# ROBOT MOTION PLANNING USING REAL-TIME HEURISTIC SEARCH

BY

Jarad Cannon

B.S., University of New Hampshire (2010)

THESIS

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

Master of Science

in

Computer Science

December 2011

This thesis has been examined and approved.

Thesis director, Wheeler Ruml,
Assistant Professor of Computer Science

Philip Hatcher,
Professor of Computer Science

Radim Bartoš,
Associate Professor of Computer Science

Michel Charpentier,
Associate Professor of Computer Science

Date

# ACKNOWLEDGMENTS

I would like to sincerely thank Professor Wheeler Ruml for all the help he has given me throughout the entire process of finding a topic, doing the research and writing this thesis. I would also like to thank Kevin Rose, who I have worked with closely over the past year. We worked jointly on a robot simulator framework, as we are both doing theses in the same area. Finally, I would like to thank all my friends and my girlfriend Beth to whom I have ranted to endlessly about my robots.

# CONTENTS

# LIST OF FIGURES

# ABSTRACT


## ROBOT MOTION PLANNING USING REAL-TIME HEURISTIC SEARCH

by

Jarad Cannon
University of New Hampshire, December, 2011
Wheeler Ruml

Autonomous mobile robots must be able to plan quickly and stay reactive to the world around them. Currently, navigating in the presence of dynamic obstacles is a problem that modern techniques struggle to handle in a real-time manner, even when the environment is known. The solutions range from using: 1) sampling-based algorithms which cut down on the shear size of these state spaces, 2) algorithms which quickly try to plan complete paths to the goal (to avoid local minima) and 3) using real-time search techniques designed for static worlds. Each of these methods have fundamental flaws that prevent it from being used in practice.

In this thesis I offer three proposed techniques to help improve planning among dynamic obstacles. First, I present a new partitioned learning technique for splitting the costs estimates used by heuristic search techniques into those caused by the static environment and those caused by the dynamic obstacles in the world. This allows for much more accurate learning. Second, I introduce a novel decaying heuristic technique for generalizing cost-to-go over states of the same pose $(x, y, \theta, v)$ in the world. Third, I show a garbage collection mechanism for removing useless states from our search to cut down on the overall memory usage. Finally, I present a new algorithm called Partitioned Learning Real-time A$^*$. PLRTA$^*$ uses all three of these new enhancements to navigate through worlds with

dynamic obstacles in a real-time manner while handling the complex situations in which other algorithms fail.

I empirically compare our algorithm to other competing algorithms in a number of random instances as well as hand crafted scenarios designed to highlight desirable behavior in specific situations. I show that PLRTA* outperforms the current state-of-the-art algorithms in terms of minimizing cost over a large number of robot motion planning problems, even when planning in fairly confined environments with up to ten dynamic obstacles.

# CHAPTER 1

# INTRODUCTION

## 1.1 Robot Motion Planning

Robot motion planning is an important area of research that has been heavily studied in Robotics. The problem of robot motion planning focuses on finding collision-free paths from a start configuration to a goal configuration. The topic has been studied for many years from a variety of angles in both control theory and artificial intelligence. The specific problem comes in a number of flavors including path planning with only static obstacles, planning with movable obstacles (Van Den Berg et al., 2009), planning in dynamic environments such as opening and closing doors (Bond et al., 2010; Koenig and Likhachev, 2002) and planning with dynamic obstacles which is where there exist other moving obstacles in the world (Kushleyev and Likhachev, 2009; Snape, Guy, and van den Berg, 2010; Phillips and Likhachev, 2011). There are two major approaches to performing the necessary planning to attack these sorts of problems. The first is called offline planning, which is where the entire trajectory of the robot is planned up front and then later executed by the robot. The other approach is called online planning, which interleaves phases of planning and execution until the goal configuration is reached. During each phase of online planning the planning and execution can be performed either sequentially or concurrently.

While the offline method may work in simple static environments and dynamic environments where the changes are deterministic and known beforehand, it could fail catastrophically in the presence of dynamic obstacles. The only way for offline planning methods to accurately deal with dynamic obstacles is when the trajectories of the dynamic obstacles

are known completely during the planning phase. Otherwise, the planning agent would not be able to account for any unforeseen actions the dynamic obstacles may take, and thus, may return plans that collide with the dynamic obstacles. The online method can attempt to remedy this issue by interleaving the planning and execution of the plan. That is, the agent would observe the current world state, generate a plan to reach the goal and then partially execute that plan until the next planning stage begins. This allows the robot to re-plan at every phase in case its plan is no longer valid. For example, if a dynamic obstacle now blocks the path being followed, the robot must re-plan to reach the goal given the new world information. Had this plan been generated by an offline planning algorithm, the robot would not be able to account for the unforeseen changes in the world and may even collide with the dynamic obstacle.

Each planning stage of the online method can be limited to some fixed duration to prevent the agent from spending too little or too much time planning. The duration of the online planning phase can greatly affect both the quality and performance of the robot's motion plan. If the planning stage is too short, the robot will have trouble finding reasonable paths to the goal as it is not given sufficient time to search far enough ahead in the state space to find complete paths to the goal. However, shorter planning and execution phases allow the robot to re-plan more often which can help make it more reactive to the world around it. Longer planning phases can have the reverse effect, allowing more search to be performed which leads to potentially more informed plans, while making it less reactive to changes in the world.

Planning for some fixed amount of time before issuing an action to take is how real-time search (Korf, 1990) works. The planner is given a fixed amount of time to search for the *best* action or series of actions to take. Once that time is up, the robot executes the best action, observes the world state and begins the planning stage once more. It does not need to search all the way to the goal during each planning phase before making this decision, which can, in some cases, allow real-time search to be misled. Because it was not able to find a complete path to the goal, a promising looking path may turn out to be a dead end.

Therefore, real-time algorithms must learn improved heuristic estimates as they explore the state space to avoid becoming stuck in these dead ends. This is a consequence of not planning complete paths, yet it is unavoidable with real-time constraints. Real-time search continues in this plan-act-plan-act progression until the goal configuration is reached. If the goal state cannot be reached, the algorithm will never terminate.

In the following chapters we will describe the problem domain in more detail, review the applicable previous work and explain why the current techniques are insufficient for handling the complexities of the problem. We will then present three new techniques utilizing a decaying heuristic that is specifically designed to allow the planner to efficiently find better collision free paths to the goal configuration while avoiding local minima in the search space.

## 1.2 The Thesis

My thesis is that by using partitioned heuristic and heuristic decay techniques, real-time search algorithms will be able to outperform the current state-of-the-art in solving robot motion planning problems with dynamic obstacles.

## 1.3 Outline

- Chapter 2 will define our robot motion planning problem as well as discuss the framework we have created for running experiments in the new domain.
- Chapter 3 surveys several previously proposed algorithms for this (and similar) problems. We explain why many of them have inherent issues that prevent them from being suitable for this domain.
- Chapter 4 introduces our new techniques of partitioned heuristics and heuristic decay. We also show a garbage collection technique for effectively managing memory in a domain that contains states that can quickly become irrelevant. Finally, we present a new algorithm called PLRTA$^*$ which utilizes all of these techniques.

- Chapter 5 reviews our experimentation process and shows the performance of our new algorithm against some of the best algorithms presented in the previous work. We show empirically that PLRTA* can outperform the current state-of-the-art algorithms in this area.
- Chapter 6 summarizes our work and the results of our technique. We then discuss future directions for our techniques.

# CHAPTER 2

# PROBLEM DOMAIN

## 2.1 Problem Setting

The problem addressed in this thesis is real-time robot motion planning with dynamic obstacles in a known environment. That is, the planning agent knows the static world map in advance and has complete knowledge of its position in the world and the location of its goal. In addition to static obstacles in the world, there are also dynamic obstacles, each of whose pose $(x, y, \theta, v)$ is known for the current time, however, their future trajectories are unknown and must be approximated by some model. Thus, at each planning phase of the algorithm, our planner is given its current pose in the world, along with its current speed and the current locations of the dynamic obstacles as well as their projected trajectories as a series of Gaussian distributions indexed by time as shown in Figure 2-1. The Gaussian distributions represent the probability that at time $t$ in the future the dynamic obstacle will be at a given $(x, y)$ coordinate.

The combination of planning with time and dynamic obstacles with uncertain trajectories makes this a very difficult problem. If this problem were attempted by planning in a space with too few dimensions, i.e. 2D grid world planning, the planner may fail to find a solution even if one exists, as shown in the Figure 2-2. This is because the lack of a representation of time prevents the planner from recognizing that in the future the car blocking its path will likely exit the hallway and allow it to then travel to the goal. If planning in a high dimensional space, including time, with no bound on the amount of time the planner spends on each planning phase, it can take an unbounded amount of time to

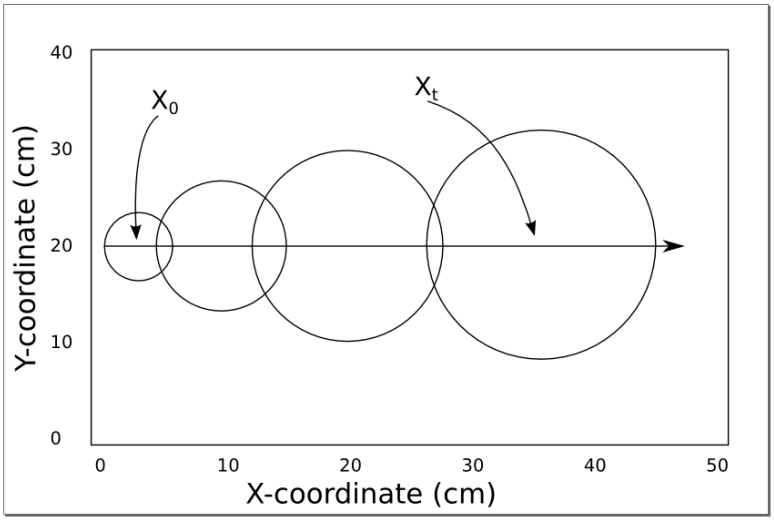


Figure 2-1: The Gaussian distributions indexed by time for the future location of a dynamic obstacle.

find the next action to take, which in some cases can be just as bad as not finding a solution at all. For example, if an object is hurtling towards the agent, it should not spend a long time finding the optimal way to avoid it, we simply need to take any action which will get us out of harm's way. These unbounded planning times and their potentially undesirable results are why real-time search algorithms are necessary for this type of problem. Our real-time search approach searches in a high dimensional space, taking time into account, while restricting itself to a bounded amount of search in each planning phase. At the end of each planning phase, the best action to take is returned. In the following sections, we describe this method in more detail.

### 2.1.1 Problem Specification

A robot path planning problem $P$ in this domain is defined as $P = \{S, s_{start}, g, A, \alpha, O, D, T_p, T_a, c\}$, where:

- $S$ is the state space where $s \in S$ is represented as a 5-tuple $s = (x, y, \theta, v, t)$ of $x$ and $y$ location, heading, speed, and time.

