \cdot R^2} - \frac{B_r}{M} \cdot \theta
\end{aligned}$$

where F is the force exerted by the thrusters and τ is the torque exerted by the thrusters. B_t and B_r are the translational and rotational friction coefficients (both set to 0). M is the mass of the robot and R is the radius of the robot (both set to 1).

5.4.4 Blimp

The mesh used for the Blimp vehicle is shown in Figure 5-4 panel (b). The equations defining the Blimp's motion and control inputs in OMPL are as follows:

$$\ddot{x} = u_f \cdot cos(\theta),$$
$$\ddot{y} = u_f \cdot sin(\theta),$$
$$\ddot{z} = u_z,$$
$$\ddot{\theta} = u_\theta$$

where (x, y, z) is the position, θ the heading, and the controls (u_f, u_z, u_θ) control their rate of change.

5.4.5 Quadrotor

The mesh used for the Quadrotor vehicle is shown in Figure 5-4 panel (c). The equations defining the Quadrotor's motion and control inputs in OMPL are as follows:

$$m\ddot{p} = -u_0 \cdot n - \beta \cdot \dot{p} - m \cdot g,$$
$$\alpha = (u_1, u_2, u_3)^T,$$

where p is the position, n is the Z-axis of the body frame in world coordinates, α is the angular acceleration, m is the mass, and β is a damping coefficient. The system is controlled through $u = (u_0, u_1, u_2, u_3)$.

In the Kinematic and Dynamic Car domains the goal radius was set to 0.1, the remaining domains each used a goal radius of 1. The goal distance of a state was based only on the distance in the XY or XYZ dimensions. Other parameters that were used included a propagation step value of 0.05, min and max control durations of 1 and 100 respectively, and intermediate states were included during planning. The workspace was bounded by $-30 \leq x \leq 30$, $-30 \leq y \leq 30$ and $-5 \leq z \leq 5$.

KPIECE and RRT were run using their default parameters in OMPL. P-PRM was also run using its suggested parameters described in the paper. The state radius size for sampling was shared between P-PRM and Beast. This value was set to 6, which gave good visible coverage over the abstract regions and the best performance over those state radii tried: {2,4,6}. We also leveraged OMPL's built in sampleNear functionality for this purpose.

The workspace obstacle mesh used for the experiments is presented in Figure 5-4 panel (d). For each vehicle, 5 start and goal pairs were used, and for each start and goal pair 50 different random number generator seed values were used. This provided 250 runs for each of the domains. The start states were biased toward the center of the workspace while the goal was biased toward the lower center of the workspace. This set-up tends to generate problems in which the optimal solution threads its way carefully between the obstacles, but it is much easier to take a more costly route around the obstacles. This wide diversity of planning time/solution cost trade-offs directly tests the ability of Beast to estimate planning effort and adjust its behavior accordingly. A motion planner that explicitly tries to find plans quickly ought to exhibit superior performance. A planning timeout of 60 seconds was used.

5.4.6 Results

The results of the experiments are presented in Figure 5-5. Each box represents the middle 50% of the data, with a horizontal line at the median. Whiskers extend to the furthest point within 1.5 times the interquartile range. The remaining outliers are plotted with circles. The 95% confidence interval around the mean is depicted with a gray rectangle. The plots in each panel are sorted according to their means. In order to have enough data points to create plots, algorithm runs that timed out without providing a valid solution are still included in the plot. These runs are represented by the time collected by OMPL after the timeout was issued. Several of the plots have been clipped at the top so that the top two performers remain legible.

In Kinematic Car domain, Beast very clearly outperforms the other three algorithms. In the Dynamic Car domain, Beast is still the best performer, but is more similar to P-PRM. The other algorithms are performing much worse.

In the Blimp domain, Beast has the lowest mean planning time as well as the lowest variance in its performance. In the Quadrotor domain, Beast again has the lowest mean, but P-PRM

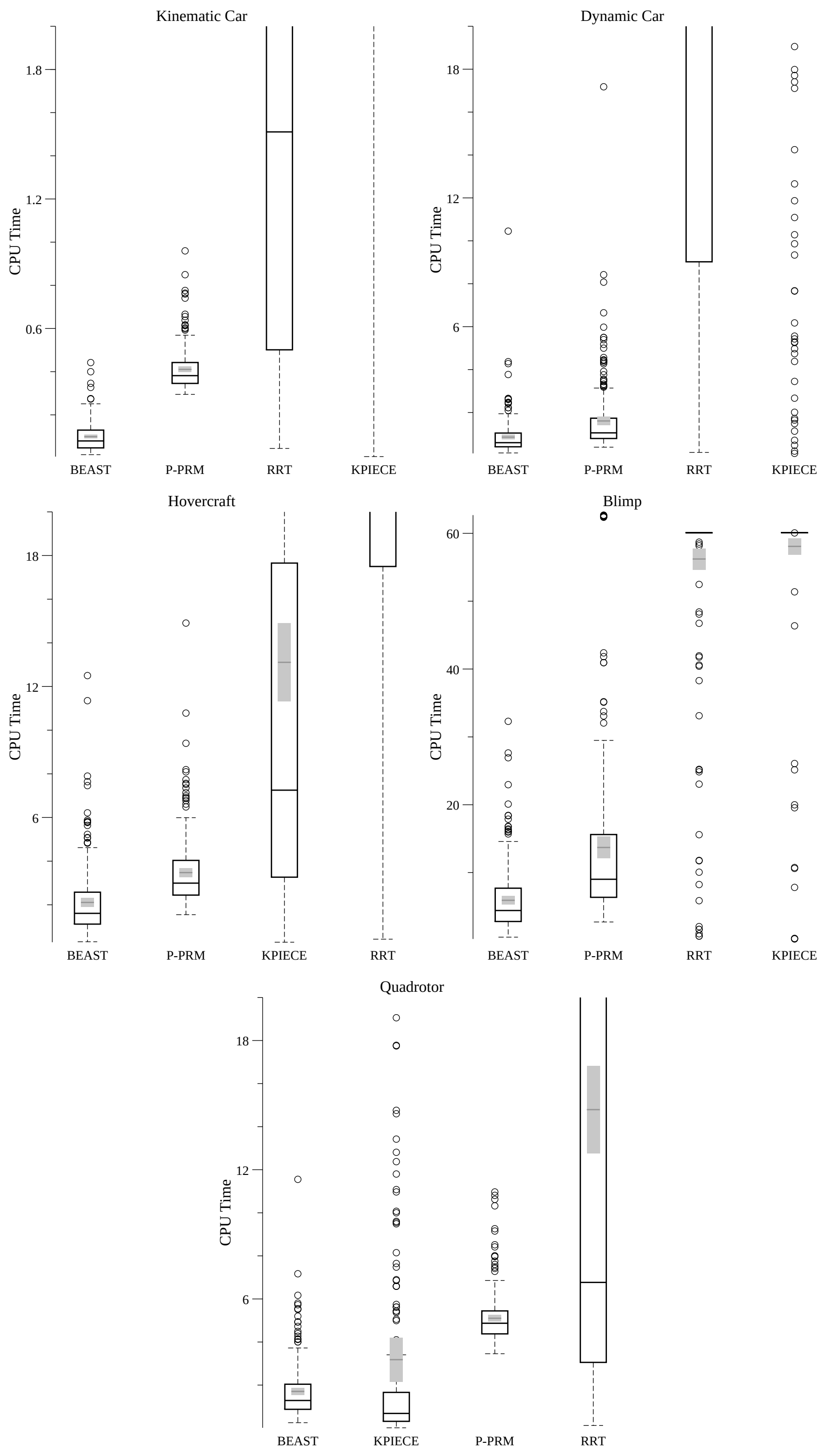


Figure 5-5: Computation time for 5 start goal pairs and 50 random seeds (250 instances).

	RRT	KPIECE	P-PRM	BEAST
Kinematic Car	0	99	0	0
Dynamic Car	108	189	0	0
Hovercraft	116	8	0	0
Blimp	221	238	11	0
Quadrotor	12	2	0	0

Figure 5-6: Number of unsolved instances for 5 start goal pairs and 50 seeds (250 instances).

appears to have slightly lower variance.

A video of the sampling and tree growth of each of the algorithms considered in this half of the chapter can be found at `https://www.youtube.com/watch?v=Or8sQBOrVh4`. It is a top down visualization of a Quadrotor planning instance. It illustrates RRT's slow coverage of the entire state space, KPIECE's rapid coverage of the state space, P-PRM's focus on estimated low cost paths and Beast's focus on finding low effort solutions.

The number of runs where each algorithm was unable to solve an instance is provided in Figure 5-6. In the Blimp domain, Beast is the only algorithm that is able to find a solution to all the instances within the timeout. In the Quadrotor domain, Beast and P-PRM are both able to find solutions to all instances while KPIECE and RRT are not able to within the timeout.

5.5 Discussion

One of the major benefits of Beast is that it explicitly focuses on areas of the state space that it believes will be easy to traverse while heading toward the goal. KPIECE will eventually explore the same regions of the state space but does so without focusing on paths toward the goal. P-PRM does focus on paths leading to the goal, but focuses on paths associated with low cost. These paths can be arbitrarily difficult to construct given obstacle configurations.

This is shown in Figure 5-7 where P-PRM generates many samples (green dots) along abstract paths to the goal, but it is challenging to grow the motion tree (red lines) toward them. Eventually

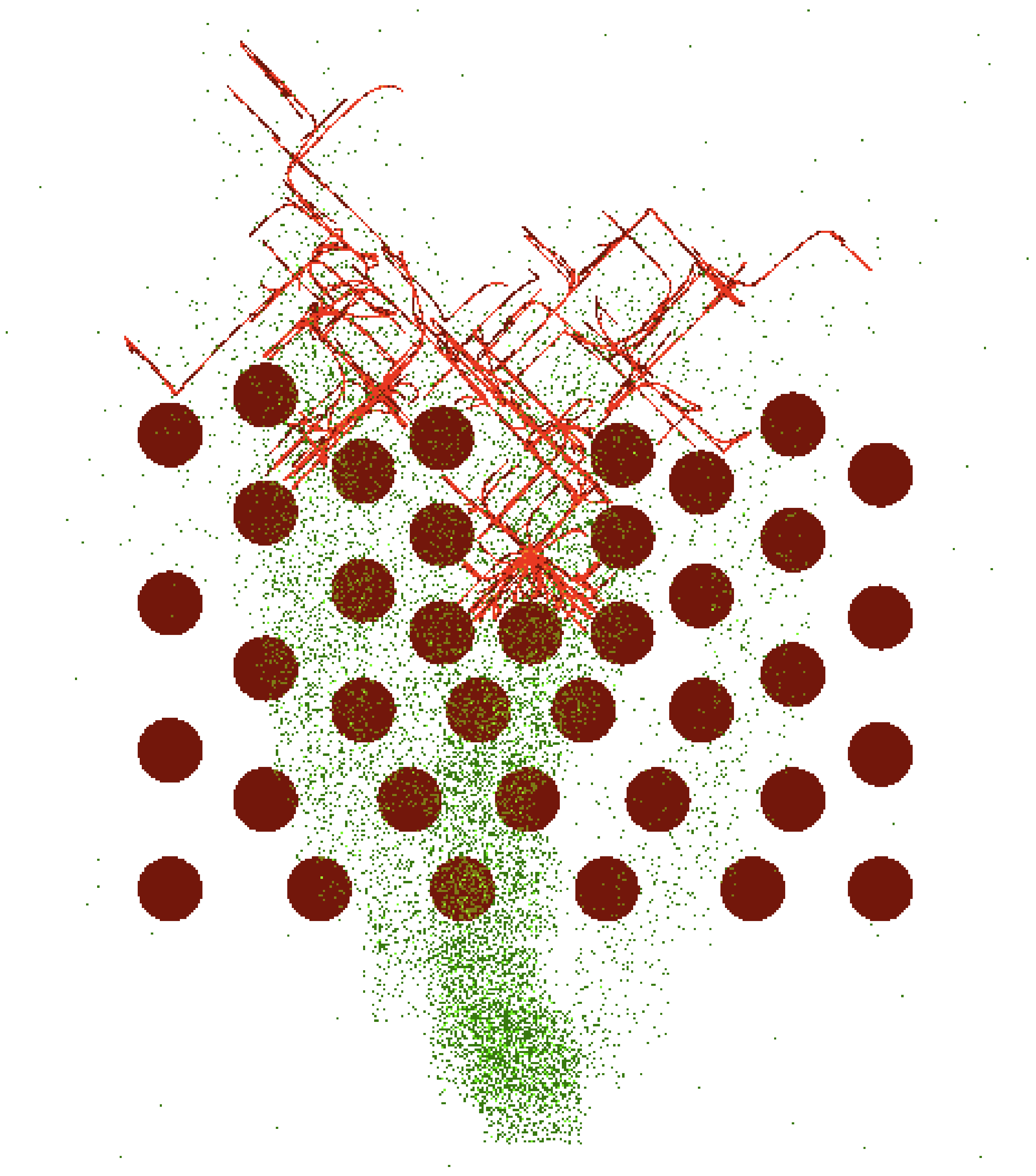


Figure 5-7: P-PRM sampling and tree growth example in the Quadrotor domain (top down view).

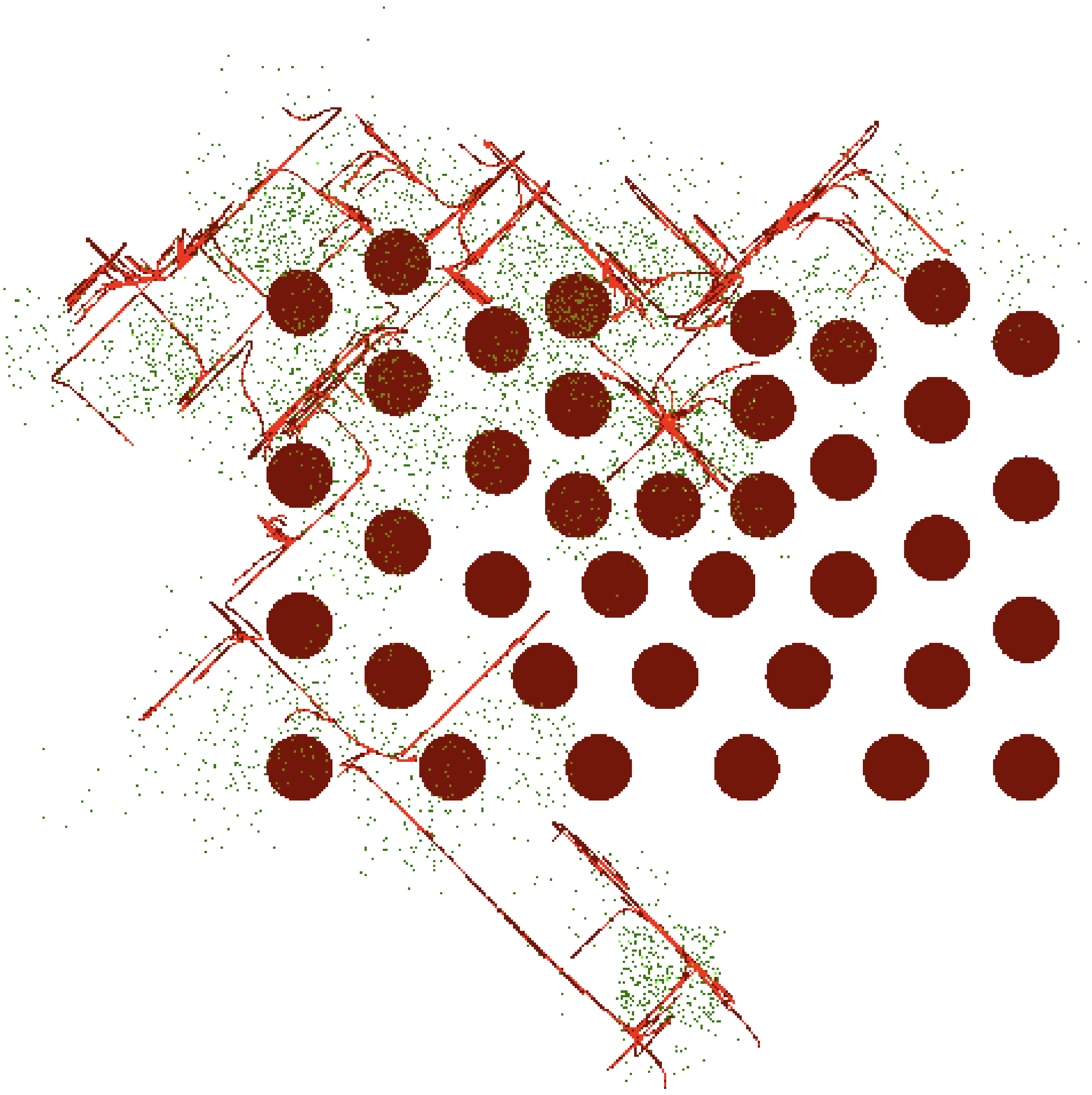


Figure 5-8: Beast sampling and tree growth example in the Quadrotor domain (top down view).

from the uniform random sampling and increasing cost estimates for the states it has selected many times, search begins to spill around and through the obstacles (red circles).

Another feature of Beast that helps it construct its tree more efficiently is that it focuses its tree growth either internal to the existing tree or directly along the fringe of the existing tree. This focus on the boundary of the motion tree is very similar to that of KPIECE, yet the two methods allocate their exploration effort very differently. P-PRM does not focus its sampling near the existing tree and can generate samples arbitrarily far away, which are less helpful when growing the tree through tight spaces.

There are other motion planners that leverage heuristic cost-to-go, but in ways very different from Beast. Informed RRT* (Gammell, Srinivasa, & Barfoot, 2014) uses ellipsoidal pruning regions to ignore areas of the state space that are guaranteed not to include a better solution. BIT* (Gammell, Srinivasa, & Barfoot, 2015) uses heuristic cost estimates directly in its search strategy. Their algorithm tries to expand from the start state toward the goal state through a field of randomly generated samples within the current cost bound. They search for edges in this field in a best first manner looking at effectively f values of nodes. For kinodynamic planning it requires a boundary value problem solver to rewire trajectories between sampled states, making it inapplicable to many problems.