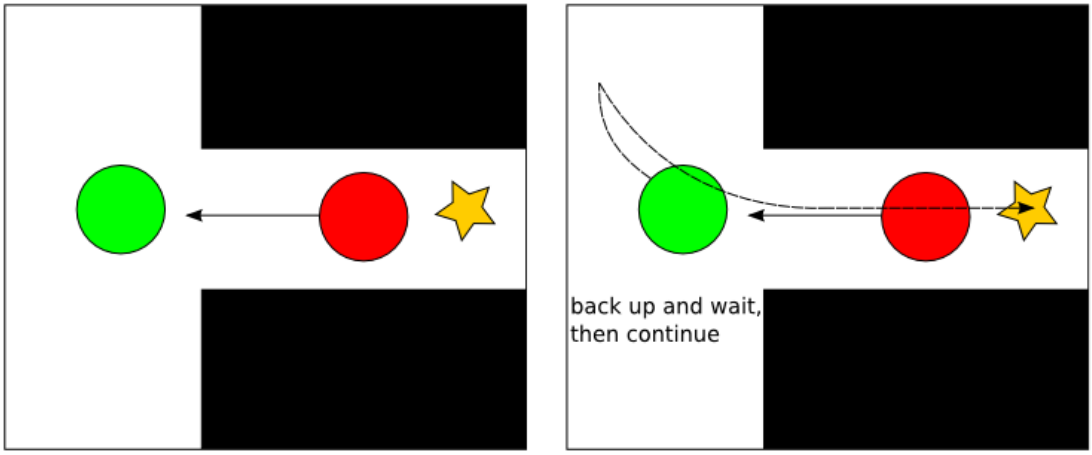


Figure 2-2: Left: 2D planner finding no solution. Right: Solution when accounting for time.

- $s_{start}$ is our starting state.
- $g$ is our agent's goal pose as a 4-tuple $g = (x, y, \theta, v)$, that is, the goal time is not defined and can be any time.
- $A$ is the set of all possible actions, that is, the motion primitives available for the agent to execute, respecting its dynamics.
- $\alpha$ is a function $\alpha : S \rightarrow A$, such that $\forall s \in S, \alpha(s) = A_{fs}$ where $A_{fs} \subseteq A$ is the set of dynamically feasible actions such that $\forall a, a \in A_{fs}$, the action $a$ can be executed given the input state $s$. This distinction is necessary because it is possible that not all actions in $A$ are dynamically feasible for any given state $s$. For example, an agent cannot execute an action to move in reverse at maximum speed if it is currently traveling forward at maximum speed.
- $O$ is the set of static obstacles represented as a matrix of Boolean values, identifying whether a $x, y$ location in the world is blocked or not.
- $D$ is the set of dynamic obstacles represented as a series of Gaussian distributions indexed by time.

- $T_p$ is the static duration of each planning phase.
- $T_a$ is the static duration of each action where $T_a \geq T_p$. All actions have the same duration.
- $c$ is the cost function to be minimized, which is outlined in section 2.1.5.

### 2.1.2 Input and Constraints

During initialization the planner is given $\{A, \alpha, O, T_P, T_a\}$. It is then given the following at the beginning of each planning phase:

- The agent's current state $s \in S$ (initially $s_{start}$).
- The agent's current goal state $g$.
- The projected trajectories of all the dynamic obstacles $D$ out to the current maximum time bound $T_b^{max}$, that is, the time out to which the opponent model can reasonably predict where the dynamic obstacle may be.

The planner does not know, however, the goals of the other dynamic obstacles as well as information about what actions they are going to take. Nor may the planner assume that they are running the same planning algorithm, as the dynamic obstacles in the world may be a mix of intelligent and "dumb" agents. This lack of knowledge about the planning algorithms behind the dynamic obstacles is important as some of the previous work (Phillips and Likhachev, 2011; Snape, Guy, and van den Berg, 2010) makes the assumption of either knowing the dynamic obstacle's intentions or knowing they are acting according to the same algorithm.

### 2.1.3 Output

At the end of each planning phase, the planner's estimate of the best action $a \in \alpha(s_{start})$ to take is returned. That is, before $T_p$ has been exceeded, the planner will return action

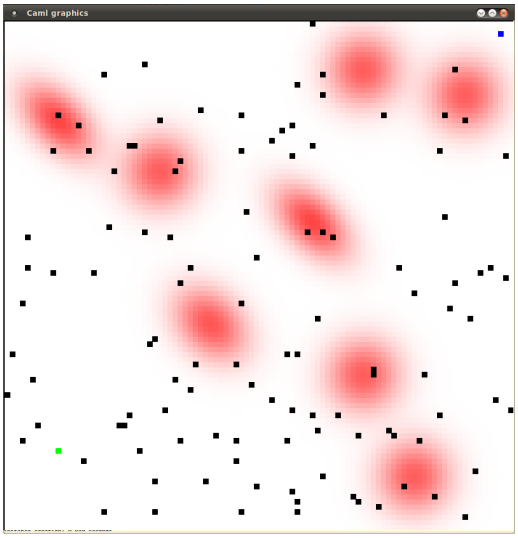


Figure 2-3: Dynamic Obstacles shown as the red high cost areas at a given time-step. The black dots represent static obstacles in the world.

to execute that it estimates will minimize the cost function. Then the world state is once again observed and another planning stage is initiated.

### 2.1.4 Representation of Dynamic Obstacles

We have chosen to represent dynamic obstacles as Gaussians indexed by time in the same manner as Kushleyev and Likhachev (2009). This intuitively seems to be a good way to model the inaccuracy of predicting future positions of each opponent as well as the noise inherent in sampling their locations. The highest points reside at the center of the distribution which represents most strongly where we believe the opponent to be, yet there still exist high cost areas as we deviate from the center to show the uncertainty about where the opponent is precisely. Figure 2-3 shows a representation of our dynamic obstacles frozen in time. As time progresses in the search, however, these distributions will begin to spread out and their centers shift as our opponent model predicts. This represents the growing uncertainty in the future about what these obstacles may do.

We calculate the probability of a collision with a dynamic obstacle given a specific time

and location we first get the set of Gaussian distributions representing the location of the $n$ dynamic obstacles at the specified time. The probability of a collision is then calculated as:

$$P(col) = 1 - \prod_{i=0}^{n}(1 - P_i(col))$$

where $P_i(col)$ is the probability of collision according the the $i^{th}$ Gaussian distribution. This is the same technique used in the work of Kushleyev and Likhachev (2009).

### 2.1.5 Cost Function

The cost function for this problem is as follows: $C_{col} = 1000$ is the cost of a collision, $C_{act} = 0$ is the cost of sitting on the robot's goal configuration and $C_{act} = 5$ is the cost incurred whenever acting outside of its goal configuration. It was set up in this way to discourage colliding with dynamic obstacles, while making it lucrative to reach the goal by incurring no cost. This cost function is to be minimized at each planning phase, as cost is incurred at each time step.. The cost of acting is incurred at every time step regardless of whether or not the agent is moving or not, meaning it still has a cost of 1 to sit still unless it is on its goal configuration.

## 2.2 Simulator

We have created a testing environment shown in Figure 2-4 to carry out our experiments. The existing simulators were not used as they did not provide all the functionality necessary to run and evaluate our experiments. We previously tried using the Player/Stage environment, but ended up using it solely as a graphical front end. We still needed to compute collisions and movements to be able track statistics about collisions. Therefore, we decided to build our own simulator to fit the needs of our problem domain and to be flexible, robust, and modular enough to facilitate rapid changes and new features. We wanted our test framework to be able to span multiple machines, as to offload the heavy planning computations onto their own machines while the simulator could run on a central machine. Shown

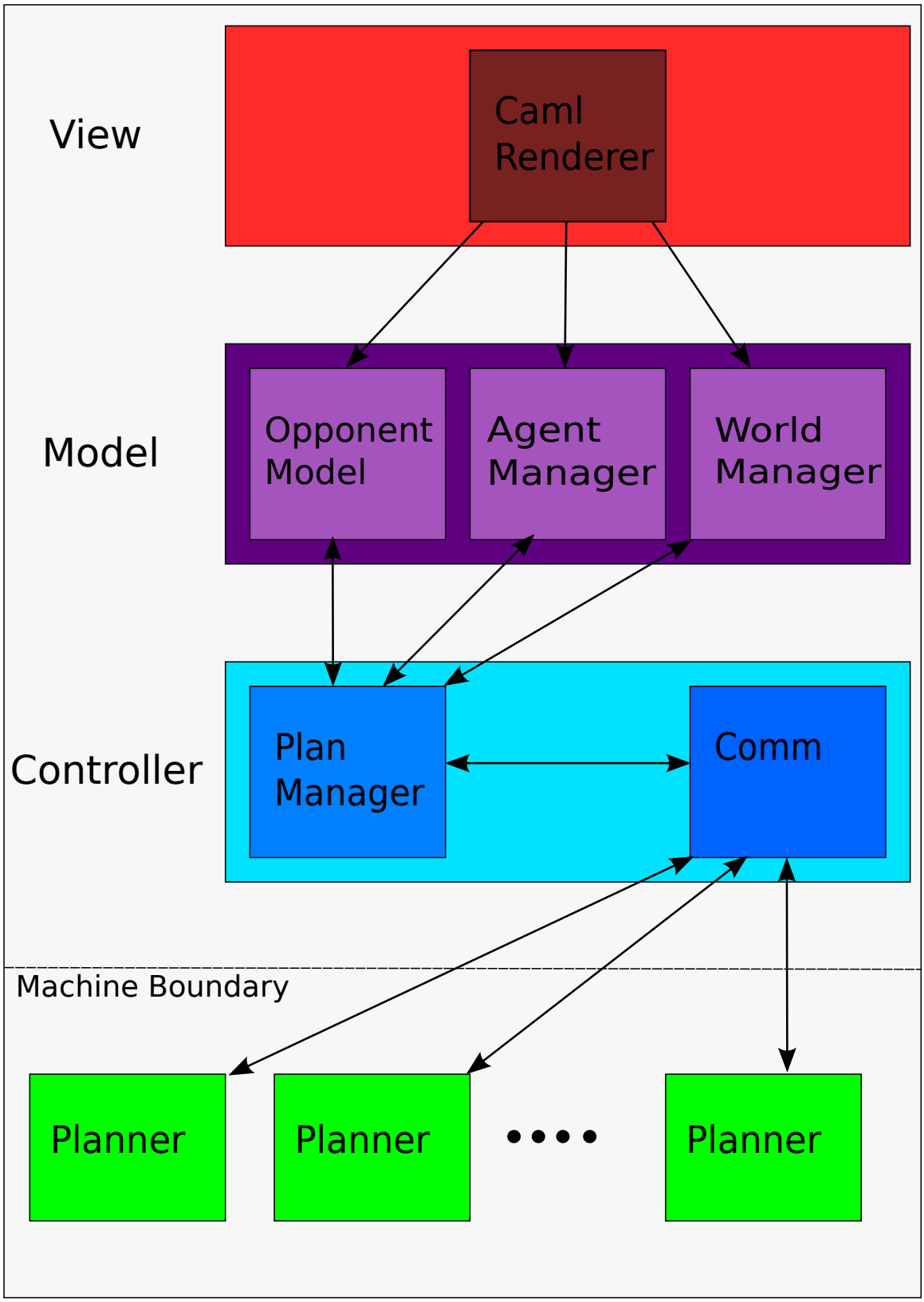


Figure 2-4: Simulator Architecture

in the figure is the physical machine boundary. This shows that the planning algorithms can operate on remote machines, all communicating with the central simulator.

The plan manager is at the heart of the simulator. Its responsibility is to be the mediator between the simulator's model of the experiment and the individual planners. It is tasked with reading in the problem definition and spawning the planners on remote machines to maximize the CPU time allotted per planner. It communicates with these planners through the comm module. The plan manager is responsible for retrieving the best action to take from each planner and translating the planner's action to low level actions that the simulator understands.