Xie, van den Berg, Patil, and Abbeel (2015) propose a combination of BIT* with a sequential quadratic programming implementation of a boundary value problem solver. They show that state of the art performance can be attained with this method rather than avoiding the use of a two point boundary value problem. However, after contacting the authors to ask for implementation details, they shared with us that the solver they used has since transitioned to a commercial product and the existing freely available products simply can not match their reported performance.

We have presented a new algorithm called Bayesian Effort-Aided Search Trees. Beast exploits and updates Bayesian estimates of propagation effort through the state space to find solutions quickly. Results on a variety of domains showed that Beast has the lowest mean time to a solution and was the only algorithm to find solutions to every instance in the benchmark set. We see this work as reinforcing the current trend toward exploiting ideas from AI graph search in the context of robot motion planning, and providing further evidence that searching under time pressure is a distinct activity from searching for low-cost solutions. This work was published in a 2016 ICAPS PlanRob paper (Kiesel & Ruml, 2016).

Finding solutions quickly is an important feature in many applications, but convergence to an optimal solution is also highly desirable. In the next section, we combine our effort based planner Beast with heuristic cost estimates, yielding an anytime planner which quickly finds a solution and then spends its remaining planning time improving its incumbent solution cost.

5.6 Anytime Search

A common trend between AI graph search and sampling-based motion planning has been the concept of anytime planning. In an anytime planning setting, there is an unknown deadline by which a solution needs to be returned. The strategy most planners take in this setting is to find a solution as quickly as possible and improve that solution as time permits. With the success of the Beast algorithm, we will now introduce the Anytime-Beast (A-Beast) algorithm which builds upon this success. Intuitively, A-Beast leverages Beast to find a solution as quickly as possible and then finds improving solutions by restricting its search to portions of the state space it believes contain lower cost solutions than those already found.

5.6.1 Previous Work

There are several sampling-based motion planning algorithms that operate in this anytime setting but they make assumptions about the solvability of a boundary value problem induced by trying to connect two arbitrary continuous states exactly. As discussed in the previous section, one such algorithm is BIT* (Gammell et al., 2015). BIT* relies on the ability to sample the state space sparsely, and connect states to locally sampled states. This is not always easy to do numerically and it can take a long time for popular methods to converge to a solution with low enough error to be considered as solved. RRT* (Karaman & Frazzoli, 2011) is an older algorithm that attempts to rewire the existing motion tree by similarly requiring exact solutions to the boundary value problem posed by connecting a state to local states within some hypersphere.

General Cost Pruning

There are algorithms that are able to operate in an anytime setting and can guarantee asymptotically optimal convergence without requiring the use of a boundary value problem solver. Such algorithm are presented by Hauser and Zhou (2015). In this algorithm, they add an additional dimension to the underlying planning problem, representing cost. They are then able to provide this problem to an existing motion planner, such as RRT, and rely on its constraint checking to prune states from its tree that would exceed the current cost bound. Each time a solution is found the cost bound is reduced and the planner restarts from scratch.

Stable Sparse RRT

Another algorithm that is able to provide convergence guarantees (without a boundary value problem solver) is Stable Sparse RRT (SST) and its asymptotically optimal variant (SST*) (Li, Littlefield, & Bekris, 2015). The ideas leveraged in this work are similar to the duplicate detection ideas in discrete graph search when deciding how to handle multiple paths to the same vertex in the graph. There has been much work on this in the heuristic search community (Likhachev, Gordon, & Thrun, 2003; Hansen & Zhou, 2007; Thayer, Ruml, & Bitton, 2008). In fact there has even been earlier work leveraging these ideas in lattice-based motion planning (Gonzalez & Likhachev, 2011). In this work, the authors introduce Anytime Repairing A* with Equivalence Classes. These equivalence classes are non-grid aligned duplication detection regions that allow for the identification of dominated states in an action set dictated partitioning of the space. They recognize the incompleteness of applying strict pruning based on dominated states and therefore introduce an ϵ multiplicative penalty to the heuristic value of a dominated state.

SST and SST* attempt to track minimal cost paths to lazily instantiated regions in the state space. As mentioned earlier, this is similar to the notion of duplicate detection. SST is provided with a "selection radius" and "pruning radius". SST* requires a few additional parameters used for dynamically shrinking the size of both of these values over time.

The "pruning radius" value defines the size duplicate detection regions. When a propagation occurs and a new state is being added to the motion tree, the pruning radius around the new state

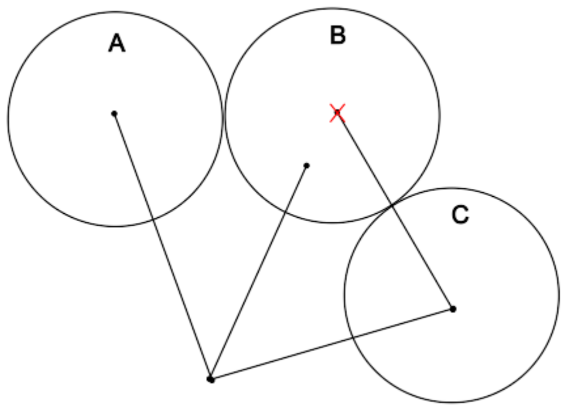


Figure 5-9: SST style pruning.

is considered with all of its previously contained states. Figure 5-9 shows the cases of this purning. If it is empty, the new state defines a new region of the state space, a new duplication detection region (Figure 5-9 region ‘A’). The new state is also defined as the representative for that region, being the state in it with the lowest g value. When a new state is going to be added into this same region, it is compared against the minimal g value state, the representative for the region. If the new state has a larger or equal g value, this new motion tree branch is discarded. This is the case in (Figure 5-9 region ‘B’, the existing state has a higher g value than the new state and is removed. If the new state has a smaller g value, the new state becomes the representative for the region and the old state is marked as “inactive”, and the tree is possibly cleaned up.

The tree cleanup process is one of the benefits of using SST. This helps maintain a sparse tree which enables faster nearest neighbor queries against the standard KD-tree data structure commonly used for tracking the motion tree. States can be safely removed from the tree using a leaf to root sweep. At a leaf, there are no children, so if the state is “inactive”, it can be removed from the tree. The algorithm then proceeds to its parent: if this state has no “active” children, it can be removed. This effectively removes any states from the motion tree that are no longer needed to maintain the motion tree. Any “inactive” state, with no “active” children, is essentially a “dead subtree”, and can be safely removed. This can be seen in (Figure 5-9. The region defined by ‘B’ contains a state that can be safely removed from the tree since no other states rely on its presence in the tree. This process can also be implemented lazily to avoid unnecessary tree traversals.

The search strategy used in SST is similar to RRT, wherein a random sample is generated within the state space, a selection process is applied, the motion tree is propagated from the selected state, and finally for SST only, the duplicate detection reasoning is applied.

Anytime Explicit Estimation Search

The last algorithm of significant relevance to A-Beast is Anytime Explicit Estimation Search (Thayer, Benton, & Helmert, 2012). In this work, the authors adapt EES (Thayer & Ruml, 2011), a discrete graph search algorithm, to an anytime setting. Anytime EES operates similarly to EES continually searching for the nearest solution (in terms of distance to goal) within the current cost bound, which in the AEES is dictated by the current incumbent.

5.6.2 A-BEAST

In the anytime version of Beast, A-Beast, we adopt ideas from all of these previous works. We add three new major components to the algorithm. From a high level they all have to do with various aspects of cost reasoning. The first of the three is cost pruning the motion tree based on ideas from Hauser and Zhou (2015). We are very easily able to prune any extensions to the motion tree that exceed the cost of our current incumbent.

The second is to directly apply the ideas from SST and SST* to the insertion and maintenance of the motion tree. We use this algorithm as a filter to help decide which states and motions we add into the tree. By doing this, we can continually reduce cost in the motion tree as the planner has more time. RRT* could also be leveraged for the same purpose if the underlying problem admitted closed form solutions to the boundary value problem for the motion model. We do not consider these types of problems in this work.

Lastly, to help A-Beast focus its search, we introduce cost reasoning along each edge. We maintain a belief about how expensive we believe a path through an abstract edge will be. This is a similar notion to how we tracked an estimate of how difficult we thought constructing a path through an edge would be. We can then create an estimate of the probability that a solution less than the current incumbent could exist through this region/edge. This will be explained in detail later on. However, we can then emulate the ideas from Anytime EES by searching for the lowest effort solution with a high probability of being within the current bound.

The pseudocode for A-Beast is presented in Figure 5-10. The algorithm operates along the same lines as the original Beast algorithm. The major changes to be discussed are on Lines 6, 7, 8, 11, 12, and 25.

The algorithm begins similarly to Beast. In A-Beast however, we track an incumbent which is initially set to infinity. The algorithm then proceeds to initialize its abstraction and add the initial edges to the open list. On Line 6 the loop changes to a time based loop which allows the algorithm to continue searching until the time runs out. To reiterate, in this anytime context, there is a time limit but it is not known to the algorithm.

The edge selection process in Line 7 is only slightly more complicated than its predecessor. In Beast the "best" edge was simply the one with the lowest estimated effort to goal. This updated version needs to take into account a notion of cost. At a high level, we want to focus our search on edges that have a low effort to goal and a high probability of containing solutions with a cost less than the incumbent. We need to consider a probability because there is much uncertainty in the cost to an abstract state and from that state to the goal. We create informed estimates based on cost estimates computed from the abstract graph as well as providing online updates to our estimates based on cost experience while constructing the motion tree.

For each edge, we track a g, cost-to-come, and h, cost-to-go, estimate. The g estimate is in the form of a Gaussian distribution estimated using the g values of the "active" states that fall within an edge's start abstract region. We incrementally track the Gaussian for each edge using the following update rules:

$$\mu' = \mu + \frac{x - \mu}{|history|}$$

```
A-Beast(Abstraction, Start, Goal)
 1.  Incumbent = INFINITY
 2.  AbstractStart = Abstraction.Map(Start)
 3.  AbstractGoal = Abstraction.Map(Goal)
 4.  Abstraction.PropagateEffortEstimates()
 5.  Open.Push(AbstractStart.GetOutgoingEdges())
 6.  While HaveMoreTime
 7.    Edge = SelectEdge(Incumbent)
 8.    StartState = SSTSelect(Edge.Start.Sample())
 9.    EndState = Edge.End.Sample()
10.    ResultState = Steer(StartState, EndState) // Or Propagate With Random Control
11.    If ResultState.g ≥ Incumbent
12.          OR Not SSTUpdate(ResultState)
13.      Edge.UpdateWithFailedPropagation()
14.      Continue
15.    Success = Edge.End.Contains(ResultState)
16.    If Success
17.      Edge.UpdateWithSuccessfulPropagation()
18.      If Edge.End == AbstractGoal
19.        Open.Push(GoalEdge) // Goal Region To Goal State
20.    Else
21.      Edge.UpdateWithFailedPropagation()
22.    Abstraction.PropagateEffortEstimates()
23.    If Success
24.      Open.Push(Edge.End.GetOutgoingEdges())
25.    CheckGoal(ResultState, Open)
```

Figure 5-10: Pseudocode for the A-Beast algorithm.

```
CheckGoal($ResultState, Open$)
26.  If Goal(ResultState)
27.      Incumbent = State.$g$
28.      Open.clear()
29.      Open.Push(AbstractStart.GetOutgoingEdges())
```

Figure 5-11: Pseudocode for the handling a new path to the goal

and

$$\begin{aligned} t_1 =& \sigma^2(|history| - 1) \\ t_2 =& (x - \mu)(x - \mu') \\ \sigma' =& \sqrt{\frac{t_1 + t_2}{|history|}} \end{aligned} \tag{5.1}$$

The h value is a scalar that is extracted from the abstract graph. This is simply the Euclidean distance between generators in the abstract graph of the edge in question's start region, along the shortest path to the abstract goal region scaled by the maximum velocity of the vehicle. This value is not an admissible estimate of cost-to-go because it can overestimate the actual cost-to-go. The simplest reason for this is because the generators exist centered in abstract regions which can cause an abstract path to pass widely around obstacles whereas the true shortest path could hug more closely to obstacles. The last component used to construct an estimate of the cost of a solution passing through an abstract edge is an estimate of the error between the actual h cost values if the abstract graph were followed and those extracted from the abstract graph. This error estimate is also tracked as a Gaussian distribution. It is constructed by comparing the difference in g values from the abstract graph and the g values from the motion tree. Comparing these two values provides an estimate of how much the abstract distances in our graph differ from what the true capabilities of the vehicle are. By using g values to construct this error term, we can incrementally update it as search is progressing rather than waiting to arrive at the goal to compute the difference in h values from the true cost-to-goal found. The total estimate of cost through an edge to the goal is then:

```
SelectEdge(Incumbent)
30.  Loop Over edge $\in$ Open
31.      Return edge $\propto$ edge.$g$ + edge.$h$ < Incumbent
```

Figure 5-12: SelectEdge function called from Line 6 in the A-Beast pseudocode.

$$edge_{cost} = edge_g + edge_h \cdot error_{global}$$

Based on this estimate we then calculate the probability of a solution with cost less than the incumbent existing through an edge using the cumulative density function:

$$P(edge_{cost} < incumbent) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{incumbent - edge_{cost}.\mu}{edge_{cost}.\sigma\sqrt{2}}\right)\right]$$

As show in Figure 5-12, we approximate choosing the lowest effort edge with the greatest probability of containing a solution path less than the incumbent by iterating through the Open list, sorted on effort, and returning an edge with probability proportional to the previously described probability.

On Line 8, we adopt a strategy similar to SST's when selecting a state in the existing tree to propagate from. Instead of a uniformly random sample state, we use a random state within the selected edge's start abstract region. We then select the nearest "active" state to the target state with minimum g value.

With a state from within the motion tree and a target state selected the algorithm proceeds as Beast would. It propagates from the selected motion tree state toward the target state. In the experiments that follow, controllers were used if available in OMPL, otherwise 10 random trajectories were constructed and the one that minimized the difference between the initial state and target state was selected.

After the trajectory is constructed, the resulting state is evaluated against the current incumbent cost bound in Line 11. If the new state exceeds the cost bound, the trajectory is not added to the motion tree and the resulting propagation attempt would be recorded as a failure. This is

```
SSTSelect($State$)
32.  States = NearestWithin(State, SelectionRadius)
33.  **Return** $\arg\min_{s \in States : s.active} s.g$ // *This set may be empty*

34.  k = 1
35.  **While True**
36.      States = KNearest(State, k)
37.      **Return** $\arg\min_{s \in States : s.active} s.g$
38.      k += 5
```

Figure 5-13: SSTSelect function

similar to how cost pruning can be seen as additional cost obstacles along an additional axis in the state space. However, this subtlety has another effect in A-Beast's reasoning. When states were originally pruned in Beast, it was because of obstacle collisions. Therefore, the effort estimates reflected the raw estimate of how difficult it was traverse along an edge. However, with the addition of cost reasoning, the effort estimates become a moving estimate of how difficult it is to traverse along an edge given the current cost bound. As the cost bound dictated by the stream of incumbent solutions shrinks, more expensive edges, with regard to cost, are seen as harder to propagate along with respect to the current constraints.

Note that A-Beast uses g cost pruning rather than f cost pruning. With an admissible heuristic estimate of cost to goal, h, we could construct an admissible f value for a state:

$$f = g + h$$

However in our experiments, using an admissible estimate of cost-to-go in the concrete space (that did not overestimate the actual cost[3]) proved to be a weak contribution to the pruning function

[3]The heuristic that was being computed was the direct Euclidean distance between the state and goal scaled by

```
SSTUpdate(State)
39.  Witness = ClosestWitness(State)
40.  If Witness == None Or distance(Witness, State) > PruningRadius
41.      Witness = New Witness
42.      Witness.state = State
43.      Witness.rep = State
44.  If Witness.rep == State Or State.g < Witness.rep.g
45.      OldRep = Witness.rep
46.      Witness.rep = State
47.      MarkInactive(OldRep)
48.      CleanupTree()
49.      Return True
50.  Return False
```

Figure 5-14: SSTUpdate function

and added unnecessary overhead. This is why we used a g value instead of an f value.

If the cost check is not violated, the next check is to apply the logic from SST on Line 12. The logic is contained within Figure 5-14. This function encapsulates the ideas from SST of checking whether or not a new region is reached or if an existing region is reached via a cheaper path. If a cheaper way is found to an existing region, the previous representative for that region is marked as inactive and replaced with the new state. The tree is then examined to see if any inactive branches can be removed.

If the pruning decides that the new tree branch should be pruned, it is not added to the motion tree and the propagation attempt is seen as a failure. Similar reasoning is applied to the propagation failure here as in the general cost pruning. As the algorithm runs, the g value of the representative

the maximum velocity of the vehicle.

of each region will decrease. It will at some point become very difficult to reduce the g value any further and the algorithm will recognize this through its tracking effort estimates.

If the pruning allows the addition of the new branch, it is added and the original Beast logic is applied. In the original Beast algorithm, when an initial solution was found, the algorithm terminated. In A-Beast, a new state is checked to see if it is satisfies the goal criteria on Line 25. In Figure 5-11 if the state is a goal state, the incumbent is updated, the open list is cleared, and the initial abstract edges are re-added to the open list. The motion tree and the distributions are all maintained, but the best first search among edges to propagate along is restarted with an empty open list.

5.7 Experiments

We ran similar experiments for A-Beast as we did with Beast, however we added several new workspaces. In Figure 5-15 are the 6 workspaces. Panels (a)-(e) were used for the 2 dimensional workspace vehicles (Kinematic Car, Dynamic Car, Hovercraft), while (e)-(f) were used for the 3 dimensional workspace vehicles (Blimp and Quadrotor).

All experiments were run in OMPL. In order to encourage coverage of all algorithms we allowed each run to execute for 5 minutes. For each vehicle and workspace pairing we present 6 plots. In the legend of each plot, an algorithm name will be suffixed with "{+}" to notate that the algorithm was able to solve all instances in the set within the timeout.

Each plot includes 5 anytime algorithm variants: A-Beast using SST (static radii) and SST* (dynamically decreasing radii), the original SST and SST* algorithms, and Restarting RRT with Pruning. The A-Beast algorithms follow directly from the description above. SST and SST* follow from the original journal paper as well as the existing OMPL implementation which had to be slightly modified. (The new implementation outperforms the existing implementation.) Restarting RRT with Pruning is a straightforward algorithm where a plain RRT is constructed and each time a goal is found, the algorithm restarts using pruning on the previous incumbent.

Also included are three single shot planning algorithms from the previous section for comparison purposes. KPIECE is included as well as P-PRM and the original Beast. The inclusion of this

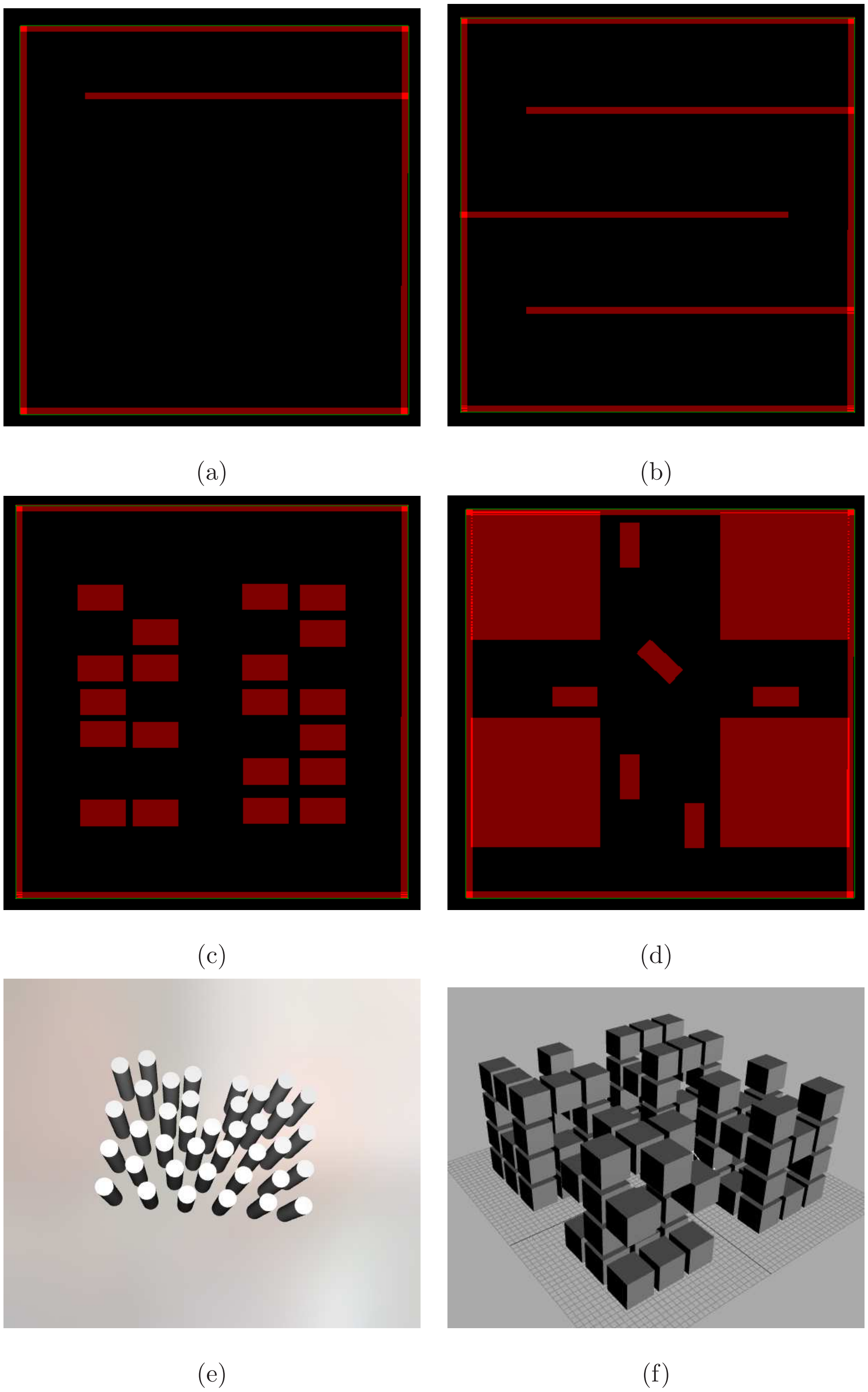


Figure 5-15: Workspaces used in the A-Beast experiments. (a) Single wall, (b) ladder, (c) parking lot, (d) intersection, (e) forest, (f) cube world

algorithms can help create a strong comparison between the two types of planning.

In all line plots, the legend is sorted in the order of the lines for easier comparison of algorithms.

The first plot in each series is instance coverage as a function of time. This is a very important metric to exemplify how quickly algorithms are able to solve all instances within the set.

The next plot is solution cost as a function of time. At a given time point, if an algorithm has not solved an instance, the solution cost for that instance is considered to be a large finite constant: 100,000. Using a large finite constant rather than infinity allows us to compute a finite average across the x-axis. This leads to algorithms not being shown on the plot until they have provided at least one solution for every instance in the set.

The next plot is Goal Achievement Time (GAT) as a function of time (Kiesel, Burns, & Ruml, 2015). This metric is how quickly a goal is actually achieved after the instance was issued. It is the sum of planning time and execution time. This is a very practical metric when a solution will actually be executed. Minimizing planning time can result in very long solutions, while minimizing solution cost can result in very long planning times. These plots are targeted to reward a balance between these two quantities.

The next plot is another GAT plot. However, in this case, we assume an oracle that could decide when the optimal time to terminate planning would be to give the best GAT for the planning period. Also, under each algorithm label is the mean planning time from which the GAT values were taken.

The next plot in the series is an anytime solution quality plot which is the IPC Anytime Metric. These plots show solution quality as a function of time. They attempt to reward coverage and low solution cost by rolling these values together. This type of plot is used by the International Planning Competition and has become their traditional anytime plot. As you may notice, they can be hard to interpret without seeing the underlying data. For example, it may be desirable to solve all instances rather than only find low cost solutions in only a subset of instances. However, it is difficult to extract coverage information and cost information when only solution quality is shown.

The last plot in the set shows how frequently solutions are being found by each algorithm. It maybe be very desirable, when the planning deadline is not known, to find improving solutions as quickly as possible with small cost deltas rather than finding solutions infrequently with larger cost deltas. This notion is described by Thayer (2012).

5.7.1 Kinematic Car

Figure 5-16 through Figure 5-20 present the results of A-Beast in the kinematic car domain over each of the workspaces shown in Figure 5-15.

We can immediately see that the A-Beast variants are able to solve all instances in the set within the timeout. Beast and P-PRM are also able to find solutions to all instances and do so slightly faster than A-Beast as is illustrated in the coverage plots. KPIECE is unable to find solutions to all instances in any of the workspaces within the timeout.

As shown in the cost plots, while Beast and P-PRM are able to find solutions slightly faster, their solutions are often about twice as expensive as the solutions found by A-Beast. Eventually, after significantly more time in some cases, the SST variants find solutions to all instances with better cost than A-Beast.

In terms of the average goal achievement time across the planning duration, the A-Beast algorithms are consistently the best performers. Beast and P-PRM consistently follow as the third and fourth best while the SST variants are consistently fifth and sixth. Not surprisingly, in terms of best GAT across the entire planning time, the A-Beast are best as shown in the next GAT box plots.

In the Anytime Solution Quality plots, the SST variants perform best, followed by the A-Beast algorithms. This is not too surprising because the solution quality metric has to do with factor of best solution found and the SST variants eventually find the cheapest solutions in the planning duration.

In the plots showing time since last solution found, it can be seen that in most cases, the A-Beast variants are finding solutions more frequently than the other algorithms. The exception is the forest workspace where the SST variants are finding solutions more frequently on average. However, the performance difference is small in this case.

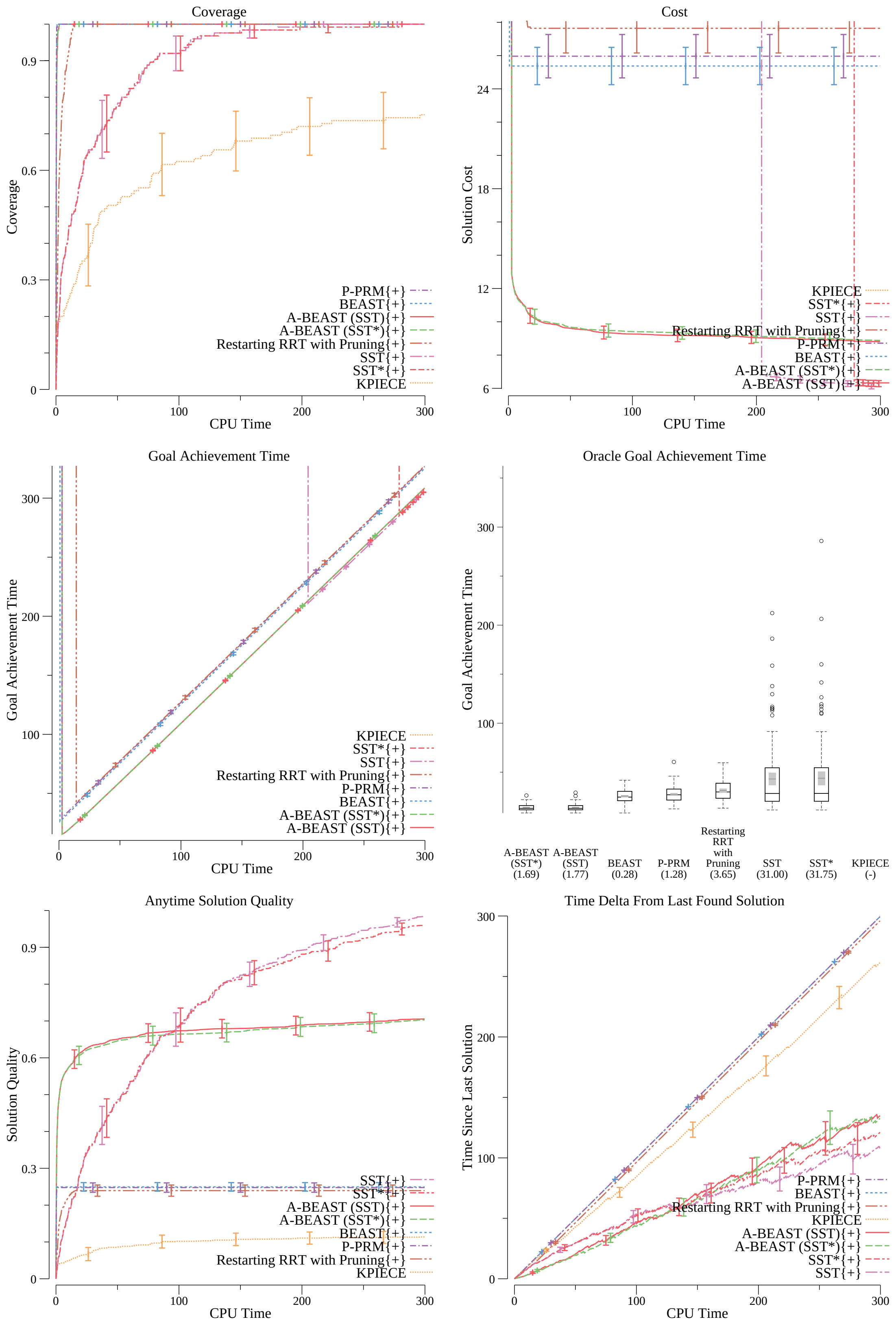


Figure 5-16: Kinematic car results in the forest workspace.

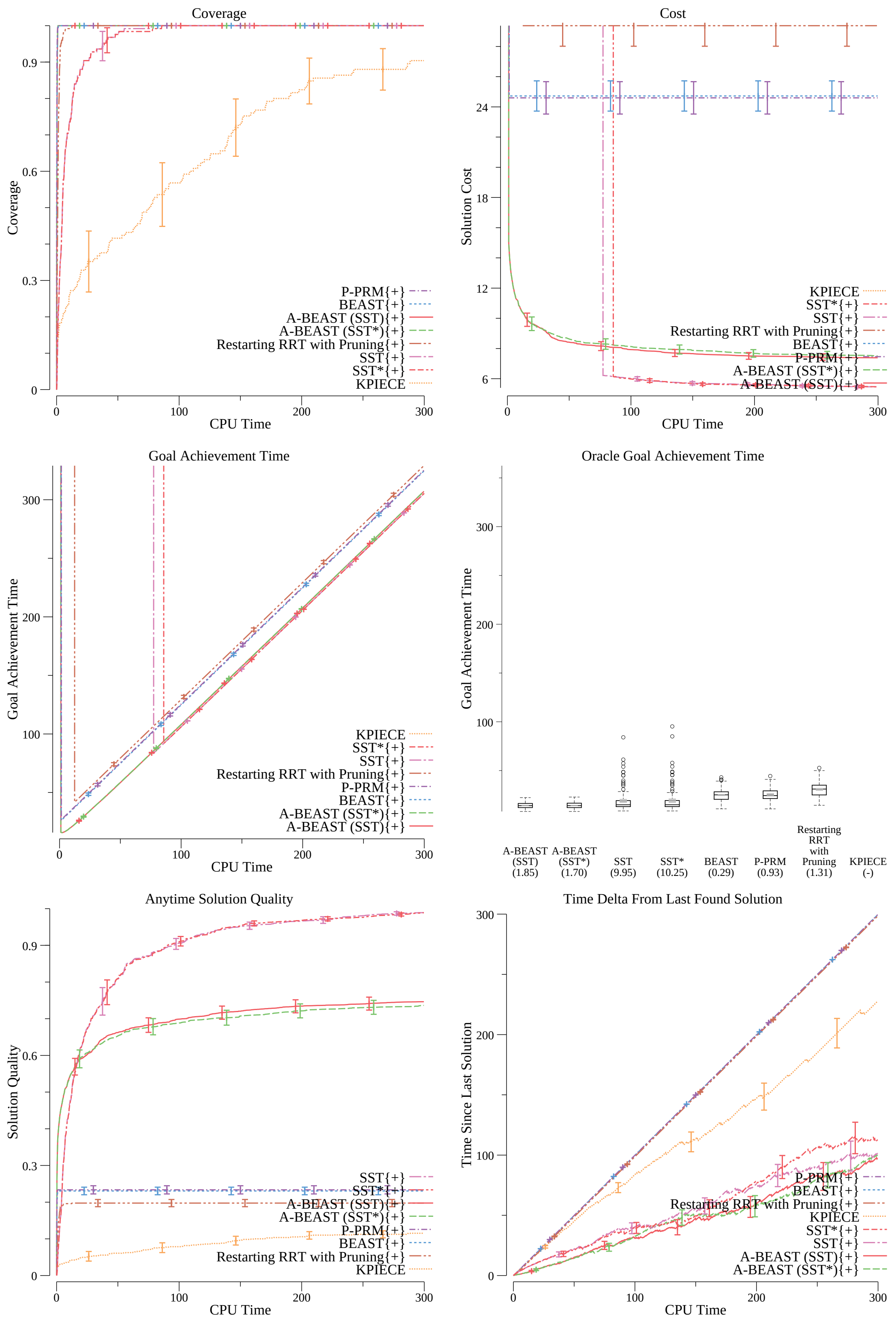


Figure 5-17: Kinematic car results in the single wall workspace.

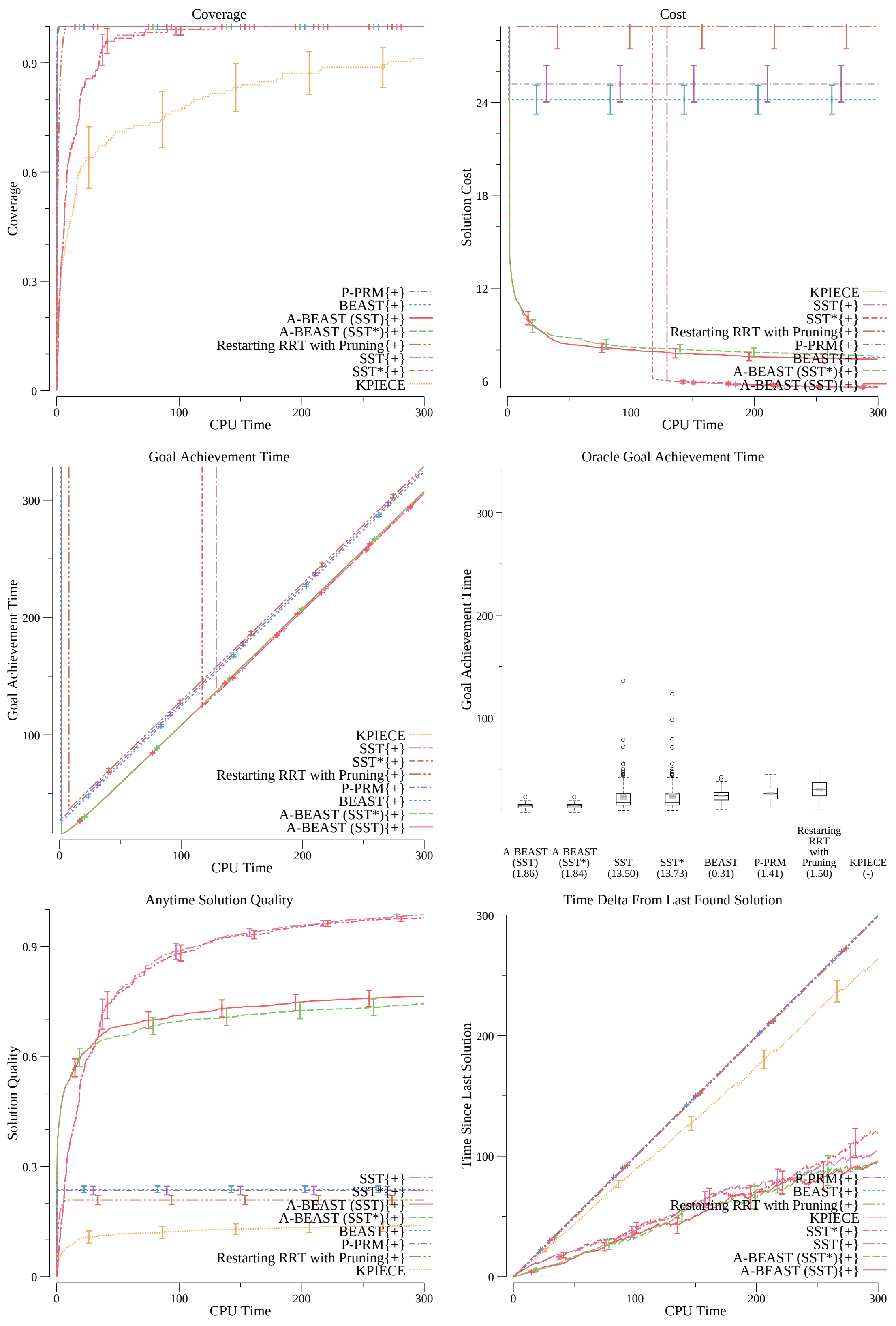


Figure 5-18: Kinematic car results in the ladder workspace.