In the context of the problem definition, the plan manager provides the inputs to our planning algorithm and handles the planner's output. These inputs and outputs are translated from/to the model of our experiment. The plan manager feeds the planning algorithm what the state of the world will look like one $T_a$ step in the future. Once $T_p$ has expired, the best action is output by the planners and the renderer module begins to execute these actions, while the plan manager concurrently interprets what the world will look like at the beginning of the next planning phase given the actions the planners have just output and forwards this information to the planners as their new current states. Therefore, at all times, with the exception of the first planning phase, the planners are planning for start states that represent the location of their agent one full $T_p$ time step ahead, as is shown in figure 2-5.

The renderer is responsible for drawing the simulation on the screen by executing the actions stored in the agent model. These actions are updated after each planning phase of the agent by the plan manager and thus must constantly be updated and animated. The renderer module is also responsible for doing collision checking. The actual drawing is an optional flag to the simulator, so we can perform complete experiments with collision detection even on headless compute servers.

The plan manager and the renderer run concurrently in separate threads which much each access our model of the simulation. Therefore, we needed to employ synchronous

techniques to ensure the integrity of the data being modified and read from each of the models.

### 2.2.1 Features

Our robot simulator has a number of features which make it useful for running our experiments in a number of conditions:

- Each experiment is driven by configuration files which can control everything necessary. The configuration files can be used to change:
  - Simulator properties such as which map to use, size and resolution, frame rate and colors.
  - Planning properties such as how many planning iterations the experiment will take, the $T_p$ and $T_a$ parameters, as well as $C_{col}$ and $C_{act}$.
  - Agent specific settings such as name, dimensions, algorithm, host to run the planner on, the motion primitive set to use, size and color, start and goal locations.
- Text based communication protocol for interacting with the planners. This means the planners may be written in any language that supports standard I/O.
- Easy communication for planners. Simply read in state descriptions on stdin and output actions on stdout. Logging is done via stderr.
- Supports graphical models as well as motion models of different robot types.
- Supports multiple dynamic obstacles who may run their own algorithm or use hand-traced paths.
- Can be run without graphics while still performing all necessary collision detection.

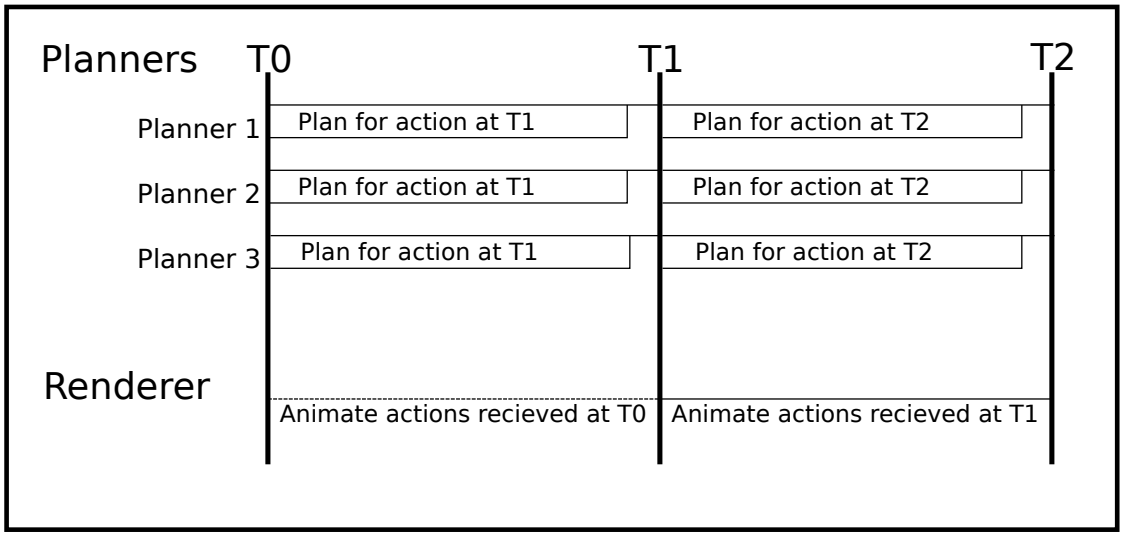


Figure 2-5: Flow of a simulation.

- Logs statistics on the simulation side such as number of collisions and actual cost incurred for each agent. Also collects statistics from each planner, such as nodes expanded, expected cost and other metrics.

## 2.3 An Example

An experiment starts by calling the simulator with the name of a configuration file. From this the simulator dynamically creates the experiment environment and spawns the necessary planners on whatever machine they were configured to run on using password-less SSH. The plan manager then sends the initialization message out to all of the planners, giving to them the same information shown in section 2.1.2. At this point, the plan manager blocks waiting for all the planners to respond to tell the simulator they are ready.

At this point the simulator transitions to its main loop where it tells the agents what the state of the world will be at the beginning of the next planning iteration and awaits the return of their best action to take. Figure 2-5 shows the typical flow of this process. It is shown working with real-time planners who are respecting their $T_p$ bound. Once the actions have been received, the renderer begins animating them while the plan manager

concurrently deciphers what the next state of the work will look like and sends this out to the planners to start the cycle over.If a real-time algorithm ever exceeds their alloted $T_p$ and does not return an action in time, the experiment is terminated and considered a failure.

The simulation continues in this way, constantly planning for the next action to take, while the previous returned action is executed. If there is ever a collision detected, the simulator stops the bots and throws out the action the planners involved in the collision return at the next time step, as they are no longer feasible given the collision. The planners are then sent their new starting state to begin planning for when the next round of state messages are sent out.

The simulator also is equipped to deal with non-real-time algorithms. The case when a non-real-time algorithm misses the $T_p$ deadline to return an action is handled differently. If the planner has been designated to run a non-real-time algorithm, they are instead suddenly stopped in their tracks and interrupted when the action is not returned in time, once the next set of state messages are issued to the planners. More clearly, assume there is a planner running a non-real-time algorithm. It has been given its state for the next time step of the simulator. Assume further that once $T_p$ has expired, the planner has not returned an action to take. If the agent is currently moving, it is allowed to complete the action it had chosen, but is instantaneously stopped once it is completed. Because they were moving, the state the planner is currently planning for is no longer valid. As such, we interrupt the algorithm and supply them with their new state in which they are stopped.

There is no penalty for being stopped like this, however, it may cause the planner to be left in an undesirable position. This only happens if the agent is both non-real-time and moving. This is because if they are not moving, the physical pose in which they are planning for currently when they failed to return an action in time will still be their physical pose when they return the action to take. Therefore, if they are stopped when they do not return an action in time, they are not interrupted and are allowed to continue planning.

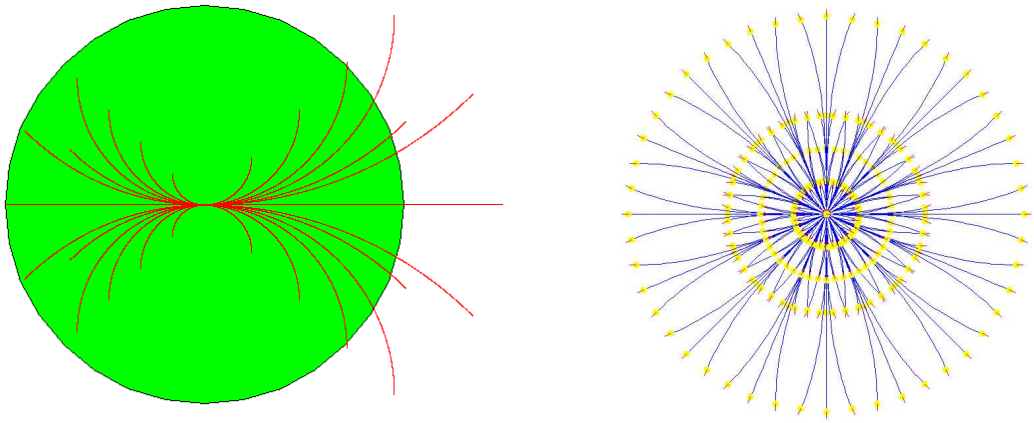

Figure 2-6: **Left:** A sample of the motion primitive available to our agent **Right:** All possible motion primitives for our differential drive bot at sixteen different starting headings.

## 2.4 Motion Model

Kevin Rose designed the motion model, but it requires just a brief explanation. More information about it can be found in his Masters thesis (Rose, 2011).

Our motion model is flexible in that it is treated as a black box from the perspective of our planners and the simulator itself. If one were to come up with a new model of motion and could describe it in our simple format, the simulator and planners alike would not skip a beat. For all of our examples shown in the thesis, we use a differential drive motion model which features four different speeds: two forward speeds, stopped and a reverse speed.

As discussed earlier, we have a notion of dynamically feasible motions given the current pose of the agent. A sample of our motion model for a stopped state is shown in figure 2-6. This set of actions is different than those available to the agent while executing a fast moving forward action. That is, if the agent is currently moving at maximum speed, it is unable to execute the reverse action shown in the figure. This is because the reverse action is not dynamically feasible for the current state of the agent.

## 2.5 Heuristic Search

The key to searching efficiently using heuristic search is to use an ordering function which arranges the search nodes in such a way that those deemed most promising to lead to a goal while minimizing their objective function are explored before those deemed less promising. The function used to estimate the cost of the best solution while including a given node in the solution path is defined as:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost incurred from the start node to node $n$, and $h(n)$ is the estimated cost-to-go from the node $n$ a the goal node. A node's $g$ value is calculated as his parent's $g$ value plus the expected cost of moving from the parent to the child. The root node has a $g$ of 0.

The expected cost $C$ of taking an action in our domain is computed as follows, assuming $P_{col}$ is the probability of colliding with an obstacle (static or dynamic):

$$C = P_{col} * C_{col} + C_{act}$$

$P_{col}$ for a cell containing a static obstacle is always 1 as we know with perfect accuracy where all the static obstacles are. The $P_{col}$ for a all other cells is value given by summing the formula for detecting the probability of a collision as shown in section 2.1.4 over all the points the motion primitive passes through.

We are searching over an implicitly defined graph, that can be generated on demand by using the starting state and the motion primitives available to it. Applying the motion primitives to a given state will generate what are known as its predecessors in the graph.

To maintain the search nodes in the order of best $f$ value, we store them in a priority queue implemented as a min heap. This priority queue is referred to as the openlist. It is sorted on minimum $f$ value nodes. Nodes which have already been explored in the search are added to a hashtable called the closedlist for quick duplicate checking.

### 2.5.1 Inadmissible $g$ values

An interesting thing to note is that the $g$ values in our domain are inadmissible whenever dynamic obstacles are present. This is because of the way we represent the dynamic obstacles. Each opponent is treated as a Gaussian distribution through time, which represents the uncertainty about its current and future location. We showed previously that the cost of a given edge is $C = P_{col} * C_{col} + C_{act}$. This means that the cost associated with an action can vary from $C_{act}$ (when $P_{col} = 0$) to the maximum possible value of $P_{col} * C_{col} + C_{act}$. This value is almost always going to be less than $C_{col}$, however, since we calculate the probability of colliding by looking up the $(x, y)$ locations covered by our motion primitive and sum the probabilities, as provided by our Gaussian distributions representing the dynamic obstacles. So unless we accurately predict where the dynamic obstacle will be and our motion primitive completely covers the whole Gaussian distribution, the probability of collision will not sum to 1. This means evaluating nodes which result in a collision, therefore having an actual cost of $C_{col}$, will almost always underestimate the cost as calculated by our cost function. This is of course admissible. However, the Gaussian distributions representing the dynamic obstacles can grow quickly and cover large areas of the graph where the dynamic obstacles will not actually be in the future. Due to this, the $g$ values will grow and cover areas that actually have low cost. This means that we have inadmissible $g$ values in the presence of dynamic obstacles. More clearly, we can calculate inadmissible $g$ values if we predict a dynamic obstacle will go somewhere in the state space, and then at the next time step, it does not go where we believed it would. Any nodes expanded during the previous iteration might now have inadmissible $g$ values. Although we use the same cost function as Kushleyev and Likhachev (2009), it is not clear that they realized that their $g$ values can be inadmissible. It can be argued, however, that in the context of their problem their $g$ values are not inadmissble. They do not perform any learning in their algorithm, so these values are forgotten between search iterations. Therefore, their $g$ values were not inadmissble given their model at the specific search iteration they were explored in. Ours only become

inadmissble in future search iterations because we remember these values between search iterations, at which point they may become inadmissible.

As far as we can tell, inadmissible $g$ values have not been explored in the literature. This would suggest that this problem may be an entirely new type of graph search problem. The technique I've devised, separates out the admissible from the inadmissible portions of the $g$ value, allowing us to do search while maintaining provable properties of completeness. Although, the technique is simple, it is easy to understand and works well in practice.

### 2.5.2 Heuristics

Admissible heuristic are those which will never overestimate the cost-to-go for a given state. Consistent heuristics are those defined as follows:

$$\forall s \in S, h(s) \leq c(s, s') + h(s')$$

$$h(g) = 0$$

that is, the estimated cost-to-go for all goal states is 0. For all other states, the estimated cost-to-go is always less than or equal to the cost of moving to a successor added to the estimated cost-to-go of the successor. All consistent heuristics are also admissible.

The heuristic functions we use in this thesis are both admissible and consistent:

- Straight line heuristic: Ignores both static and dynamic obstacles and calculates for a given $x, y$ location of the agent, what the cost of driving straight to the goal's $x, y$ location would be if moving at maximum speed.

- Dijkstra heuristic: We run a precomputed Dijkstra's algorithm starting at the goal node in a 2D $(x, y)$ grid world representation of our state space. Dynamic obstacles are not modeled. We also first expand the size of all obstacles by the radius of our agent before running this computation, as to prevent the 2D grid world planner from finding paths through static obstacles which we would not be able to fit between. The minimum number of grid world moves from a state to the goal is returned as the heuristic estimate.

- $h(n) = 0$: The weakest of our heuristics, it simply returns the estimated cost-to-go of 0 for all search nodes.

# CHAPTER 3

# PREVIOUS WORK

There have been a number of previous techniques proposed in the domain of robot motion planning, however, none that we have found have been designed with this specific problem in mind. In this chapter we give an overview of the current heuristic search techniques that solve similar problems and discuss the features of each that make these algorithms less attractive for this domain.

## 3.1 Real-Time Search Algorithms

Real-time search algorithms work by interleaving the planning and execution of a plan in such a way that they adhere to a strict time bound on how long their planning phases are allowed to take. The following algorithms follow this scheme.

### 3.1.1 Real-Time A* (RTA*)

Real-time A* (RTA*) was first described in Korf (1990). It was the first real-time algorithm to be invented and has been the basis of many other real-time algorithms since. RTA* was shown to sub-optimally solve very large instances (at the time) of the sliding tile puzzle problem. It works by generating the successors of the current state of the agent and doing some form of limited lookahead search to determine which successor to move to. The key step was a heuristic update that took place once the algorithm had decided which child to move to. After picking the best successor to move to, it updates a cached $h$ value of the node you were leaving to be the $f$ value of the second best successor. The intuition here

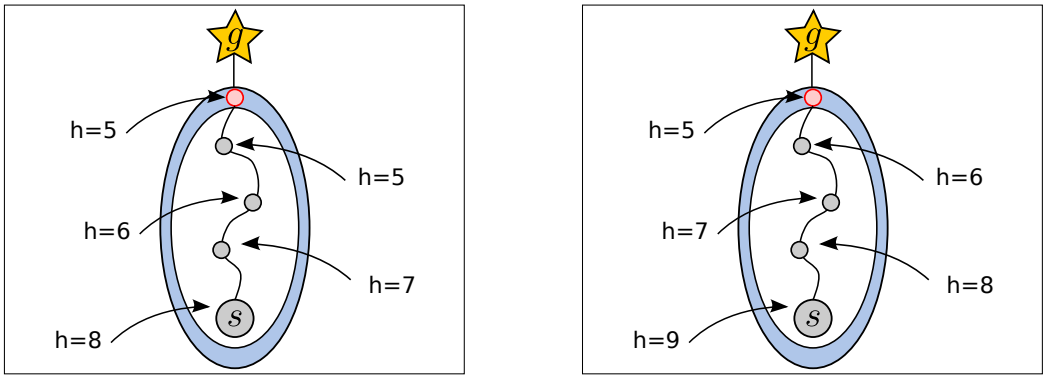


Figure 3-1: Left: Shows a high-level view of the LSS-LRTA* search after performing a limited lookahead. Right: Shows a high-level view of the LSS-LRTA* search after the Dijkstra backup rule has been performed.

is that if the algorithm ever returns to the node the algorithm is currently leaving, its $h$ value would have to be at least the $f$ of the second best node since it had already taken the best $f$ and returned. This works in domains with reversible operators and in the limit of search iterations, guarantees completeness. This means it is able to overcome admissible yet misleading heuristic functions that may lead the agent into local minima. However, this may take a very long time as we are learning improved $h$ values for states slowly. After all our lookahead search we only end up updating one search node's $h$ value. This wastes a lot of work and consequently requires a great many planning iterations to escape local minima.

### 3.1.2 Local Search Space Learning Real-Time A* (LSS-LRTA*)

The state-of-the-art real-time search algorithm Local Search Space Learning Real Time A* (LSS-LRTA*) (Koenig and Sun, 2009), works by performing A* search (Hart, Nilsson, and Raphael, 1968) forward from the current location of the agent towards the goal state, yet limits the number of node expansions it performs per search cycle to a fixed bound ($H_b$). This limited search generates what they call the local search space. At this point the algorithm selects the node that A* would have expanded next as its local goal $g'$, which is the node in OPEN with the lowest $f$ value. It then performs Dijkstra's algorithm back from

the search frontier throughout the local search space until all the nodes in the local search space have been touched. The algorithm is illustrated in figure 3-1. More information on the technique can be found in the paper (Koenig and Sun, 2009). The figures show a limited A* being performed and the subsequent learning step. The learning step is able to update the $h$ values, as some shown were too low. The Dijkstra step is performed to learn a more informed $h'$ value for the nodes in the local search space. The value $h'$ is more informed than $h$ because our original $h$ value for a node we expanded could have actually been an underestimate of its true $h^*$ value. Now by performing Dijkstra's algorithm back from the frontier throughout the local search space we are learning more accurate $h'$ values for each node in the local search space as Dijkstra's algorithm is calculating the cheapest path back from the frontier to every node in the local search space. These costs are then used as the node's new heuristic values. The algorithm then follows the path found by the A* search from $s$ to $g'$ and leaves the local search space before repeating this process. It is important to note that in updating the learned heuristic $h'$, LSS-LRTA* only ever updates the $h$ of a node if the learned $h'$ is larger than it previously was. The proof for this can be found in the work done by Koenig and Sun (2009). This makes it a more accurate estimate of the cost-to-go, as higher $h$ values give a better evaluation of the true cost to go as long as admissibility is maintained.

One problem with this technique is that as the agent is executing its plan in the local search space, it can be simultaneously doing search to find possible better plans now that it has more time to search and is given new observations of the world. LSS-LRTA*, as it is proposed, wastes this time and just continues to follow its returned plan until it has reached $g'$ unless the costs along the path from $s$ to $g'$ rise. This highlights yet another problem; the $h$ value given to a state never decreases in LSS-LRTA*.

To help explain this problem assume there is a planning agent running LSS-LRTA* that uses a heuristic function that gives high $h$ values for states containing static or dynamic obstacles and 0 for other states. Further assume that the robot is in the situation shown in Figure 3-2. During the A* search the robot first observes that at time $t = 0$ the state

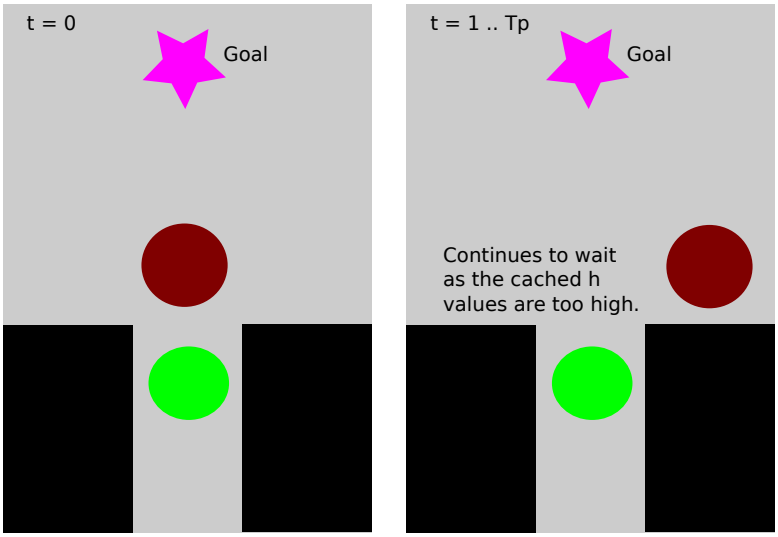


Figure 3-2: LSS-LRTA* struggles with states whose $h$ values should decrease.

directly in front of it contains a dynamic obstacle. The opponent model then predicts that this object will stay there for the foreseeable future. This means that each time that a state with time $0 < t < T_p$ is expanded by the A* search it will be given a high $h$ value indicating that there is a high cost associated with moving into the state. Once $T_p$ expires the robot will return the action to sit still as it is the best looking action to take. Now at time $t = 1$ assume the dynamic obstacle moves out of the way of the robot. The robot should then move into the space that was previously occupied by the dynamic obstacle, however, because during the previous search phase it was believed the dynamic obstacle would still be occupying that location it was awarded a high $h$ value. This $h$ value was then cached for reuse in future search phases. This means that the robot will continue to wait to move into the state in front of it for as many time steps as the state was believed to contain the dynamic obstacle. Only once that number of time steps has passed will it be able to realize that the $h$ value for that state is no longer high and will move into it. Although this is a pathological example, there are others like it that are less pronounced yet will still have this negative effect of not decreasing $h$ values when they should be decreased.

### 3.1.3 Real-Time D* (RTD*)

Bond et al. (2010) take a state-of-the-art search algorithm targeted at domains with dynamic environments, D* Lite (Koenig and Likhachev, 2002), and combine it with a state of the art real-time search algorithm LSS-LRTA* (Koenig and Sun, 2009). Their technique splits the planning phases into two sub-phases, the first runs D* Lite, which attempts to search *backwards* from the goal state to the current location of the robot. If a complete path from the goal to the robot is found, the robot follows this path, as D* Lite returns optimal solutions. If, however, a complete path cannot be found in time, it switches to LSS-LRTA* to quickly find a suitable action to take in the remaining time. The agent then executes the given action and returns to the planning phase. A nice feature of D* Lite is that it is able to reuse work from previous searches across planning stages, as it plans backwards from the goal towards the agent. This means it can quickly converge to the optimal solution even as the robot moves about the world.