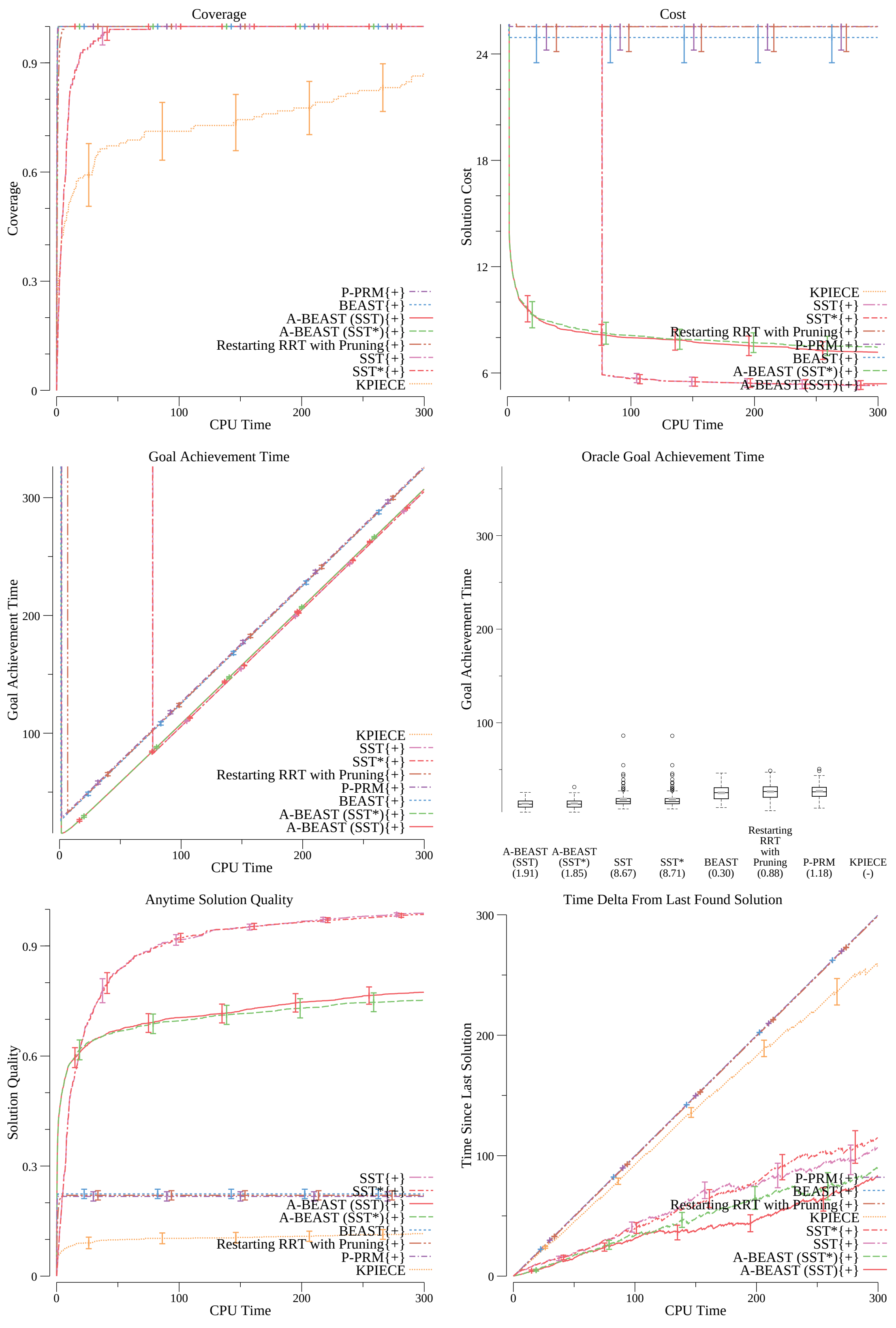


Figure 5-19: Kinematic car results in the parking lot workspace.

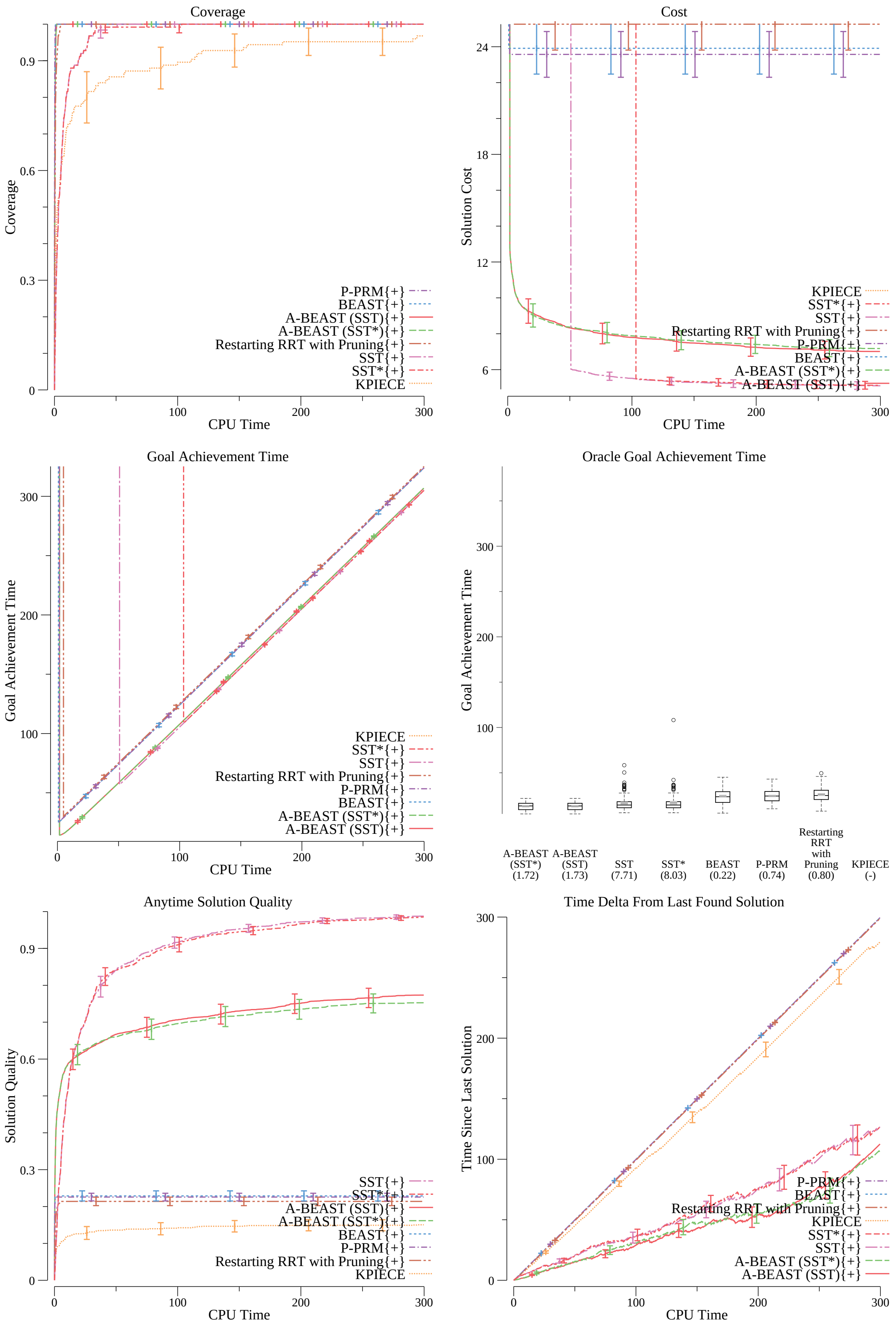


Figure 5-20: Kinematic car results in the intersection workspace.

5.7.2 Dynamic Car

Figure 5-21 through Figure 5-25 present the results of present the results of A-Beast in the dynamic car domain over each of the workspaces shown in Figure 5-15.

The dynamic car results are very similar to the kinematic car results. A-Beast, Beast, and P-PRM are the only algorithms to solve all instances across all workspaces. The SST variants and Restarting RRT with Pruning start to be unable to solve all instances in some of the workspaces within the timeout. Again, KPIECE is unable to solve the entire instance set for any workspace with the dynamic car.

A-Beast, Beast, and P-PRM are all quite competitive with each other in terms of the time until all solutions are found in each of the workspaces. Beastand P-PRM slightly outperform A-Beast again with this vehicle. In terms of cost, we can see that A-Beast, while slightly slower to get full coverage, is on average finding solutions about half as expensive as Beast and P-PRM.

For average GAT across the planning duration, A-Beast is consistently best, with the exception of the forest workspace where Beast slightly outperforms A-Beast using SST. In this case, A-Beast with SST* is still the better than Beast. When using an oracle to select the best time to terminate planning and start executing to minimize GAT, this domain is more diverse. In some workspaces A-Beast performs best, while in others one of the SST variants performs best. However, on average Beast has fewer outliers in the box plots than the SST variants.

In the anytime solution quality plots, the SST variants still perform best where they are able to have full coverage over the instance for the workspace. The exception here is in the forest workspace where A-Beast performs better on average across the planning duration. In the time since last solution found plots, the A-Beast algorithms initially are able to find solutions at a good pace, but then start taking longer between finding solutions. A-Beast only is able to find solutions on average the most frequently in the forest workspace. In the other workspaces A-Beast dominates the beginning of the planning duration but is eventually overtaken by the SST variants and Restarting RRT with Pruning.

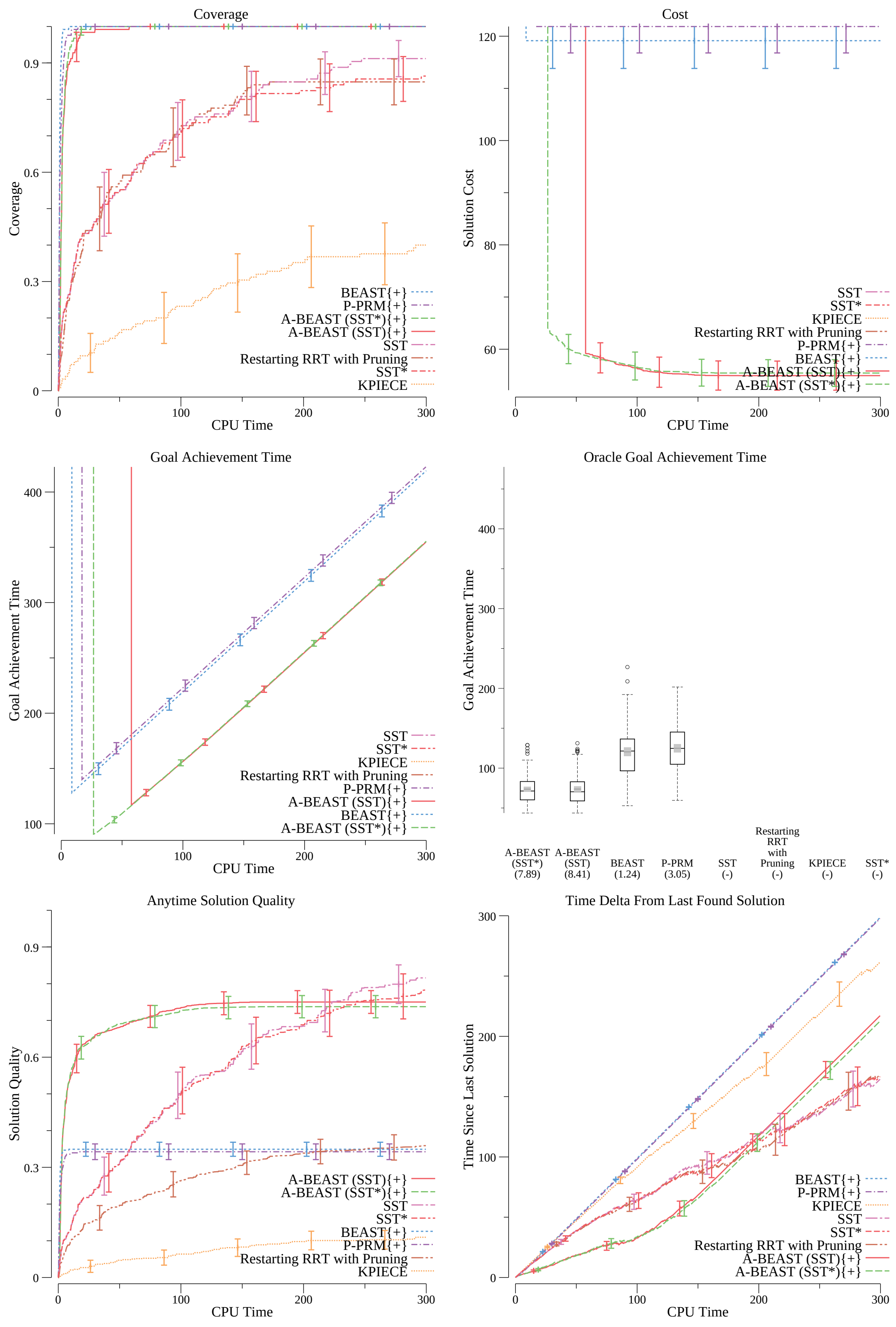


Figure 5-21: Dynamic car results in the forest workspace.

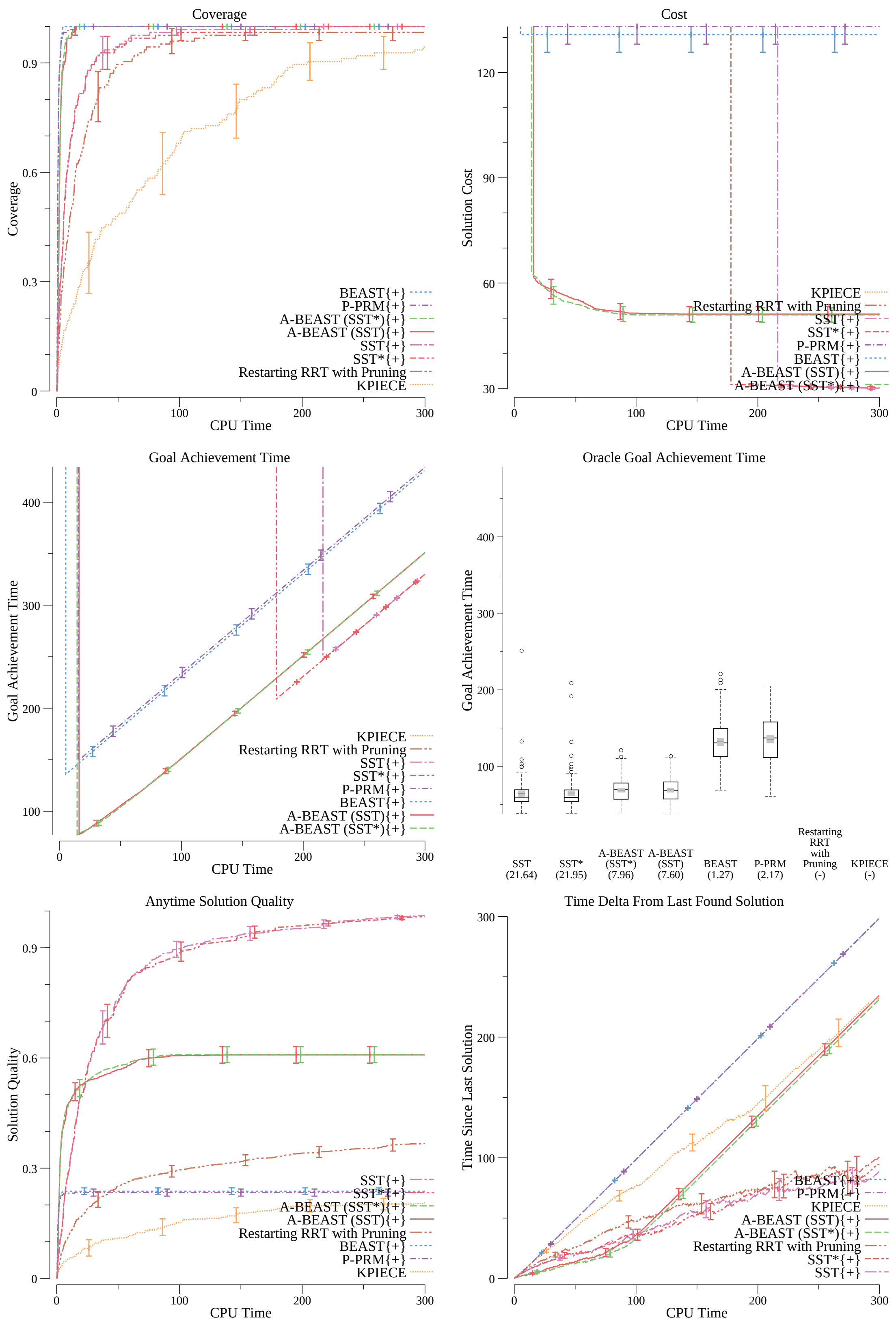


Figure 5-22: Dynamic car results in the single wall workspace.

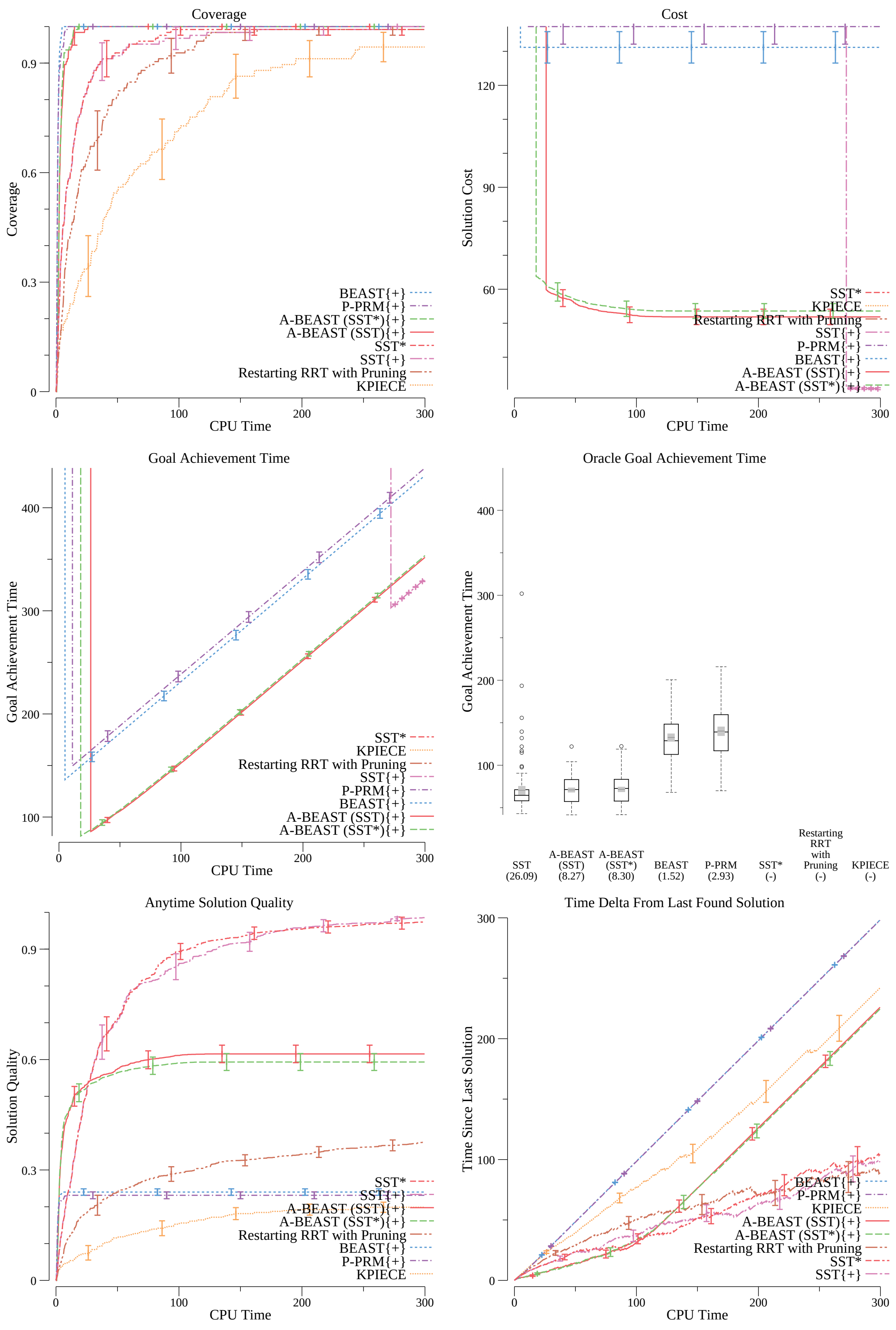


Figure 5-23: Dynamic car results in the ladder workspace.

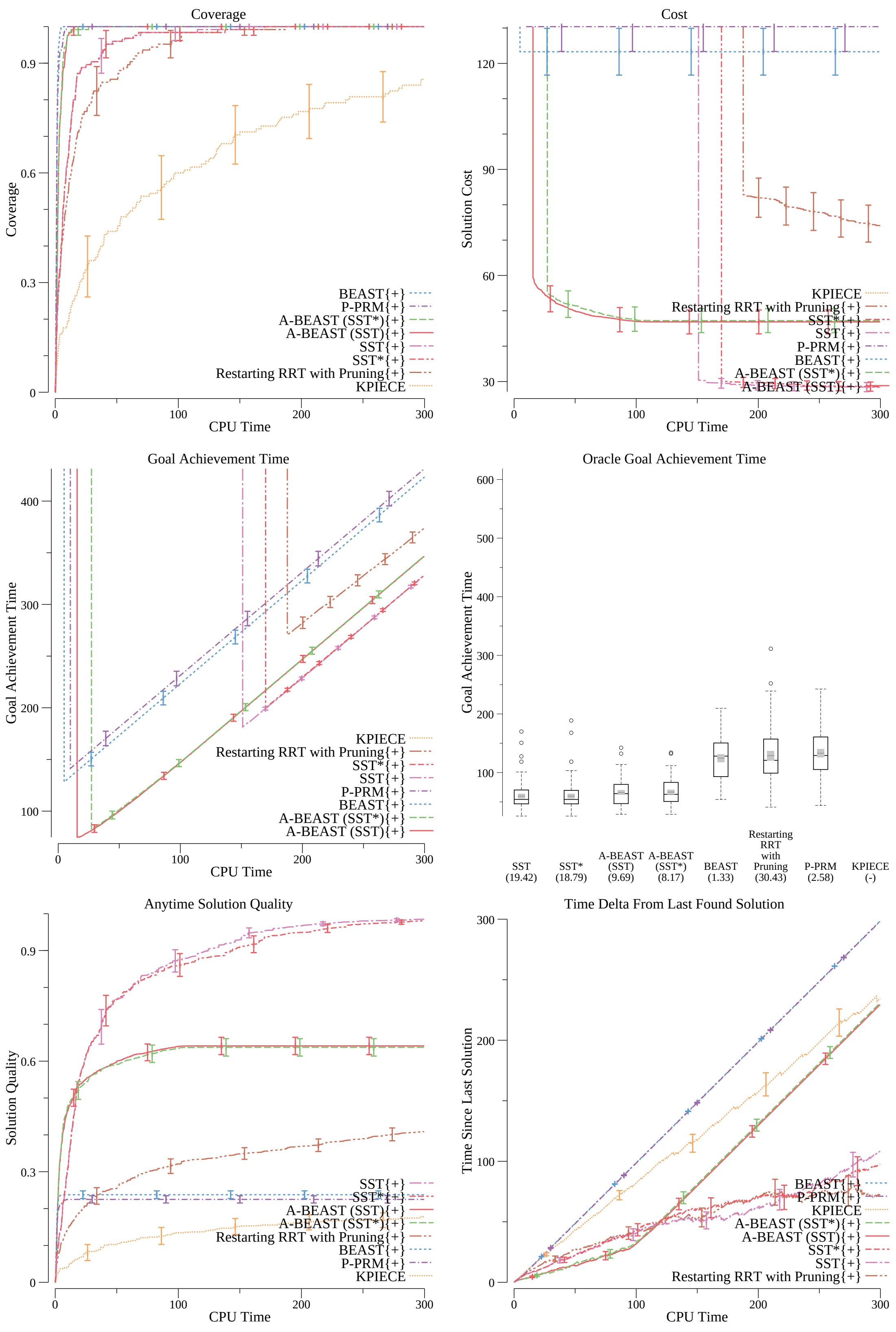


Figure 5-24: Dynamic car results in the parking lot workspace.

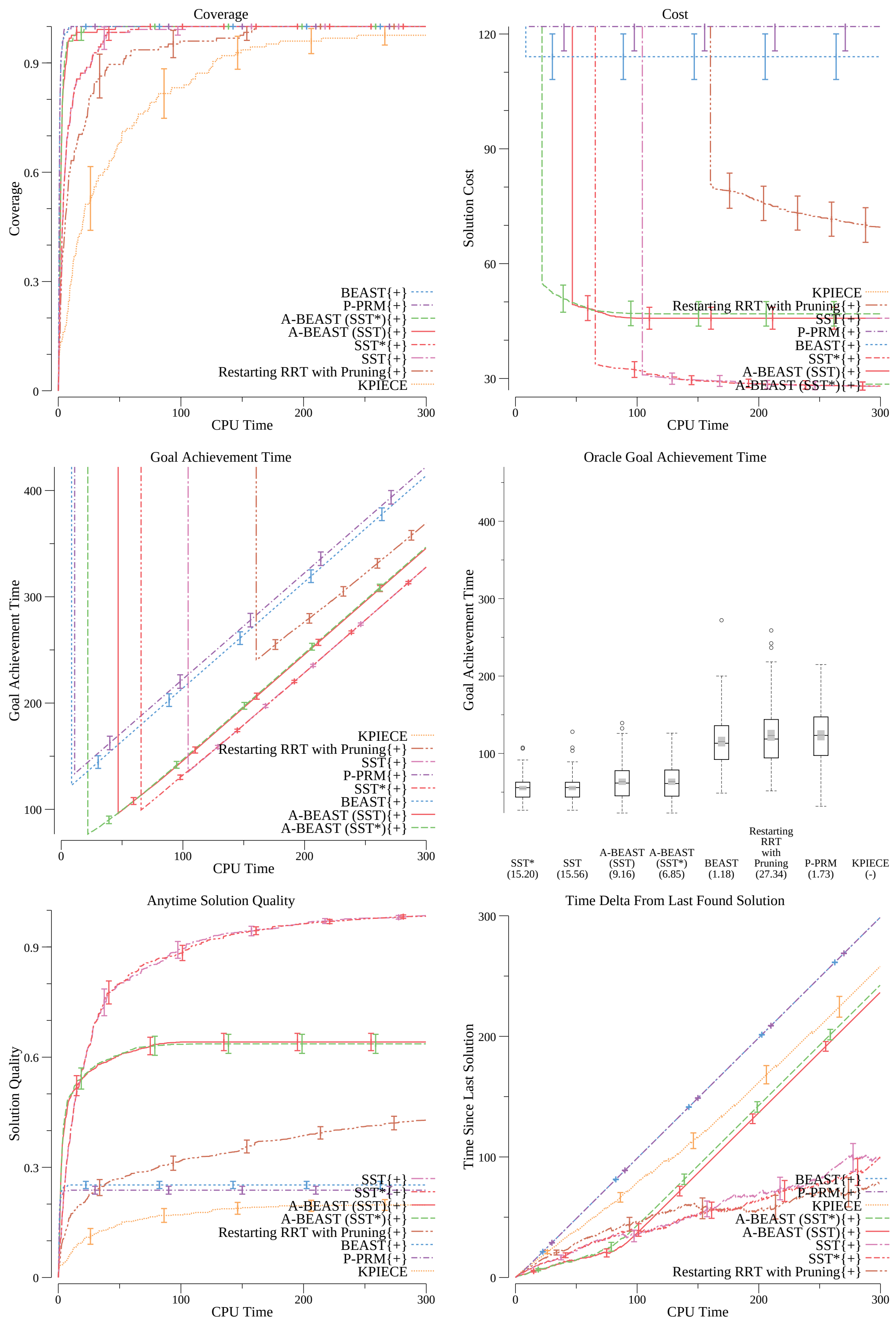


Figure 5-25: Dynamic car results in the intersection workspace.

5.7.3 Hovercraft

Figure 5-26 through Figure 5-30 present the results of A-Beast in the hovercraft domain over each of the workspaces shown in Figure 5-15.

In the hovercraft domain, the dynamics of the vehicle start to be more difficult to control. As a result, more algorithms start being unable to find solutions to all the instances. A-Beast with SST*, Beast, and P-PRM are now the only algorithms to find solutions to all instances within the timeout. These three algorithms, and in some cases A-Beast with SST fight for the position of top performer in terms of time to full coverage. It is interesting to observe that in this domain, the A-Beast variants begin to outperform in some workspaces, the Beast and P-PRM algorithms with respect to this metric (even though they are all pretty close).

In terms of cost, we can see that when an A-Beast variant has full coverage, its solution cost is roughly 1/3 less than that of Beast and P-PRM. In the cases where an SST variant finds full coverage, it is also finds solutions roughly 1/3 less than A-Beast.

The GAT plots over time show that an A-Beast variant of Beast performs best with the exception of in the parking lot workspace where P-PRM performed slightly better. For the oracle GAT plots, A-Beast with SST* is consistently the best.

Similarly in terms of solution quality over time, the A-Beast perform second best to the SST variants with the exception of the forest workspace where the A-Beast algorithms dominate. The time since last solution found plots show a similar story to the previous workspaces. The A-Beast variants perform strongly at the start of planning but are eventually surpassed by another algorithm, typically the SST variants.

5.7.4 Quadrotor

Figure 5-31 through Figure 5-32 present the results of A-Beast in the quadrotor domain over each of the workspaces shown in Figure 5-15.

In the quadrotor domain, we see a similar trend to all previous domains where A-Beast, Beast, and P-PRM vie for the top spot. In the cost plots, the A-Beast variants are best in both workspaces. In the GAT plots over time, the A-Beast algorithms are the strongest performers.

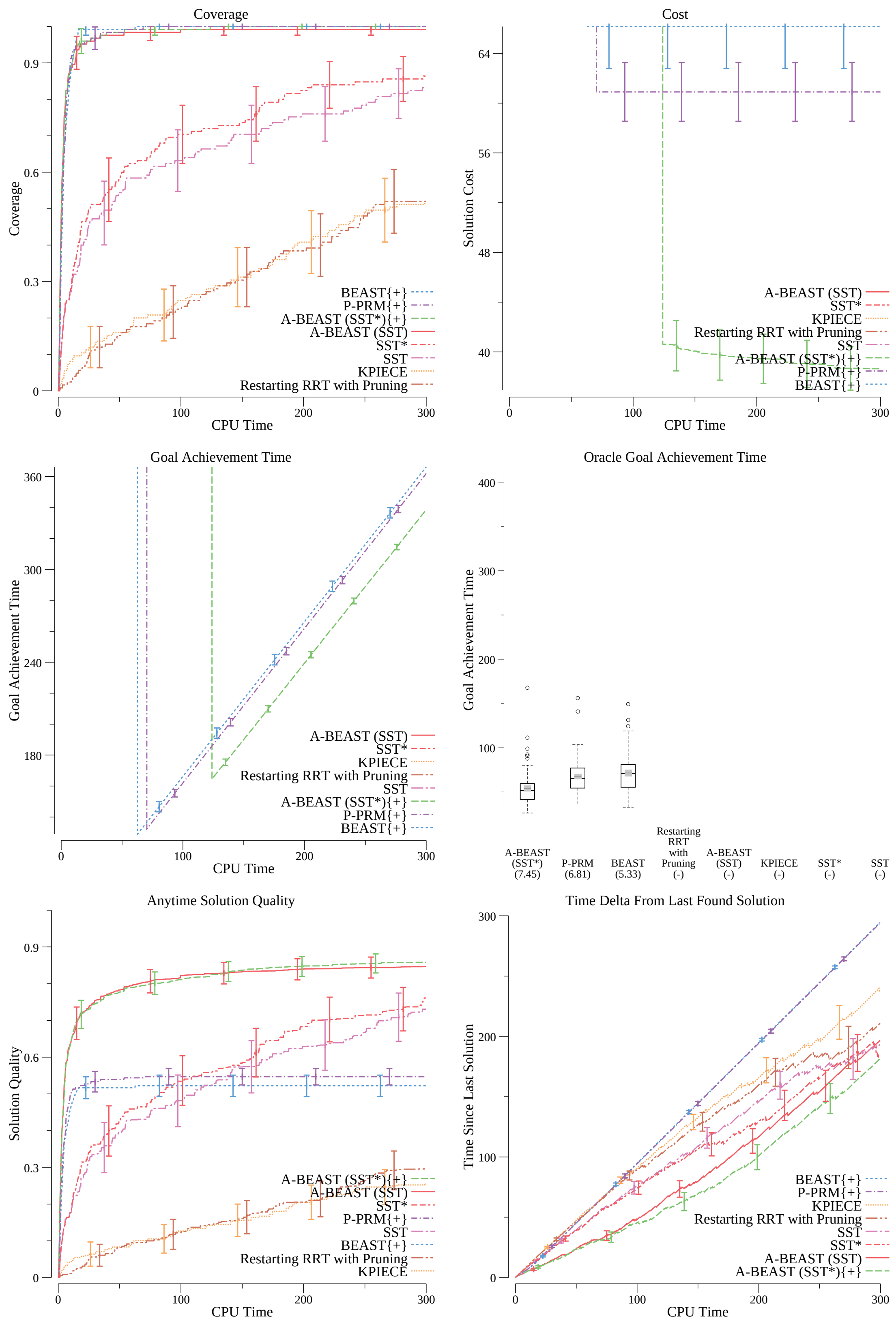


Figure 5-26: Hovercraft results in the forest workspace.