This algorithm, called Real-Time D*, works well in dynamic environments, however, it is infeasible in our domain as the inclusion of time into the state space makes it impossible to search backwards from the goal. This is because we do not know what time the robot will reach the goal, and furthermore, the inclusion of time prevents us from predicting where the dynamic obstacles will be during the backward search. If it cannot be determined what time the agent will reach the goal state, it cannot be determined what the locations of the dynamic obstacles are as the backward search progresses. One could of course plan backwards starting at *all* possible future times out to some arbitrary bound, but this would present such a massive explosion of the state space, in addition to being incomplete if the bound is not set correctly, so it would be completely infeasible.

### 3.1.4 Real-Time Adaptive A* (RTAA*)

Real-Time Adaptive A* (Koenig and Likhachev, 2006) is a tweak on LSS-LRTA* which attempts to reduce the overhead of the learning step of LSS-LRTA*. RTAA*'s learning step

simply takes the iterates over the closed list generated from its A* lookahead and updates all node's $h$ values according to the following rule:

$$h(s) := f(g') - g(s)$$

While this update is indeed much faster taking $O(n)$ time where $n$ is the size of the closed list which is controlled by a *lookahead* parameter, and therefore allows for larger lookaheads, the heuristic learning is not as accurate as using LSS-LRTA*'s learning rule. In their published results the algorithm does worse, yet very similar to LSS-LRTA* in the grid-world domain. In this way it is very similar to LSS-LRTA* allowing for larger lookahead and less heuristic learning per search iteration. It inherits the same flaws as LSS-LRTA* discussed above.

### 3.1.5 Shortcomings

All the real-time algorithms overviewed in this section can ultimately be mislead by even a "more informed" heuristics such as a 2D Dijkstra and become stuck in this local minima. Normally real-time search algorithms use a learning technique to escape local minima, which all those presented do. However, one could build an arbitrarily large example in the same format shown in figure 3-3 which will not be escapable using the current techniques. The example shows a green agent in a local minima. Both the straight line heuristic and the 2D Dijkstra heuristic will lead the agent into this path. Although the windy path is wide enough to handle the agent, if the agent does not support the ability to turn in place, such as a car, it will not be able to traverse the windy path. Because neither the straight line heuristic, nor the Dijkstra heuristic take the agent's motion model into account, the heuristic function will still give these areas very promising looking $h$ values which will mislead the search. Therefore, these algorithms will not be able to escape this local minima because the states in our space have a time stamp. This means that the information they learn to escape will be for a state now in the past. This is a major shortcoming of the current state-of-the-art techniques which can make them incomplete.

Another shortcoming that is important to note is that, the real-time algorithms pre-

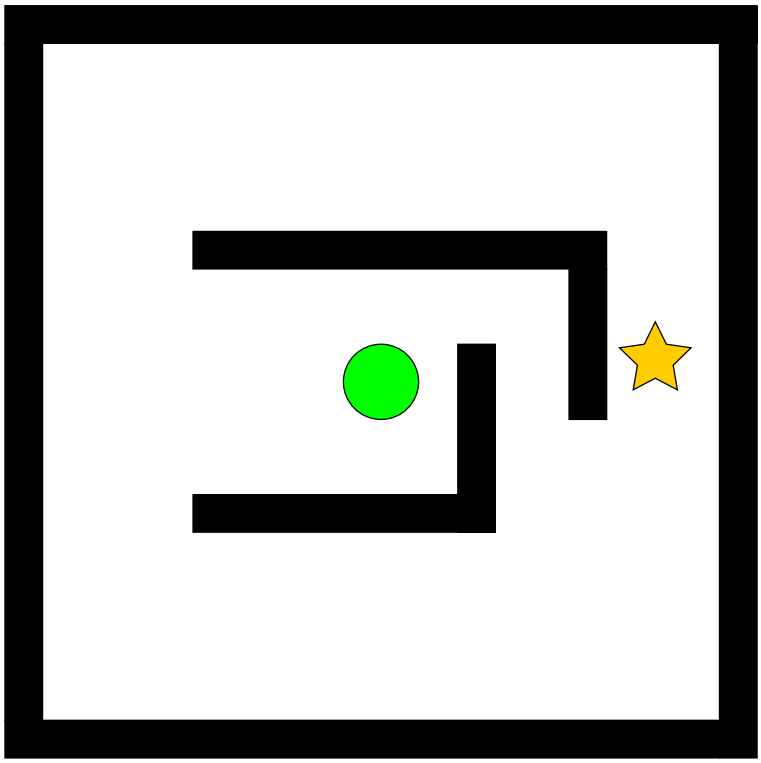

Figure 3-3: A straight line heuristic as well as a 2D Dijkstra will mislead real-time algorithms into the local minima.

sented in this section are not be able to escape local minima when applied to our domain in a straightforward way if their lookahead is not sufficiently large. This is due to each state having a time stamp associated with it. This means that the $h$ value that the algorithm caches for a given state will no longer be relevant at some point as each successive planning iteration will cause many nodes to become obsolete as their timestamps represent states in the past. If they were to partition their heuristic values, however, they could store the $h_s$ value of the state independent of time to allow them to escape the local minima created by the static world once more.

## 3.2 Anytime Algorithms

Anytime algorithms work by finding incrementally better solutions to a search problem and then returning the best found when they are interrupted. If interrupted before they find their first solution they return no solution.

### 3.2.1 Anytime Repairing A* (ARA*)

Anytime algorithms have also been utilized in robot motion planning. An anytime algorithm was demonstrated by Likhachev and Ferguson (2009), which utilizes Anytime Repairing A* (ARA*) (Likhachev, Gordon, and Thrun, 2004) with great success. It was one of the main path planning algorithms used in the vehicle BOSS which won the DARPA Urban Challenge (Urmson et al., 2008). ARA* operates by attempting to find a suboptimal solution as quickly as possible and then continue searching for better solutions until the search is interrupted, at which point the best incumbent solution is returned. ARA* provides bounds on the sub-optimality of the solutions it finds. The issue here is that finding that initial solution can take an unbounded amount of time. If interrupted before the initial solution is found these algorithms return no solution. We are also not concerned with complete solutions to the goal, we are concerned with only coming up with the best action to take at the current time. While planning a complete path to the goal configuration during each planning phase can help, it is not necessary.

## 3.3 Offline Algorithms

Offline algorithms work by planning from a start state all the way to the goal at each time step. This means they are able to return complete paths to the goal instead of just a single action to execute.

### 3.3.1 Time-Bounded Lattice

Recently, a technique addressing nearly our problem domain was described by Kushleyev and Likhachev (2009). Their method attempts to deal with the issue of dynamic obstacles discussed earlier by planning in a 5D space $(x, y, \theta, v, t)$ out to a point where their predictions of the dynamic obstacle's movements can no longer be reliably projected ($T_b^{max}$) and then switches to a 2D grid world $(x, y)$ to plan the remaining steps to the goal. They use WA* as their search algorithm, which is simply a form of A* with a user defined weight on the heuristic value of each state. This works well most of the time, however 2.4% of the time, it took over a half second, and sometimes up to 10+ seconds to come up with the next action to take in their experiments. Clearly, this is not desirable. Further, the amount of time taken to find these plans will only increase as the worlds become larger with the start and goal locations of the robots being further from each other, and as the number of dynamic obstacles in the world rises. The weight chosen for the heuristic can also greatly affect the search times. As such, this technique will not meet the real-time requirement of the problem domain. Furthermore, if there were no path to the goal, this algorithm would never terminate as our state space is infinite due to time. This means that the robot could be left vulnerable to dynamic obstacles in the world while it is stuck planning.

It is also important to note that due to the fact that after the time bound has expired and the search reverts down to a low-level 2D Dijkstra search, this technique can also become trapped in local minima as demonstrated in figure 3-3. This is again because after the timebound, all dynamics are stripped from the problem, including motion constraints and dynamic obstacles. So for a large enough situation similar to figure 3-3 the planner would become trapped. This would not happen, however, if the algorithm never switched down to the 2D grid search.

### 3.3.2 Iterative Accelerated A* (IAA*)

Iterative Accelerated A* (Kopriva et al., 2010) expands on the earlier algorithm Accelerated A* (AA*) (Šišlák, Volf, and Pěchouček, 2009) algorithm, that both make use of adaptive sampling, that is, they choose larger action primitives for their expand function when far from obstacles and shorter action primitives when near obstacles. This helps them cut down on the number of states in the search space. The iterative version takes another step forward by only including a small set of obstacles in its first planning iteration. After it finds a plan, it checks to see if there are any collisions considering all of the obstacles in the world. If the solution is found to be collision free, the algorithm exits and returns the plan. If however collisions are detected, the obstacles that caused the collisions are added to the obstacle set considered while planning, and the search repeats until a collision free path is found and returned. This was tested in the domain of trajectory planning for aircraft with no fly zones. The paper did not represent time in their state space and as such it is unclear how this method would perform with the dimension of time. They also did not include speed in their state space, however, they did limit themselves by only using motion primitives that obey the nonholonomic movements of an aircraft. In addition, they tested their technique by planning 369 flights using real Federal Aviation Administration (FAA) data. It is to be noted that they plan the path for each of these flights sequentially, that is, they control the path that each plane takes and as such they can ensure that all paths will be collision free. So although their technique of using a subset of the obstacles for finding paths and expanding the subset if necessary may be useful in this domain, they dealt with an inherently different problem domain. It is also unclear how to use this technique with a bounded time allotted to each planning phase. Because their technique potentially runs multiple iterations of search until a solution is found, it is not clear what to do if a solution could not be found within $T_p$ time units.

## 3.4 Other Approaches

The work of Snape, Guy, and van den Berg (2010) introduces the idea of optimal reciprocal collision avoidance for collision and oscillation free navigation of an agent in a world with other dynamic agents. They allocate the responsibility for avoiding collisions to both the agents, assuming that both agents want to avoid the collision. However, their approach only works if both agents are running the same algorithm, and will most likely fail if one is not attempting to avoid a collision, i.e. randomly moving obstacles or antagonistic obstacles. They also did not supply any results of their algorithms in practice, nor did they go into detail about the underlying search algorithms, so it is unclear if this takes a real-time approach to the motion planning problem.

# CHAPTER 4

# PARTITIONED LEARNING TECHNIQUES

In this chapter we introduce the three main contributions of the thesis. First, we will cover the partitioned heuristic concept which is used to aid in effectively learning heuristic costs in this domain. Second, we discuss heuristic decay, a technique to be used in conjunction with the partitioned heuristic, allowing us to both make dynamic cost generalizations over states differing only by time in the search in addition to allowing us to decay inadmissible heuristic estimates to keep the algorithm complete. Third, we will see a garbage collection technique that can be used to reduce the memory overhead of the partitioned heuristic and decay technique. Finally, a new algorithm called PLRTA$^*$ is presented that integrates all three of the techniques.

These techniques were all introduced to deal with deficiencies present in the current state-of-the-art algorithms when applied to the robot motion planning domain. Because of the inadmissibility of the $g_d$ costs due to the dynamic obstacles, new techniques and algorithms that utilize those techniques must be introduced.