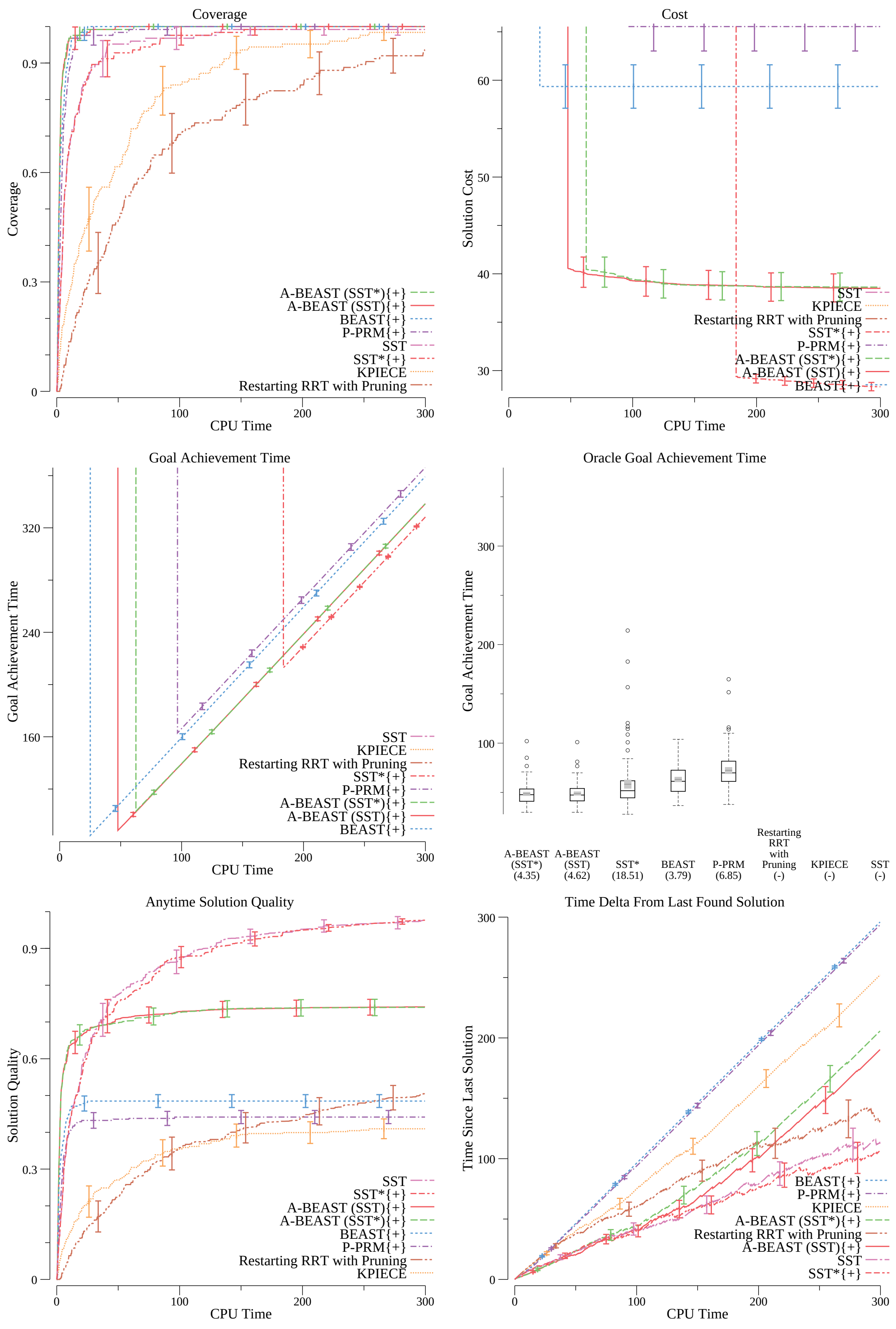


Figure 5-27: Hovercraft results in the single wall workspace.

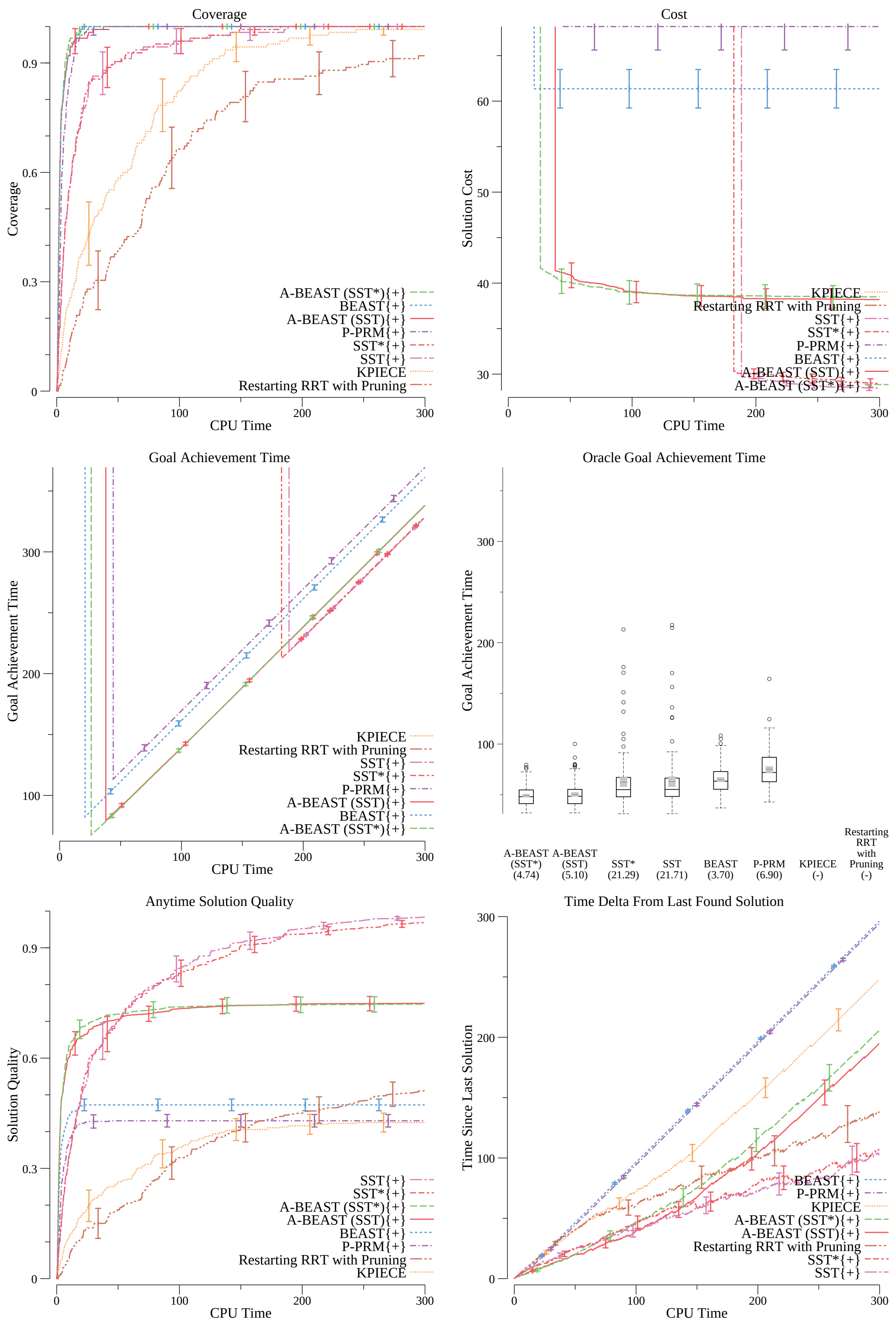


Figure 5-28: Hovercraft results in the ladder workspace.

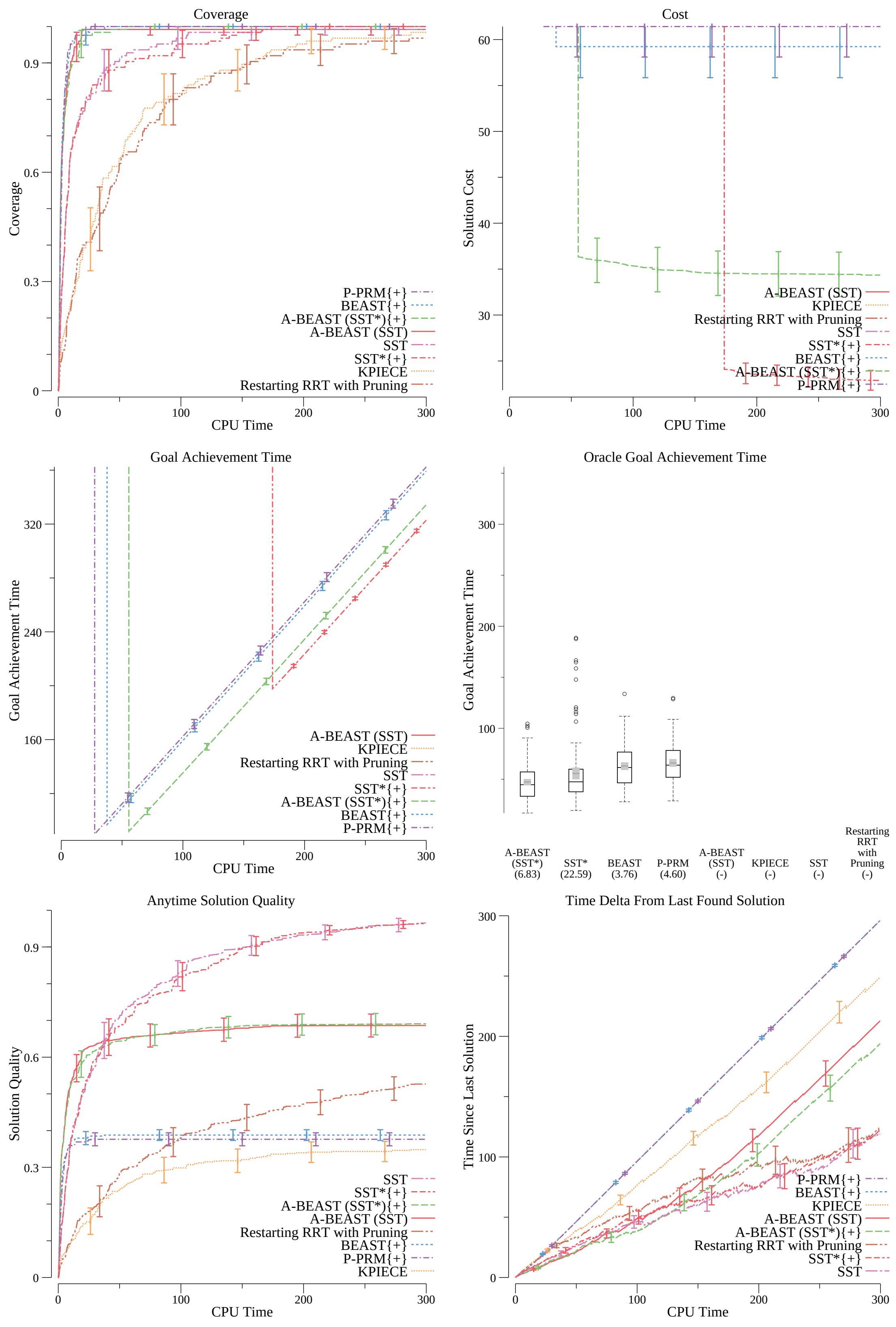
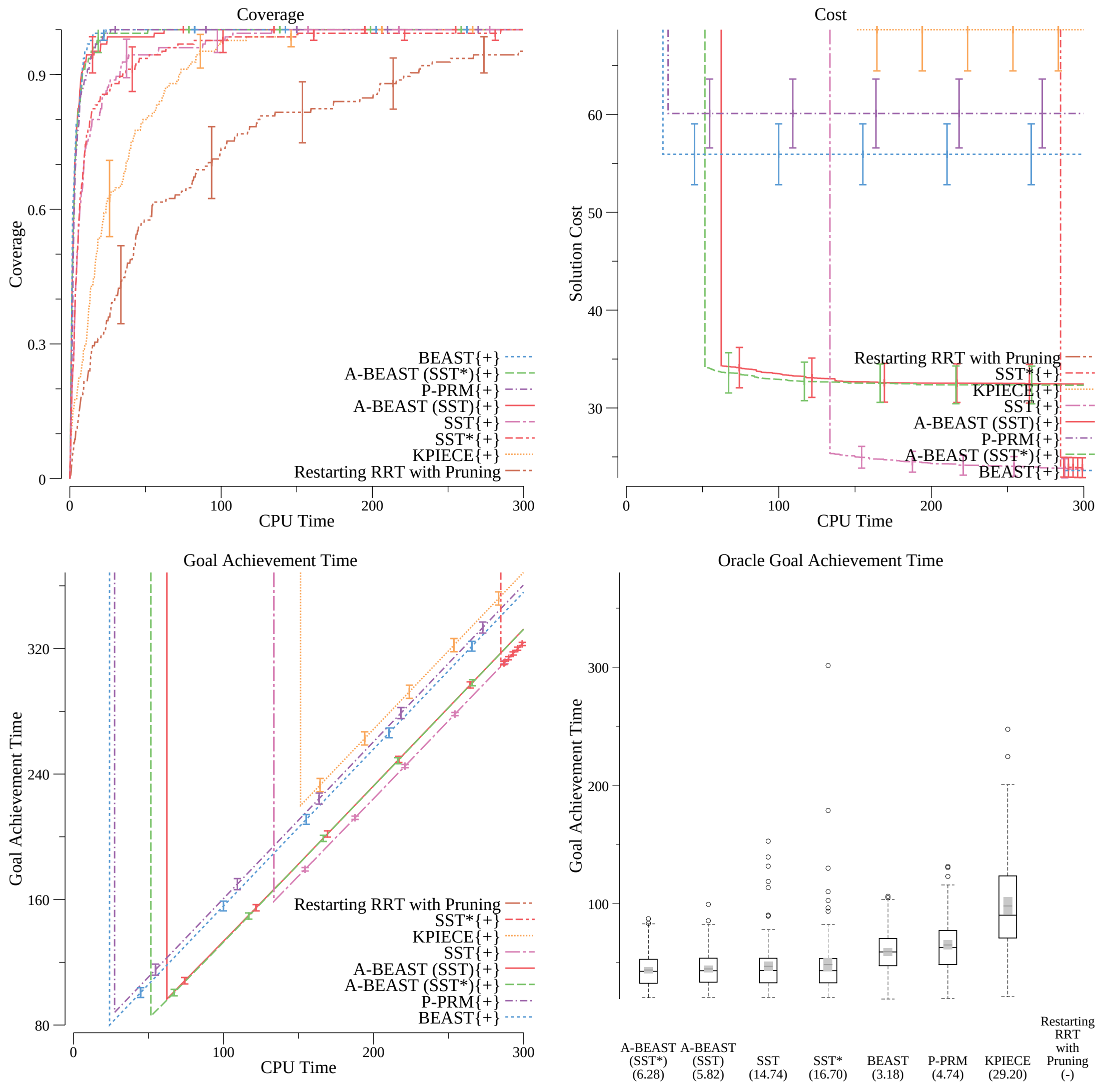


Figure 5-29: Hovercraft results in the parking lot workspace.



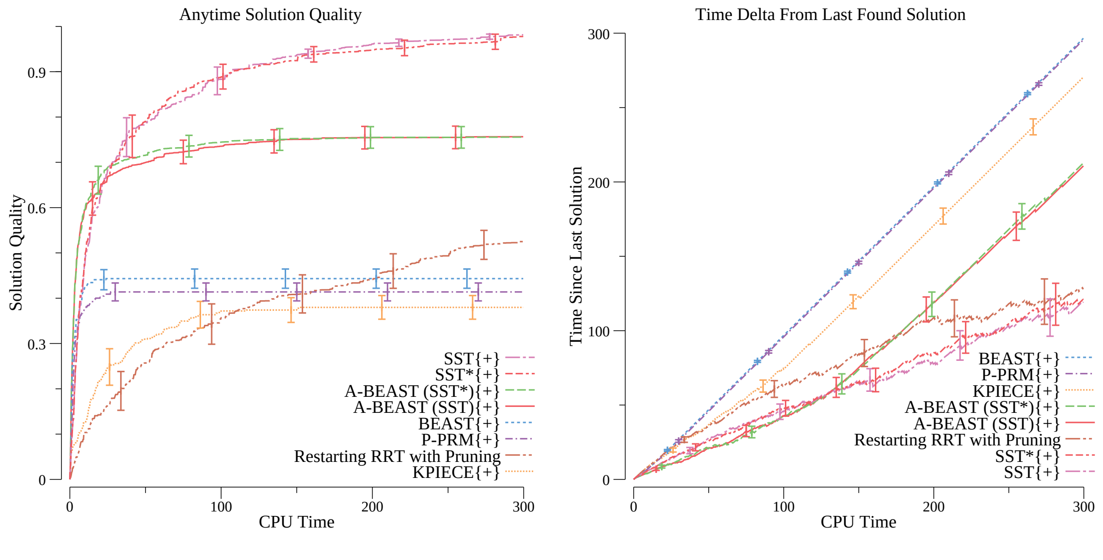


Figure 5-30: Hovercraft results in the intersection workspace.

However, in the GAT oracle plots, the A-Beast are consistently third and fourth best. The top two performers in these two workspaces for this metric switch between Restarting RRT with Pruning, KPIECE, and P-PRM. In terms of anytime solution quality, for the first time, Restarting RRT with Pruning does very well. The next best performers are A-Beast and SST variants. Similarly in the solution time delta plots, it also does the best in the two workspaces. The next best performers are the A-Beast variants.

5.7.5 Blimp

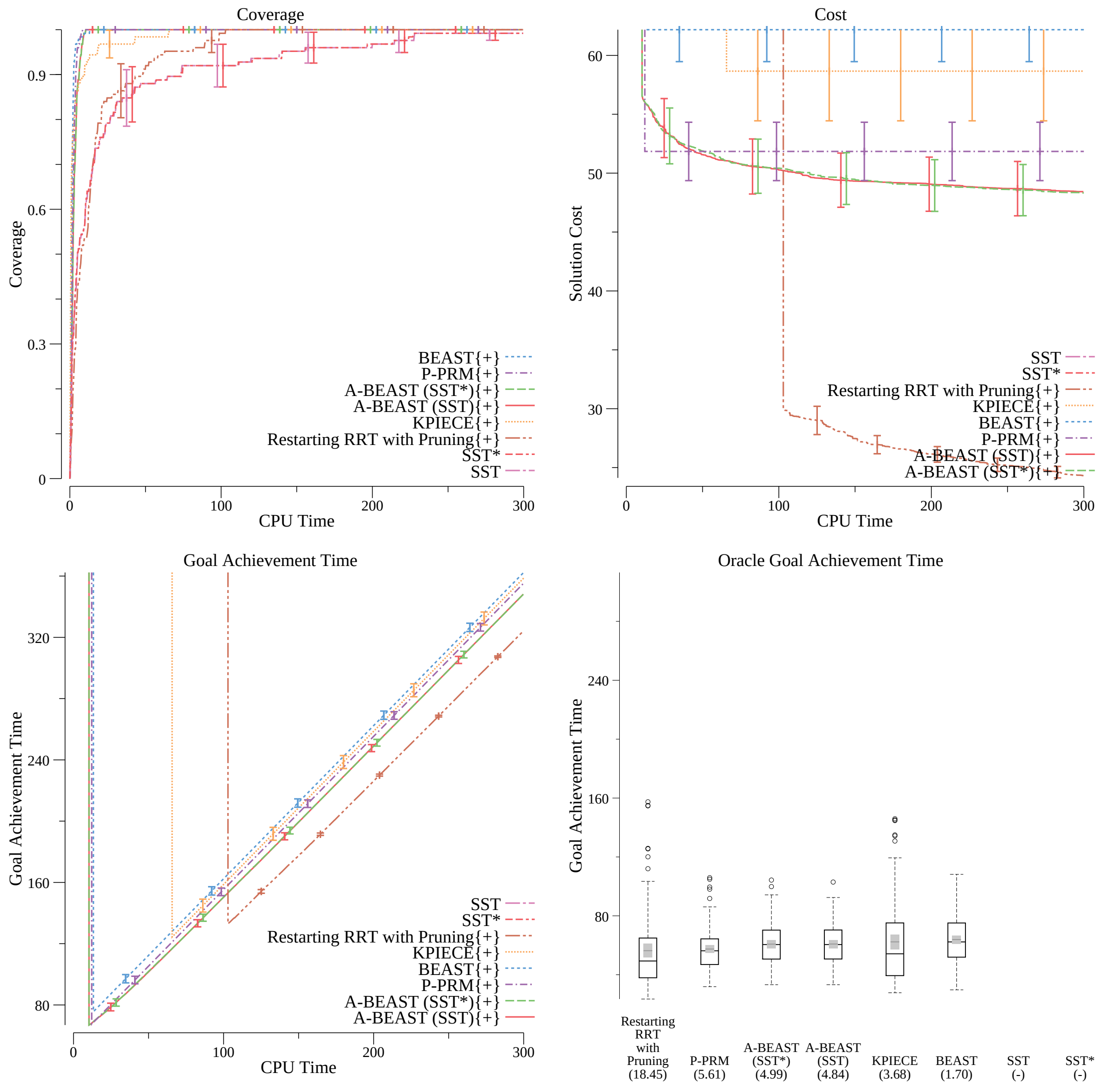
Figure 5-33 through Figure 5-34 present the results of A-Beast in the blimp domain over each of the workspaces shown in Figure 5-15.