## 4.1 Partitioned Heuristics

Partitioned heuristic tracking is a technique for separating the cost-to-go into two components: the portion due to the physical act of moving around the world, and the portion due to the presence of dynamic obstacles. This gives us additional information sources to make

more informed decisions about how to learn heuristic values for states. Thus, we break the standard $h$ value up into two separate $h$ values, $h_s$ for the static cost-to-go estimate and $h_d$ for the dynamic cost-to-go estimate. Obviously, the cost of moving in a state due solely to the static world is independent of time. That is, if there are no dynamic obstacles present, the $h_s$ value of that state would hold true regardless of the time stamp. The benefit of this is seen when using an algorithm which utilizes a learning step to both help correct heuristic error and escape local minima. We can use these partitioned values to help us learn more informed static heuristic functions as well as dynamic heuristic functions by properly attributing the portions of the cost as either static or dynamic. This means, however, that we must also partition the cost-thus-far values (the $g$ values). Therefore, each node in our search space when using the partitioned heuristic technique must track the following:

$$f(n) = g(n) + h(n)$$

$$g(n) = g_s(n) + g_d(n)$$

$$h(n) = h_s(n) + h_d(n)$$

With these values tracked for each search node, it is simple to establish what each cost incurred is due to and thus more easily facilitate the learning of improved heuristic functions for each, as we shall discuss in the following sections.

### 4.1.1 Ordering Predicate

Another important note to make is that when tie-breaking on equal $f$ values in the openlist we do not want to tie break on higher $g$ values as is standard in heuristic search. This is because a $g$ value is now the linear combination of both a $g_s$ value and a $g_d$ value. This may encourage the search to first explore nodes which have a higher chance of colliding with dynamic obstacles which is counter to our objective. Therefore, we should tie-break equal $f$ values by higher $g_s$, the intuition here being the same as it is in standard heuristic search:

states with higher $g$ values are likely closer to the goal and have put less of the guesswork of a node's $f$ value into the $g$.

## 4.2 Partitioned Learning

Many real-time search algorithms make use of some form of heuristic learning ranging from LRTA* (Korf, 1990) to LSS-LRTA* (Koenig and Sun, 2009) and many other derivatives of these algorithms. These algorithms do so, first and foremost, to enable themselves to correct for inaccurate heuristic functions and allow themselves to escape from local minima present in the heuristic function. We find LSS-LRTA*'s learning step to be the most effective in its ability to learn an improved heuristic value for a large number of states in each learning step as shown in the work by Koenig and Likhachev (2006). We will now discuss modifying Koenig and Sun's learning step to work with partitioned heuristics.

### 4.2.1 Static World Learning

First, using the definitions in section 2.1.1, we make the following assumptions for the learning step:

- $\forall s \in S - G,\ c(s) > 0$
- $\forall g \in G,\ c(g) = 0$
- The static cost for a state is unaffected by time. That is, two states differing only by time must have the same static cost and estimated cost-to-go. We will refer to this idea of a state independent of time as a pose $(x, y, h, s)$ henceforth in the thesis.

The idea of the static world learning is to make up for the mistakes in your original heuristic function by utilizing the search you perform. After every forward search of a planning iteration, a learning phase may be invoked to possibly improve the heuristic estimates for those states explored. This will allow the algorithm to correct for underestimates in the cost-to-go. Heuristics are *estimates* of the cost-to-go and as such can be wrong. The $g_s$

**Algorithm 1** Initialize Dijkstra's

InitDijkstra(Closed, Open)

1: Closed$'$ = {}
2: **for** $n \in$ Closed **do**
3:  **if** $n \notin$ Closed$'$ **then**
4:  $n.s_h \leftarrow \infty$
5:  Closed$'$ = Closed$'$ $\cup$ { $n$ }
6:  **end if**
7: **end for**
8: $\forall n \in$ Open, if $n \in$ Closed$'$ then Open = Open - { $n$ }
9: return Closed$'$, Open

costs, however, are known. Therefore, we can utilize the $g_s$ costs to help reduce the amount of estimation if the $h_s$ value by updating each nodes $h_s$ to be the value which minimizes the known cost of moving to a child on the node, plus their estimated cost to go. This puts more of the estimate of the cost to go into a known value.

Much like in the learning step of LSS-LRTA*, our learning step attempts to update all values in the local search space (LSS), which is precisely the closed list after A* has executed in the case of LSS-LRTA*, but in practice can be any lookahead search. We also need the frontier of the previous iteration of the lookahead search, which is precisely the open list of A* when it terminates in LSS-LRTA*.

Unlike in LSS-LRTA*, we do not want to sort the open list by lowest $h$ but by lowest $h_s$ for the static learning phase. The main difference between the two learning steps is in the setup phase. Not only do we need to touch every node in both the open and the closed lists while setting every node's $h_s$ in the closed list to $\infty$ and reordering the open list by $h_s$, but we must also weed out duplicate states ignoring time as shown in Algorithm 1.

This algorithm works by removing duplicates in the closed list so there is only a single node representing the pose in the closed list. This is sound because we have not removed

**Algorithm 2** Dijkstra's Algorithm for learning $h_s$ values

Dijkstra(Closed, Open)

```
1: Closed, Open ← InitDijktra(Closed, Open)
2: while Closed ≠ ∅ AND Open ≠ ∅ do
3:    delete a state s with the smallest h_s value from Open
4:    if s ∈ Closed then
5:       Closed ← Closed \ {s}
6:    end if
7:    for p ∈ predecessors(s) do
8:       if  p ∈ Closed AND h_s(p) > c_s(p, s) + h_s(s) then
9:          h_s(p) ← c_s(p, s) + h_s(s)
10:         if p ∉ Open then
11:            Open ← Open ∪ {p}
12:         end if
13:      end if
14:   end for
15: end while
```

any states from the closed list entirely, only cleaned up the duplicates, so each pose that was represented in the closed list previously is still represented afterward. Also, because of our assumption that all states which represent the same pose must share the same $h_s$ we know all the equivalent states shared the same $h_s$ value and continue to do so now that they are set to $\infty$. Now, when we generate and attempt to find predecessors in the closed list during the learning state, we simply ignore the time in the predecessor states. Therefore, by removing duplicates from the openlist, we have not removed any useful information, as they share the same $h_s$ value as each other.

On line 8, we remove any node in the openlist that also appears in the new closed list. This does not hinder us in our learning in any way as the duplicate node in the open list

will have the same $h_s$ value, therefore, there will be nothing useful to learn from this state.

We then perform the learning step of LSS-LRTA*, making sure to simply abstract out the time from each state, and using the $h_s$ of a state and the static cost ($g_s$) incurred by moving from one state to its successors. The only other modification to the learning step is the termination condition. Our modified version is shown in Algorithm 2. As it is presented by Koenig and Sun (2009), LSS-LRTA*'s learning step only terminates when the closed list has become empty. This happens when every node in the closed list has been updated to its new learned $h_s$ value. However, this might never happen in our version because some of our states have no successors. If this is the case, then we must stop when either the open or closed list have been exhausted. We now prove some interesting properties of static world learning.

### 4.2.2 Properties

If the static update algorithm terminates due to the closed list being empty, then we have the same termination condition as LSS-LRTA*. If it terminates due to the openlist becoming empty, that means there are nodes left in the closed list which were not generated as predecessors of any nodes of the openlist. This can only happen when a node in the closed list had no successors during the A* search. Furthermore, because it was not generated as a predecessor, it still has $h_s = \infty$ from the initialization phase, which is precisely the $h_s$ value it should have.

**Theorem 1** *If the static learning step terminates due to an empty openlist, it is because the remaining nodes in the closedlist are those nodes whose successors lead to dead ends.*

***Proof:*** The proof is by contradiction. Assume there is some node in the closedlist when the algorithm terminates, which we will call $n$, and assume further, that $n$ has some successor that does not exclusively lead to a dead-end. The algorithm must have terminated due to an empty openlist as the algorithm only terminates when either the openlist or the closedlist becomes empty. This means that $n$ must have had a descendant (either a direct successor

or a node along the path going through a direct successor) on the openlist as some point during the search. If there did not exist a descendent of $n$ on the openlist then we have a contradiction that $n$'s descendents do not lead exclusively to dead ends. Therefore, let us call this descendent who was on the openlist $m$. Because of the termination condition, we know that $m$ was removed from the openlist as the smallest $h_s$ value at some point during the learning step. Once removed from the openlist, $m$ is removed from the closedlist if it appears in it. Then, all of $m$'s predecessors are generated, and those which appear on closedlist and have $h_s$ values greater than the cost of moving to $m$ plus $h_s(m)$ are inserted into open. The condition:

$$h_s(predecessors(m)) < c_s(predecesors(m), m) + h_s(m)$$

will hold for any of the $predecessors(m)$ on the closed list at least once, as all nodes on the closed list have their $h_s$ values set to $\infty$ in Algorithm 1. Therefore, $m$ must have at least one predecessor in the closed list which gets inserted into the openlist, otherwise, $m$ would not be in the openlist. It then follows that at some point in the future that predecessor of $m$ would be removed from the openlist and removed from the closedlist, its predecessors generated and placed on the closedlist. Ultimately, because $n$ is an ancestor of $m$, $n$ would have to be inserted onto the openlist and sometime in the future removed from both the openlist and the closedlist. But this is a contradiction, because we stated that $n$ was on the closedlist at termination. Thus, it cannot be true that $n$ has some descendent who does not lead exclusively to to a dead-end. Therefore, $n$ must exclusively lead to a dead-end. □

We have the benefit of declaring that, given enough time to explore the search space, our $h_s$ values would ultimately converge to their true values. The proof is the same as that in Koenig and Sun (2009) with the exception that we are updating $h_s$ values and not $h$ values. This is intuitively easy to grasp, as the $h_s$ values depend solely on $g_s$, which is the same as in the static problems LSS-LRTA* was originally designed to deal with.

**Theorem 2** *The $h_s$ value of the same pose is monotonically nondecreasing over time and*

*thus remains constant or becomes more informed over time.*

***Proof:*** We rely on the proof of Theorem 1 shown by Koenig and Sun (2009). One simply must substitute their use of $h$ with $h_s$ and their notion of a state with pose. We have also assumed our $h_s$ values to be consistent and we use the same Dijkstra style learning rule, which are assumptions for their proof. This means that all the preconditions for their proof have been met and as such, their proof follows trivially. □

**Theorem 3** *The $h_s$ values remain consistent and thus also admissible.*

***Proof:*** We again rely on the proof of Theorem 2 shown by Koenig and Sun (2009). The only modification to their proof is to substitute their use of $h$ with our $h_s$ and their use of state with pose. Again, our $h_s$ heuristics are consistent, and we use the same Dijkstra style learning rule, which are assumptions for their proof. This means that all the preconditions for their proof have been met and as such, their proof follows trivially. □

### 4.2.3 Dynamic World Learning

We would also like to be able to learn $h_d$ values for states in order to speed up future searches and allow the search to avoid areas of high cost caused by dynamic obstacles. These $h_d$ values of a state can frequently change with respect to time. Think of a dynamic obstacle moving throughout the world; the given cost-to-go of a node can fluctuate as time passes. Thus, each state with the inclusion of time will map to its own $h_d$ value.

It is hard to come up with an accurate heuristic function that can by computed quickly enough to be able to be run for each node generation. This is one reason we need to learn $h_d$ values. This way, we can start with a weak heuristic say $h_d(n) = 0$ and improve on it after each search iteration by performing $h_d$ value learning.

Because each subsequent search iteration will likely explore much of the same area in the graph as the previous iteration, the caching of the $h_d$ values can allow us to save and reuse the information we've learned in future searches. This allows our future search iterations to avoid the high cost areas of the graph and allow it to possibly explore more lucrative areas of the search graph.