The blimp vehicle in the forest workspace has proved very difficult to solve within the timeout. The only algorithm to solve the entire set is Beast. The A-Beast variants, P-PRM, and Restarting RRT with Pruning all approach full coverage but never quite make it during the timeout. As a result the cost plot and GAT plots for the forest domain do not provide additional comparative information. However, in the anytime solution quality and time since last solution found, the A-Beast algorithms dominate.

In the cube world workspace, A-Beast with SST, Beast, and P-PRM are able to find full instance coverage within the timeout. In this workspace, the cost plots show that the solutions found are all very comparable so the only real benefit is the time at which the solutions were found. In the GAT plots, Beast performs best on average over the duration while given an oracle to choose when to stop planning and start executing; P-PRM performs best. On the solution quality plots P-PRM performs best on average but the confidence intervals on the means for the top algorithms are all heavily overlapping. In this workspace, the A-Beast algorithms find solutions much more frequently than all the other algorithms.

5.8 Discussion

One of the major benefits of Beast was that it explicitly focused on areas of the state space that it believes will be easy to traverse while heading toward the goal. It ignored any information it



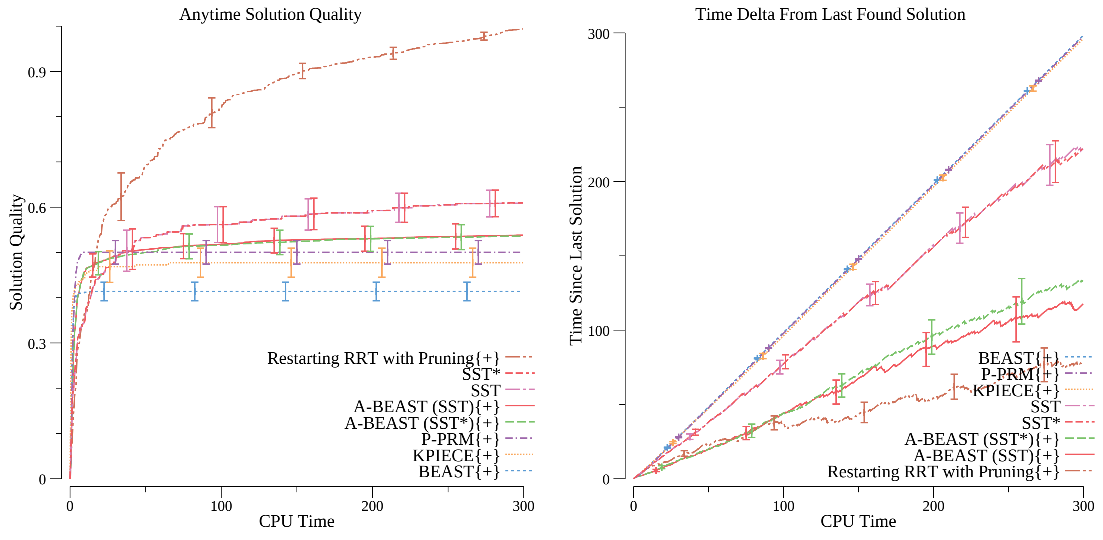


Figure 5-31: Quadrotor results in the forest workspace.

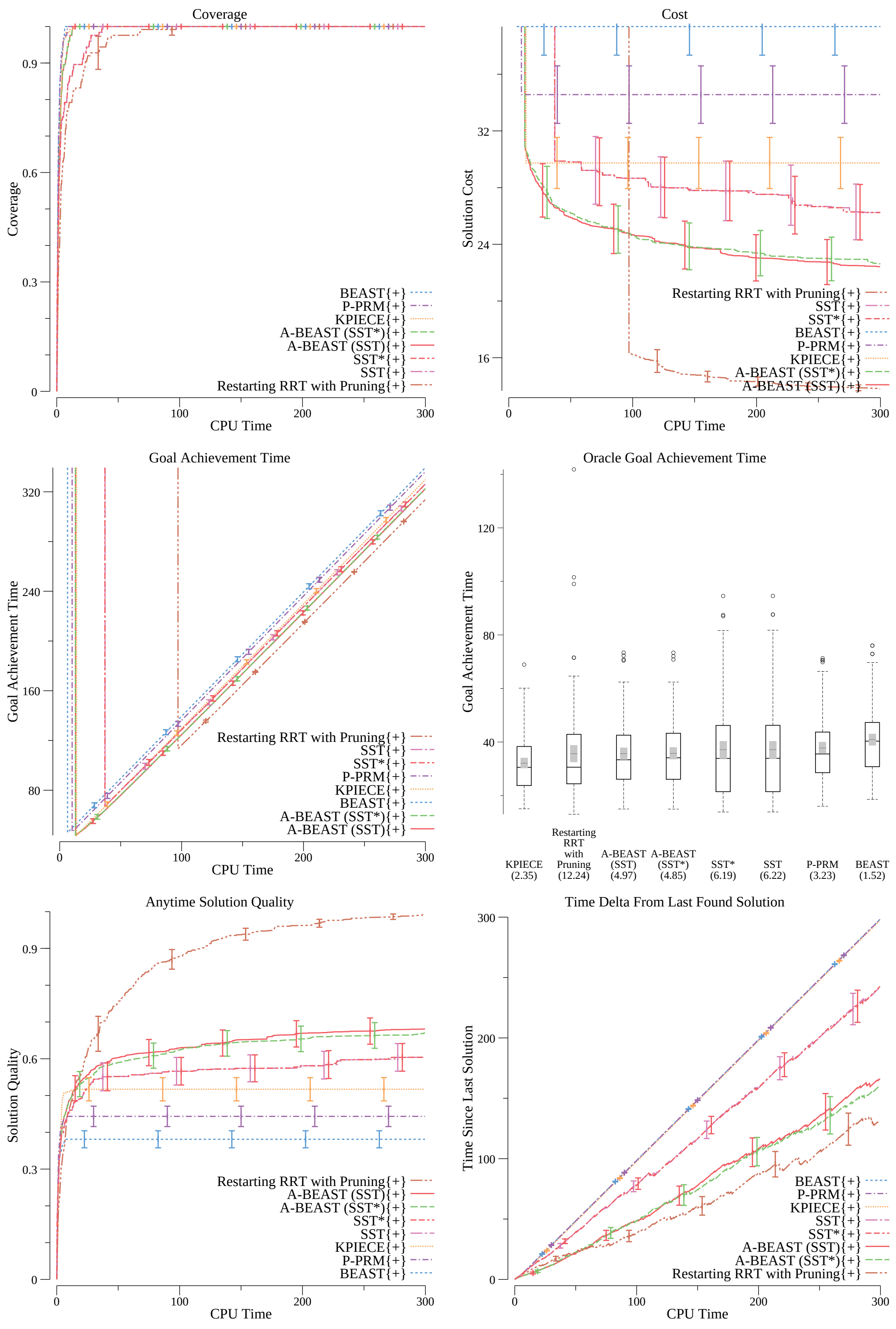


Figure 5-32: Quadrotor results in the cube world workspace.

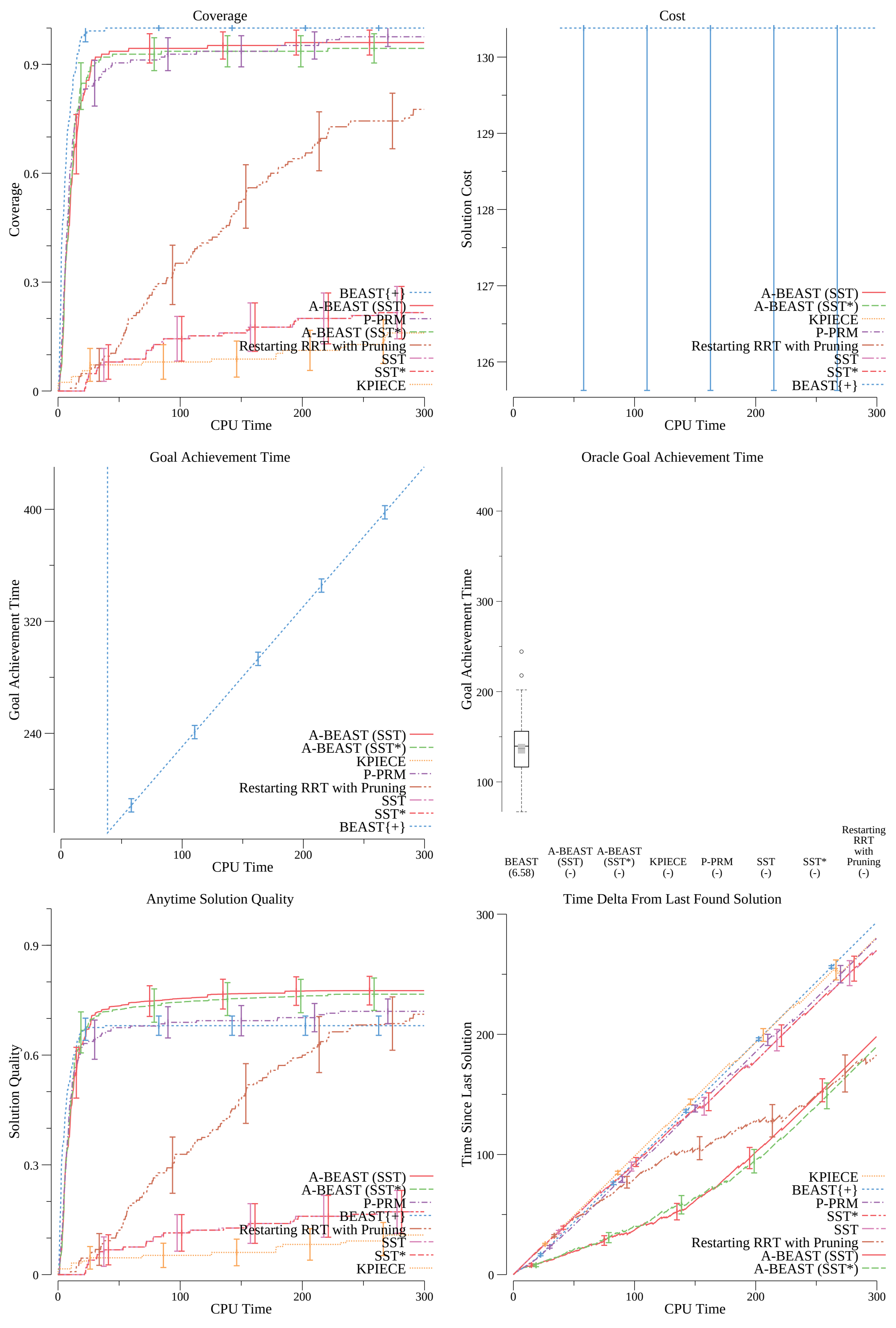


Figure 5-33: Blimp results in the forest workspace.

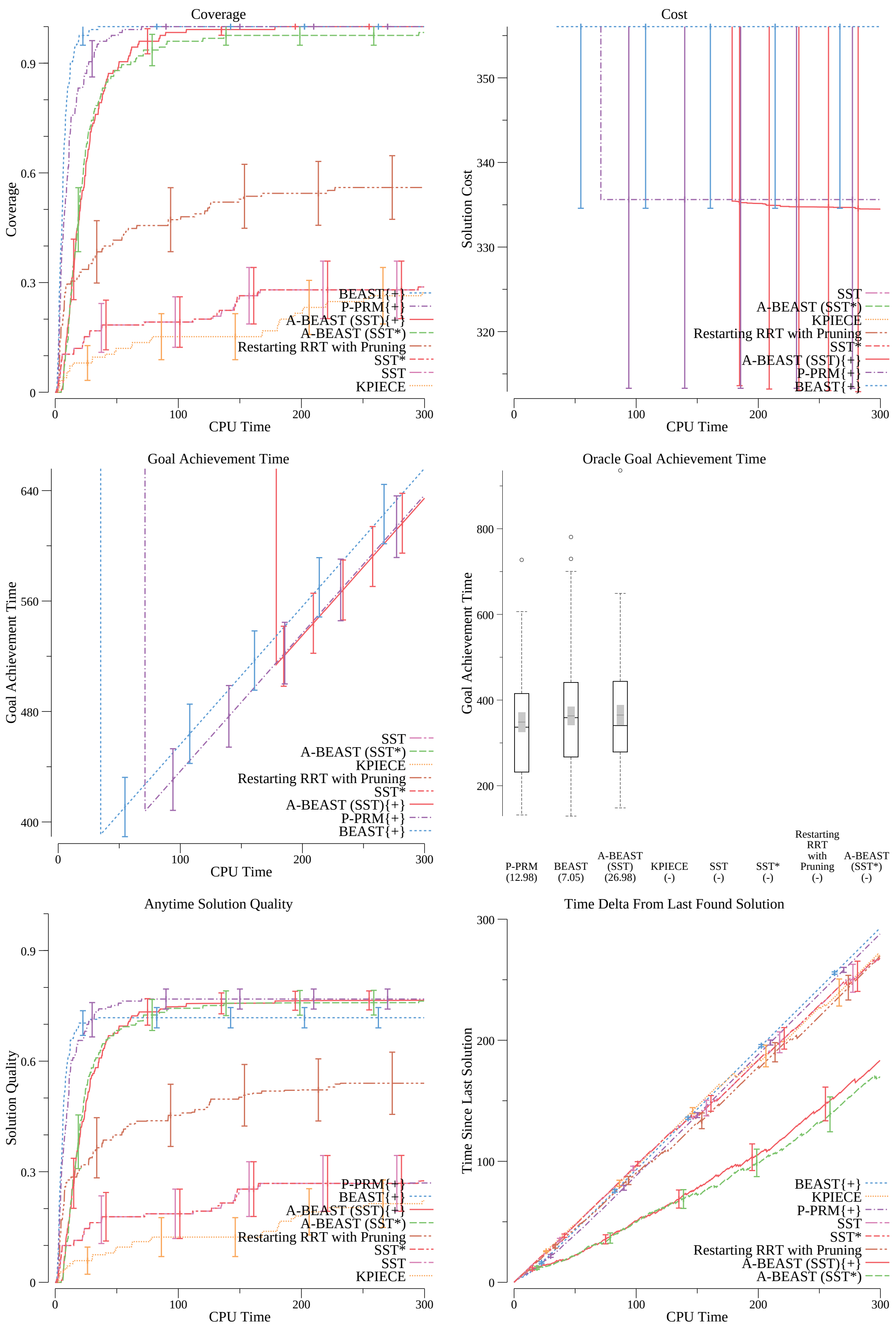


Figure 5-34: Blimp results in the cube world workspace.

could have gathered regarding cost of the solutions it was finding. In A-Beast, we take this extra information into account to find solutions quickly and improve on them as time allows.

As can be seen in the diverse set of experiments performed, A-Beast was not always the best across all metrics on all domains and workspaces. However, it was always a strong performer for all metrics. None of the other algorithms had the same level of consistency offered by A-Beast. It is able to leverage reasoning about easy to construct solutions to find an incumbent quickly and also use cost reasoning to find cheaper solutions, sometimes as much as half as costly as its predecessor, Beast.

In this Chapter we presented two new algorithms, Beast and A-Beast which apply non-classical planning ideas to guide sampling based motion planning. The algorithms are able to provide state of the art performance in this area demonstrating the power of non-classical planning in robotics research. This has continued the trend through this Section of the dissertation illustrating ways that non-classical planning can assist in focusing state space exploration in problems of interest to the robotics community.

CHAPTER 6

Conclusion

Classical planning has developed a powerful set of abstractions and assumptions that enables the study of the underlying characteristics of real world problems. While these abstractions and assumptions are beneficial in academic research, they prove to be a barrier against the direct application of classical planning to real world problems and systems. As such, non-classical planning needs to be developed, constructing the necessary bridges between classical planning's assumptions and the hard truths of operating in the real world. These techniques will remove many of the assumptions that simply do not hold while operating in the real world.

The thesis of this dissertation is the same as its title: robotics needs non-classical planning. The robotics community can benefit from many ideas about high level reasoning from the planning community. Similarly, the planning community can benefit from addressing challenges posed by the applications and domains considered by the robotics community.