Our learning rule for the $h_d$ values is as follows:

$$h'_d(n) = [\min_{n' \in succ} g_d(n') + h_d(n')] - g_d(n)$$

where $h'_d$ is the new learned dynamic $h$ value and $g_d$ is the part of the node's $g$ cost incurred from the dynamic obstacles in the world. The intuition here is the same as that of the static world learning. We learn a better heuristic value by modifying a node $n$'s $h_d$ value to become the best $g_d + h_d$ of its children, minus the cost to get to $n$, which is recursively computed in the same way.

This is precisely the type of update rule that is well suited to using a Dijkstra-style traversal of our local search space. This is performed much in the same way as the static world learning step. We do not need to prune out any duplicates, as the time of the state is important in determining its $h_d$ value. The termination condition, however, is the same.

Once these $h_d$ values have been calculated, they can potentially allow our search to avoid areas we thought to be of low cost-to-go due to our weak initial heuristic, yet were found to have a high cost through search. This information is useful for steering our search away from these areas of high dynamic cost due to dynamic obstacles in the subsequent searches which allows us to find less risky paths to the goal.

## 4.3 Heuristic Decay

Dynamic obstacles create an interesting problem for the learning step of the search algorithm. Normally, when only static obstacles are involved, the environment is not changing. This is useful to leverage because once a promising-looking path has been found to be less

fruitful than it initially seemed, the learning step will act to raise the heuristic estimate in that portion of the graph, potentially driving the search elsewhere in the following search iterations. Because the world is unchanging, the learning is correct and one can show that the heuristic estimate for a state (when using a consistent admissible heuristic) will never decrease while maintaining admissibility (Koenig and Sun, 2009). This is useful because once we have determined an area of the graph to have high $h$ values, the search can ignore that section of the graph unless it is absolutely necessary to return, for example, when the solution path lies through that area. Furthermore, it allows the algorithms to escape local minima by learning that its original estimation of the cost-to-go was in fact too low.

This is no longer true when dynamic obstacles are involved, however. Areas that may be deemed high cost and yield high $h$ values may only be that way due to a dynamic obstacle passing through at the time observed. If that obstacle were to then move elsewhere, the cost and $h$ value should decrease but do not as the algorithms are currently proposed. Again this is because our inadmissible $g$ values, due to our $g_d$ values almost always overestimate the cost incurred to reach a node in the search space. This is because if we predict a dynamic obstacle will move to a certain location, and then at the next planning iteration find it did not move as we expected, the $g$ values calculated in the previous search iteration were an overestimate and thus, inadmissible.

Adding to this issue is the inaccuracy of the opponent model. Using an inaccurate opponent model mixed with the traditional learning step can yield strange results in certain situations. Take, for example, the scenario shown in Figure 4-1. This shows an issue that can arise when using an algorithm such as LSS-LRTA*, which will learn that the state directly in front of it has a high cost associated with it. This will be learned not only for that pose at a single time but possibly up to roughly $lookahead/2$ time steps. This is because, to actually generate the node directly in front of the agent at any time step after the initial time step, you must also generate the node representing the state when the agent does not move. So, it is a two expansion step to generate the node representing the state in front of the agent, hence, $lookahead/2$. Although this looked correct at the time, if at the

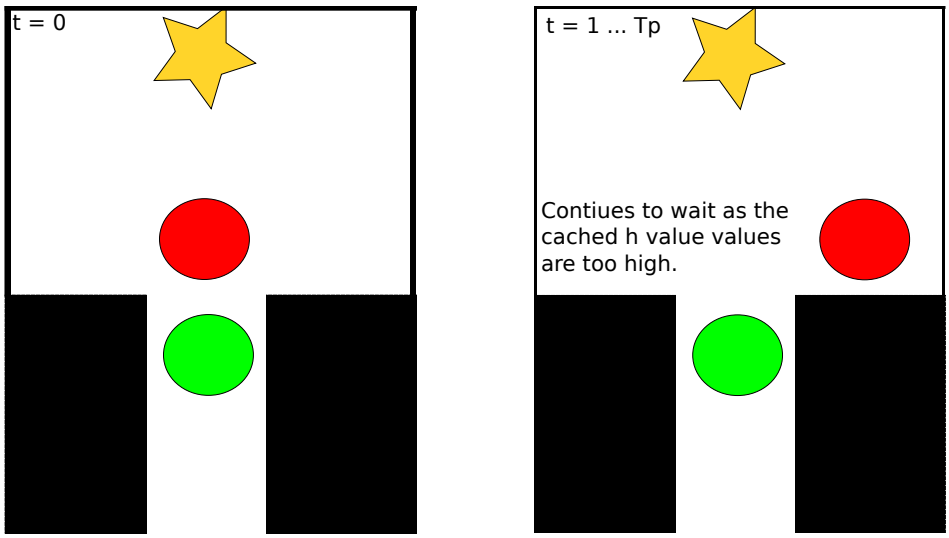


Figure 4-1: Agent (green) first observes that the state achieved by applying the move forward action is high cost and also has a high $h$ value due to a dynamic obstacle occupying the space (red). Depending on the *lookahead* and the opponent model, the agent can learn that the state directly ahead of it will have a high cost and $h$ value until the *lookahead* expires. If at the next time step the dynamic obstacle moves, the agent will remain stationary because its learned view of the state is incorrectly rated as high cost.

very next time step the obstacle moves, we are left with our cached $h$ values that tell us to wait it out until the $lookahead/2-1$ more time steps have expired. Again, these inadmissible $h$ values were learned through our inadmissible $g_d$ values. While this a dramatic example of a situation where caching the value can cause some strange behavior, it is not hard to imagine other situations where incorrectly caching a high $h_d$ value may prevent the search from heading down a path that actually has a lower cost than our cached view.

Depending on how far our search is able to look ahead, we need to be able to "unlearn" $h_d$ values we have assigned to states that may no longer be valid. We refer to these $h_d$ values as being stale. This will not only allow us to re-explore states that originally looked high cost due to the dynamic obstacles, but also more quickly re-evaluate states that seemed to be low cost when we initially cached their values. It also insures that if our learning step caused us to increase a state's $h_d$ value to an inadmissible value, it will ultimately return to being admissible.

To address these issues, we have developed a technique to decay the heuristic values dealing with the dynamic obstacles in the world or $h_d$. The general idea is as follows: assume each planning phase is numbered $p_i$ where the first is $p_0$. Whenever a $h_d$ value is learned and cached we note what planning phase $p_i$ it currently is. Then at some future planning phase $p_j$ where $0 \leq i < j$ the value of the cached $h_d$ should be decayed because it was first learned in a previous planning phase that may have learned inaccurate information. This encourages the search algorithm to potentially re-evaluate the node when it is next generated in some future planning phase instead of just using the cached value. This way, the possibly inaccurate opponent model would not prevent the planner from quickly finding paths that were previously thought to have high costs associated with them when that may no longer be the case.

There are a number of ways the $h_d$ value can be decayed. The most obvious technique would be to have some constant $t_d \geq 0$ that represents the number of planning phases that must pass before the $h_d$ value of a node is considered stale and must be re-evaluated. This technique can be handled in a few ways. The $h_d$ value can be held constant until

$t_d$ planning iterations have passed before the cached $h_d$ value is wiped and must be re-evaluated. This seems rather coarse, however, so another simple way would be to decay the $h_d$ value incrementally at each planning iteration until $t_d$ steps have passed, again at which point it would be removed from the cache.

Both of these methods suffer from having to pick a value for the $t_d$ parameter which may not be intuitive. The first method seems very rigid in that the $h_d$ value can be very high for $t_d$ planning phases before becoming very low once more. Conversely, while the second method of degrading the heuristic value at each time step may seem more natural, it still requires some tuning to find an appropriate amount to incrementally decay the $h_d$ value by.

Another decay technique would require more in-depth tracking of cost and more specifically which dynamic obstacle was responsible for the cost. The decay factor can then be dynamically selected, choosing higher rates of decay for opponents who are moving more quickly or unpredictably, and slower rates for those moving more predictably. One could then learn on-line what the underlying distribution is for correctly identifying the movements of the opponents. This would enable the decay technique to be more informed about how rapidly to decay $h_d$ values.

### 4.3.1 Algorithm

Calculating the decay of a node takes place upon the generation of the node. Whenever a node is expanded and its children generated, we must calculate a $h_d$ value for each child. To do this we use Algorithm 3. This algorithm shows how we calculate a $h_d$ value for a state using a linear decay technique. To calculate the $h_d$ value, we need the state $s$ the node represents, how many planning iterations must pass before the value becomes completely decayed (*decay_steps*), the *cache* used to lookup the stored values and finally, the *current_search_iter* which is the current search iteration the planner is in. If the state does not have a cached $h_d$ value then we simply use the standard dynamic $h$ function. Otherwise, we get both the original cached $h_d$ value as well as the search iteration it was stored in. Using the search iteration it was cached in, we can find the delta ($\delta$) of search

**Algorithm 3** Algorithm for getting the $h_d$ value of a node using linear decay

GetDynamicH($s$, *decay_steps*, *cache*, *current_search_iter*)

1: **if** $s \notin cache$ **then**
2: **return** $dynamic\_h(s)$
3: **end if**
4: $search\_iter,\ h_d \leftarrow cache.get(s)$
5: $\delta \leftarrow current\_search\_iter - search\_iter$
6: $h'_d \leftarrow h_d - (\delta * (h_d / decay\_steps))$
7: **if** $h'_d \leq 0$ **then**
8: $cache.remove(s)$
9: $h_d \leftarrow dynamic\_h(s)$
10: **end if**
11: **return** $h'_d$

iterations between the current one and when it was stored. We then calculate what the current decayed $h'_d$ value should be by subtracting the product of $\delta$ and the amount we should decay at each time step, from the original $h_d$ value. Because nodes only have their $h_d$ values decayed if they are generated during the search iteration, it may be the case that we do not generate a node for some number of search iterations after originally caching it. This leaves the possibility that calculated $h'_d$ might result in a negative number. If $h'_d$ turns out to be negative or zero, then it has been fully decayed. This means we should remove it from our cache and use the dynamic $h$ function.

### 4.3.2 Correctness

We assume that the combination of both our unmodified static heuristic function $h_s(n)$ and dynamic heuristic function $h_d(n)$ is admissible and consistent. In the sections on our partitioned heuristic learning (4.2.1), we proved that our $h_s$ values are monotonically non-decreasing and remain consistent and admissible. Because our proposed decay techniques

do not affect the $h_s$ values, this still holds. However, our $h_d$ values may actually fluctuate greatly and are by no means guaranteed to be admissible once learned. As a reminder, our evaluation function is:

$$f(n) = g_s(n) + g_d(n) + h_s(n) + h_d(n)$$

We follow the same technique for calculating $h_d$ values as we do for $h_s$ values. If the state has a cached $h_d$ value, we use the cached value, otherwise we use our dynamic heuristic function to calculate a value for the node. A value is never cached unless it is updated through the learning step. Now assuming we have a cached $h_d$ value, we know it has been learned via the learning step. Once this has been established, there are only two scenarios for changing the value, which we will now cover.

The only situation in which the $h_d$ value will increase above the original $h_d$ functions value for a given state is during the learning step. When the learning step executes, it may raise the $h_d$ value to become a more informed value, this in itself will not cause the $h_d$ value to become inadmissible. During the learning step, given the information available from the opponent model, we are learning admissible values following the proof of Koenig and Sun (Koenig and Sun, 2009). However, in the following search iterations, the value learned for a state may no longer be admissible. This is again due to the inaccuracies in the opponent model. Because we may have predicted that at some future the dynamic obstacle would be in some area, we may have learned that such an area has a high $h_d$ value, but if it turns out to no longer be the case, i.e., the dynamic obstacle did not move in the way we predicted it would, our learned $h_d$ value still may be high depending on the heuristic decay function. This could of course cause the value to now be inadmissible.