To that effect this dissertation has provided two contributions. The first in Part 1 is how uncertainty in the world model can be addressed through a non-classical planning algorithm called hindsight optimization. We considered two realistic sources of uncertainty: temporal uncertainty and open worlds. We presented the OH-wOW algorithm to handle open world uncertainty along with the Tu-Hop algorithm to handle temporal uncertainty (as well as physical location uncertainty). These works were published in an 2013 ICAPS PlanRob paper (Kiesel et al., 2013), in a University of New Hampshire technical report (Kiesel et al., 2012), and in a 2014 ICAPS PlanRob paper (Kiesel & Ruml, 2014).

The second contribution was the application of abstractions and techniques from the heuristic search community to motion planning. In Part 2, we demonstrated the power of abstraction in a complicated task and motion planning problem with temporal constraints. We presented the WAMP problem as well as a framework for finding solutions to this new problem. We then presented an f-biased sampling strategy for sampling base motion planning. Finally, we provided a state of the art motion planning algorithm for minimizing time to first solution and then extended this algorithm to find a stream of improving solutions which provides robust performance across a variety of domains and workspaces. This demonstrates how combining high level discrete reasoning, characteristic of heuristic search, can aid lower level sampling-based motion planning resulting in faster solving times and better solution quality.

This work was published in a 2012 ICAPS paper (Kiesel et al., 2012), a 2012 SoCS extended abstract (Kiesel et al., 2012b), a University of New Hampshire technical report (Kiesel et al., 2012a), and a 2016 ICAPS PlanRob paper (Kiesel & Ruml, 2016).

The sum of these works highlights the benefit and need for non-classical planning in the form of high level reasoning in robotics problems. In many cases, the application of non-classical planning ideas allow agent to focus on the decision of which of the immediately available actions to take next as shown in Section 1. Alternatively, the application of non-classical planning ideas can allow for very focused search throughout very large state spaces. Without the application of these ideas the problems studied in Part 2 of this dissertation could not have been solved as quickly, if at all. The hope of this work is to bring attention to the need for further collaboration between the AI Planning and Robotics communities.

Appendix A

Completeness Proof for Beast

(Kunz & Stilman, 2015)

Given RRT's completeness proof.

Assume there exists an ϵ-hypertube of solutions to the motion planning problem. This ϵ-hypertube will traverse through a sequence of abstract regions called an abstract solution.

Define: A neighboring region is a region that shares at least one boundary with another region.

Assume: In the abstraction, for each abstract region, there exists a set of abstract edges which connect it to at least all neighboring abstract regions.

The set of hyperplanes between each neighboring region and a base region define a convex hyperpolygon.

Lemma 1: The abstract solution is a sequence of neighboring regions, for which there exists a sequence of edges in the abstraction.

By definition of abstract solution this is true.

Lemma 2: If a low level path must be found along an abstract edge, BEAST will eventually find it.

Lemma 3: BEAST will eventually examine the sequence of edges connecting the abstract solution.

Theorem: BEAST will find a solution if a solution exists.

Bibliography

Applegate, D., Bixby, R., Chvatal, V., & Cook, W. (2006). Concorde tsp solver..

Babaian, T., & Schmolze, J. G. (2006). Efficient open world reasoning for planning. *Logical Methods in Computer Science*, *2*(3).

Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., Morris, P., Ong, J., Remolina, E., Smith, T., et al. (2012). EUROPA: A platform for ai planning, scheduling, constraint programming, and optimization..

Bhatia, A., Kavraki, L. E., & Vardi, M. Y. (2010). Sampling-based motion planning with temporal goals.. pp. 2689–2696, Anchorage, Alaska. IEEE.

Bonet, B., & Geffner, H. (2001). GPT: a tool for planning with uncertainty and partial information. In *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, pp. 82–87.

Boutilier, C., Dean, T. L., & Hanks, S. (2011). Decision-theoretic planning: Structural assumptions and computational leverage. *CoRR*, *abs/1105.5460*.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, *25*(2), 163–177.

Burns, E., Benton, J., Ruml, W., Yoon, S., & Do, M. (2012). Anticipatory on-line planning. In *Proceedings of the Twenty-second International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Cervoni, R., Cesta, A., & Oddi, A. (1994). Managing dynamic temporal constraint networks. In *Proceedings of AIPS-94*, pp. 13–18.

Chong, E., Givan, R., & Chang, H. (2000). A framework for simulation-based network control via hindsight optimization. In *IEEE Conference on Decision and Control*.

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., & Thrun, S. (2005). *Principles of robot motion: theory, algorithms, and implementation*. MIT Press.

Cimatti, A., Roveri, M., & Bertoli, P. (2004). Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, *159*(1–2), 127–206.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Dantzig, G., & Ramser, J. (1951). The truck dispatching problem. *Management Science*, *6*(1), 80–91.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial intelligence*, *49*(1), 61–95.

Dechter, R., & Pearl, J. (1988). The optimality of A*. In Kanal, L., & Kumar, V. (Eds.), *Search in Artificial Intelligence*, pp. 166–199. Springer-Verlag.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Dornhege, C., Gissler, M., Teschner, M., & Nebe, B. (2009). Integrating symbolic and geometric planning for mobile manipulation. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Doshi, F. (2009). The infinite partially observable markov decision process. In *Neural Information Processing Systems*, Vol. 22, pp. 477–485.

Dubins, L. E. (1957). On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, *79*, 497–516.

Etzioni, O., & Weld, D. S. (1994). A softbot-based interface to the internet. *Communications of the ACM*, *37*(7), 72–76.

Eyerich, P., Keller, T., & Helmert, M. (2010). High-quality policies for the Canadian travelers problem. In *Third Annual Symposium on Combinatorial Search SoCS-10*.

Frank, B., Stachniss, C., Abdo, N., & Burgard, W. (2011). Using Gaussian process regression for efficient motion planning in environments with deformable objects. In *Automated Action Planning for Autonomous Moblie Robotics*.

Gammell, J. D., Srinivasa, S. S., & Barfoot, T. D. (2014). Informed RRT*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Gammell, J. D., Srinivasa, S. S., & Barfoot, T. D. (2015). Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 3067–3074. IEEE.

Garey, M. R., & Johnson, D. S. (1991). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Ghallab, M., & Laruelle, H. (1994). Representation and control in IxTeT, a temporal planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems AIPS*, pp. 61–67.

Gonzalez, J. P., & Likhachev, M. (2011). Search-based planning with provable suboptimality bounds for continuous state spaces. In *Fourth Annual Symposium on Combinatorial Search*.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*, 267–297.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, *4*(2), 100–107.

Hauser, K., & Zhou, Y. (2015). Asymptotically optimal planning by feasible kinodynamic planning in state-cost space. In *arXiv preprint arXiv:1505.04098*.

Hoffmann, J., & Nebel, B. (2011). The FF planning system: Fast plan generation through heuristic search. *CoRR*, *abs/1106.0675*.

Hsu, D., Latombe, J.-C., & Motwani, R. (1999). Path planning in expansive configuration spaces. *International Journal of Computational Geometry & Applications*, *9*(04n05), 495–512.

Ingrand, F., Chatila, R., Alami, R., & Robert, F. (1996). Prs: A high level supervision and control language for autonomous mobile robots. In *IEEE Conference on Robotics and Automation ICRA*.

Joshi, S., Schermerhorn, P. W., Khardon, R., & Scheutz, M. (2012). Abstract planning for reactive robots.. In *Proceedings of IEEE ICRA*, pp. 4379–4384.

Kaelbling, L. P., & Lozano-Pérez, T. (2011). Hierarchical planning in the now. In *IEEE Conference on Robotics and Automation ICRA*.

Karaman, S., & Frazzoli, E. (2010). Incremental sampling-based algorithms for optimal motion planning. *CoRR*, *abs/1005.0416*.

Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, *30*(7), 846–894.

Kavraki, L. E., Švestka, P., Latombe, J.-C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, *12*(4), 566–580.

Kiesel, S., Burns, E., & Ruml, W. (2012a). Abstraction-guided sampling for motion planning. Tech. rep. UNH-CS-12-01.

Kiesel, S., Burns, E., & Ruml, W. (2012b). Abstraction-guided sampling for motion planning (extended abstract). In *Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Kiesel, S., Burns, E., & Ruml, W. (2015). Achieving goals quickly using real-time search: Experimental results in video games. *Journal of Artificial Intelligence Research*, *54*, 123–158.

Kiesel, S., Burns, E., Ruml, W., Benton, J., & Kreimendahl, F. (2012). Open world planning via hindsight optimization. Tech. rep. UNH-CS-12-03.

Kiesel, S., Burns, E., Ruml, W., Benton, J., & Kreimendahl, F. (2013). Open world planning via hindsight optimization. In *23rd International Conference on Automated Planning and Scheduling: Planning and Robotics Workshop*.

Kiesel, S., Burns, E., Wilt, C., & Ruml, W. (2012). Integrating vehicle routing and motion planning. In *22nd International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Kiesel, S., & Ruml, W. (2014). Planning under temporal uncertainty using hindsight optimization. In *24th International Conference on Automated Planning and Scheduling: Planning and Robotics Workshop*.

Kiesel, S., & Ruml, W. (2016). An anytime Bayesian effort bias for sampling-based motion planning. In *26th International Conference on Automated Planning and Scheduling: Planning and Robotics Workshop*.

Koenig, S., & Likhachev, M. (2002). D* lite.. In *AAAI/IAAI*, pp. 476–483.

Krammer, L., Granzer, W., & Kastner, W. (2011). A new approach for robot motion planning using rapidly-exploring randomized trees. In *Proceedings of the Ninth IEEE International Conference on Industrial Informatics*, pp. 263 –268.

Kuffner, Jr., J. J., & LaValle, S. (2000). RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-00)*, Vol. 2, pp. 995 –1001.

Kuhn, L., Schmidt, T., Price, B., Zhou, R., & Do, M. (2008). Heuristic search for target-value path problem. In *First International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

Kunz, T., & Stilman, M. (2015). Kinodynamic rrts with fixed time step and best-input extension are not probabilistically complete. In *Algorithmic Foundations of Robotics XI*, pp. 233–244. Springer.

Laborie, P., & Ghallab, M. (1995). IxTeT: An integrated approach for plan generation and scheduling. In *INRIA/IEEE Symposium on Emerging Technologies and Factory Automation ETFA*, Vol. 1, pp. 485–495 vol.1.

Ladd, A. M., & Kavraki, L. E. (2005). Motion planning in the presence of drift, underactuation and discrete system changes.. In *Robotics: Science and Systems*, pp. 233–240.

Lau, H. C., Sim, M., & Teo, K. M. (2003). Vehicle routing problem with time windows and a limited number of vehicles. *Europen Journal of Operational Research*, *148*, 559–569.

Lavalle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning. Tech. rep., Department of Computer Science, Iowa State University.

LaValle, S. M. (2006). *Planning Algorithms.* Cambridge University Press, Cambridge, U.K.

LaValle, S. M., & Kuffner, J. J. (2001). Randomized kinodynamic planning. *The International Journal of Robotics Research*, *20*(5), 378–400.

Lavalle, S. M., & Kuffner, Jr., J. J. (2000). Rapidly-exploring random trees: Progress and prospects. In *Proceedings of the Fourth International Workshop on Algorithmic Foundations of Robotics (WAFR-00)*, pp. 293–308.

LaValle, S. M., & Kuffner, Jr., J. J. (2001). Randomized kinodynamic planning. *International Journal of Robotics Research*, *20*, 378–400.

Le, D., & Plaku, E. (2014). Guiding sampling-based tree search for motion planning with dynamics via probabilistic roadmap abstractions. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pp. 212–217. IEEE.

Lemai, S., & Ingrand, F. (2003). Interleaving temporal planning and execution: IxTeT-eXeC. In *Proceedings of the ICAPS Workshop on Plan Execution.*

Levesque, H. (1996). What is planning in the presence of sensing?. In *Proceedings of AAAI.*

Levesque, H. J. (2005). Planning with loops. In *Proceedings of IJCAI.*

Li, Y., Littlefield, Z., & Bekris, K. E. (2015). Sparse methods for efficient asymptotically optimal kinodynamic planning. In *Algorithmic Foundations of Robotics XI*, pp. 263–282. Springer.

Likhachev, M., & Ferguson, D. (2009). Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, *28*(8), 933–945.

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems (NIPS-03)*, *16*.

Likhachev, M., & Stentz, A. (2008). R* search. In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI-08)*.

Lynch, K. M. (1999). Controllability of a planar body with unilateral thrusters. *Automatic Control, IEEE Transactions on*, *44*(6), 1206–1211.

McDermott, D. (1987). A critique of pure reason. *Computational Intelligence*, *3*(1), 151–160.

McVey, C. B., Clements, D. P., Massey, B. C., & Parkes, A. J. (1999). Worldwide aeronautical route planner. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*.

Mercier, L., & van Hentenryck, P. (2007). Performance analysis of online anticipatory algorithms for large multistage stochastic programs. In *Proceedings of IJCAI*.

Meuleau, N., & Smith, D. E. (2003). Optimal limited contingency planning. In *Proceedings of UAI*.

Morris, P., Muscettola, N., & Vidal, T. (2001). Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pp. 494–499.

Nilsson, M., Kvarnström, J., & Doherty, P. (2014). Efficientidc: A faster incremental dynamic controllability algorithm.. In *24th International Conference on Automated Planning and Scheduling (ICAPS-14)*.

Nilsson, N. J. (1969). A mobile automaton: An application of artificial intelligence techniques. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*.

Phillips, M., & Likhachev, M. (2011). Planning in domains with cost function dependent actions. In *Proceedings of the fourth international symposium on combinatorial search (SoCS-11)*.

Plaku, E., Kavraki, L. E., & Vardi, M. Y. (2010). Motion planning with dynamics by a synergistic combination of layers of planning. *Robotics, IEEE Transactions on*, *26*(3), 469–482.

Remolina, E., & Kuipers, B. (2004). Towards a general theory of topological maps. *Artificial Intelligence*, *152*(1), 47–104.

Ross, S., Pineau, J., Paquet, S., & Chaib-draa, B. (2008). Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, *32*, 663–704.

Sanner, S. (2011). Relational dynamic influence diagram language (RDDL): Language description. *NICTA, Australia*, *1*(1), 1.

Savelsbergh, M. (1985). Local search in routing problems with time windows. *Annals of Operations Research*, *4*, 285–305.

Schmidt, T., Kuhn, L., Price, B., de Kleer, J., & Zhou, R. (2009). A depth-first approach to target-value search. In *Proceedings of the Second Symposium on Combinatorial Search*.

Shani, G., & Brafman, R. (2011). Replanning in domains with partial information and sensing actions. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pp. 2021–2026.

Silver, D., & Veness, J. (2010). Monte-carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems 23*, pp. 2164–2172.

Smith, D. E., & Weld, D. S. (1998). Conformant graphplan. In *Proceedings of AAAI*, pp. 889–896.

Song, G., & Amato, N. M. (2001). Using motion planning to study protein folding pathways. In *Proceedings of the Fifth Annual International Conference on Computational Biology*, pp. 287–296.

Sturtevant, N., & Geisberger, R. (2010). A comparison of high-level approaches for speeding up pathfinding. *Proceedings of the Fourth Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-10)*, *1*(1), 76–82.

Sucan, I. A., & Kavraki, L. E. (2009). Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII*, pp. 449–464. Springer.

Sucan, I. A., Moll, M., & Kavraki, L. E. (2012). The open motion planning library. *Robotics & Automation Magazine, IEEE*, *19*(4), 72–82.

Talamadupula, K., Benton, J., Schermerhorn, P., Kambhampati, S., & Scheutz, M. (2010). Integrating a closed world planner with an open world robot: A case study. In *Proceedings of AAAI*.

Thayer, J. (2012). *Faster Optimal and Suboptimal Heuristic Search*. Ph.D. thesis, University of New Hampshire.

Thayer, J., & Ruml, W. (2011). Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the Twenty-second International Joint Conferenec on Artificial Intelligence (IJCAI-11)*.

Thayer, J. T., Benton, J., & Helmert, M. (2012). Better parameter-free anytime search by minimizing time between solutions. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Thayer, J. T., Ruml, W., & Bitton, E. (2008). Fast and loose in bounded suboptimal heuristic search. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*.

Thayer, J. T., & Ruml, W. (2009). Using distance estimates in heuristic search.. In *ICAPS*, pp. 382–385.

Thayer, J. T., & Ruml, W. (2011). Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*, pp. 674–679.

Urmson, C., & Simmons, R. (2003). Approaches for heuristically biasing RRT growth. In *Proceedings of the IEEE/RSJ International Conference on Robotics and Systems (IROS-03)*.

Štěpán Kopřiva, Šišlák, D., Pavlíček, D., & Pěchouček, M. (2010). Iterative accelerated A* path planning. In *Proceedings of the Foty-nineth Conference on Decision and Control*.

Vansteenwegena, P., Souffriaua, W., & Oudheusdena, D. V. (2011). The orienteering problem: A survey. *European Journal of Operational Research*, *209*, 1–10.

Vonásek, V., Faigl, J., Krajník, T., & Přeučil, L. (2009). RRT-path - A guided rapidly exploring random tree. In *Robot Motion and Control*, Vol. 396 of *Lecture Notes in Control and Information Sciences*, pp. 307–316.

Wilt, C. M., & Ruml, W. (2014). Speedy versus greedy search. In *Seventh Annual Symposium on Combinatorial Search*.

Wu, G., Chong, E., & Givan, R. (2002). Burst-level congestion control using hindsight optimization..

Xie, C., van den Berg, J., Patil, S., & Abbeel, P. (2015). Toward asymptotically optimal motion planning for kinodynamic systems using a two-point boundary value problem solver. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4187–4194. IEEE.

Yoon, S. W., Fern, A., & Givan, R. (2007). FF-replan: A baseline for probabilistic planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-07)*.

Yoon, S., Fern, A., Givan, R., & Kambhampati, S. (2008). Probabilistic planning via determinization in hindsight. In *Proceedings of Conference on Artificial Intelligence (AAAI)*.

Yoon, S., Ruml, W., Benton, J., & Do, M. B. (2010). Improving determinization in hindsight for on-line probabilistic planning. In *Proceedings of the Tenth International Conference on Automated Planning and Scheduling (ICAPS-10)*.

Younes, H. L., & Littman, M. L. (2004). PPDDL 1.0: The language for the probabilistic part of IPC-4. In *Proceedings of the International Planning Competition*, p. 46.