There is only one case when a $h_d$ value for a state will decrease; that is when the decaying of the value takes place. Decay conceptually takes place before each planning phase. That is, for a given planning iteration a node representing a given state at some specific time, will have the same $h_d$ value every time it is generated or expanded. It is not until the learning step or the beginning of the next planning phase that this value might change. The decay

is an important ingredient in keeping our $h_d$ values admissible.

This is a clear advantage over other algorithms in the previous work. They were not constructed to handle the inclusion of time in a state, they are unable to employ these techniques to prevent their algorithms from assigning inadmissible $h$ values to states if the dynamic heuristic information is included in the evaluation of a state. This is because they do not store both a $h_d$ and $h_s$ value for each node. They also cannot differentiate what portion of the cost of executing an action came from the dynamic obstacles in the world and which came from the static cost of moving. This cripples their learning step by allowing them to learn vastly inadmissible $h$ values for states. These values will remain inadmissible for that state until it is in the past making it irrelevant.

### Proof of Correctness

Assume we are given an admissible dynamic heuristic function $h_d(n)$. That is, it provides a lower bound on the cost-to-go to the goal due to dynamic obstacles in the world. Because it may be incorrect (an underestimate) of the cost-to-go, there may be points during the planning iterations that we may increase the given $h_d$ value of a node. During the learning step we may end up raising the $h_d$ value of a node by leveraging the the information gleaned from its successors. These values may end up being inadmissible when read from the cache in future planning iterations. However, using heuristic decay, we incrementally decrease the value assigned to a given state after each planning iteration. As long at this decrease in value is positive it is trivial to see that it will ultimately be lowered to a point at which it is no longer inadmissible.

**Theorem 4** *The value of any cached $h_d$ value learned during planning phase $p_i$ will ultimate become admissible at some future planning phase $p_j$, allowing the state to be re-evaluated, and thus will not prevent the search from reaching the goal.*

***Proof:*** Assume we are using the linear decay technique shown in Algorithm 3 and that we have a node whose $h_d$ value has been cached at some planning iteration $p_i$ and is

inadmissible. At some future planning iteration $p_j$ where $j > i$, the initial cached $h_d$ value will be decayed by some amount $\geq 0$, namely $(p_j - p_i) * (h_d / decay\_steps)$ . If $h'_d \leq 0$ after the decay, then we re-evaluate the state using the original dynamic heuristic function and remove its binding from our cache. Thus, our cached $h_d$ values will be admissible once again after

$$\frac{cached\_value - perfect\_value}{h_d / decay\_steps}$$

planning iterations. □

Furthermore, even if there is no heuristic decay employed, because the $h_d$ value is learned for a specific time-stamped state, that state will at some point become irrelevant as it will be in the past. This means that other states sharing the same $x, y, h, v$ may be re-evaluated at a future time step which may completely change the $h_d$ value of that state given the new world information and opponent model.

### 4.3.3 Note on Completeness

We cannot make any guarantee about the completeness of any algorithm used in this domain. Although some of the previous work make claims of completeness in their publications, they note that this only holds if their actions are reversible. This is not a property of our domain, so it is plausible that an algorithm may make a decision leading the planner into a dead end where it may not be able to escape from. Therefore, none of these algorithms can be proved to be complete. However, we have shown that our learning procedures will not impede completeness if a dead-end is not encountered.

### 4.3.4 Heuristic Decay Over Generalized State

Another way in which we can use this idea of heuristic decay to to use the $h_d$ of a state independent of time. Assume we have a cached $h_d$ value for some state $s_i$ which was generated in the $i^{th}$ planning iteration and representing some pose at time $t$. Assume

further that in a future planning iteration $j$ we generate a state $s_j$ with the same pose as $s_i$ yet represents the pose at a different time $t'$. If we have no cached $h_d$ information about $s_j$ we can use the information we've learned about this state at other timestamps. That is, we can use the $h_d$ value of $s_i$ and decay it according to how many planning iterations have passed since we've made that observation. For simplicity and consistency, we suggest using the same decay technique used for the decaying the heuristic value of a given cached $h_d$ value.

This technique is simply used to create an additional information source for calculating the $h_d$ value of a state. Since good heuristics for dynamic obstacles are hard to come by, any information we can use to separate good states from bad states is useful. Also, if there are multiple entries of $h_d$values for a given pose we simply use the one "closest" to the time-stamp of the state we are generating. For example, if we had a $h_d$ value for a pose at time 5 and 15, and we just generated a state representing time 12 at this pose, we will use the heuristic value found at time 15 and decay it.

### 4.3.5 Issues

There are a few known issues with the heuristic decay technique. Imagine the scenario given in figure 4-2. We see here that the agent is surrounded by an arbitrarily large wall of dynamic obstacles, creating huge local minima that the agent must search to realize the open path is to go all the way around the wall of dynamic obstacles. If decay has been enabled in any way, this wall could be constructed large enough such that the planner would never be able to escape from the local minima. This is because although the algorithm would initially learn and cache high $h_d$ values for the area and begin to leave the minima, once the values decay enough, the search would ultimately be lead back in the local minima causing this process to repeat.

Another issue is selecting a delay technique. As previously discussed, it my not be completely obvious how to decay or by how much. It is very much trial and error at this point and more research needs to be conducted to address this issue.

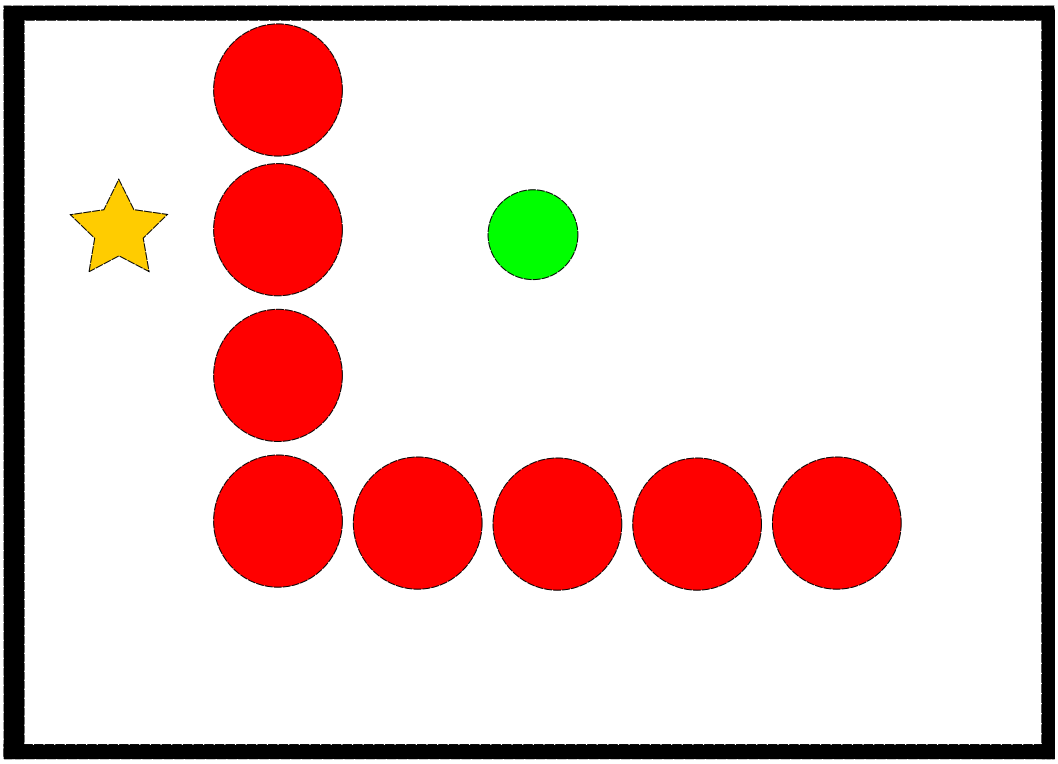

Figure 4-2: A bot surrounded by dynamic obstacles. If the values decay rather quickly and the lookahead is too limited, the planner may become stuck in the local minima created by the dynamic obstacles.

## 4.4 Garbage Collection

During our many planning iterations, states that have been explored and cached in previous search iterations will ultimately become useless as the times they represent fall into the past. If the system is memory constrained, one way in which we can save memory is to free these cached values when they're no longer needed. This can easily be done by keeping track of what time each cached search node represents and hashing them to a list of other nodes cached at that time using the time as the key. Then at the beginning of each planning iteration you simply hash into the table for the previous time and remove all the nodes found in the list from your cache.

**Algorithm 4** Partitioned Learning Real-Time A*

PLRTA($s_{start}$, *lookahead*)

1: COLLECT_GARBAGE()
2: open = $\{s_{start}\}$
3: closed = $\{\}$
4: ASTAR(open, closed)
5: $g' \leftarrow$ peek(open)
6: LEARN_STATIC(open, closed)
7: LEARN_DYNAMIC(open, closed)
8: **return** first action along path from $s_{start}$ to $g'$

If coupling this technique with heuristic decay however, it is important to keep around nodes which may be the sole representative of a time independent state. By this we mean that if a node to be garbage collected is the only node representing a given state in the $h_d$ value cache, then it must be kept around for the purpose of generalizing its $h_d$ value over other states identical in pose yet different in time. This is a simple constant time check and does not add any additional complexity to the garbage collection technique. If this cached value is fully decayed, that is, has reached the minimum value it can be decayed to until it is forgotten, however, it is no longer of any use and can be garbage collected. It is important to note that these nodes which are not garbage collected must be tracked on a secondary list to be checked at each garbage collection phase to see if they may be collected.

## 4.5 Partitioned Learning Real-Time A* (PLRTA*)

We now present an algorithm that combines all of these aforementioned techniques. Our goal for this algorithm was to combine these techniques in such a way to allow it to outperform the current state-of-the-art in our domain. Again, the objective of a search in our domain should be to minimize the cost incurred out to the simulation time limit.

Our algorithm is based on Local Search Space Learning Real Time A* (LSS-LRTA*) (Koenig and Sun, 2009), so we have named it Partitioned Learning Real-Time A*, however, these techniques are general and may be applied to any best-first search algorithm in this domain. So like LSS-LRTA*, we perform A* search (Hart, Nilsson, and Raphael, 1968) forward from the agent towards the goal state, yet limit the number of node expansions it can perform to a fixed *lookahead*. A more flexible implementation would allow the algorithm to know the amount of time it may run for, enabling the algorithm to decide when it must quit its search and return a solution, as opposed to always expanding *lookahead* nodes before returning the best action to take. We do not yet take this approach for simplicity's sake. Regardless, this limited search generates what Koenig and Sun call the local search space. At this point it selects the node that A* would have expanded next as its local goal and names it $g'$. As a reminder, we are using the following as our ordering function:

$$f(n) = g_s(n) + g_d(n) + h_s(n) + h_d(n)$$

We also tie-break equal $f$ values on higher $g_s$ values.

After determining $g'$, we then perform the static learning step described in section 4.2.1 followed by the dynamic world learning step described in 4.2.3. These two steps are performed to learn a more informed $h$ value for the nodes in the local search space. This is done because our original $h$ value for a node we expanded could have actually been an underestimate of its true $h$ value. Now by performing Dijkstra's algorithm from the frontier back through the local search space we are learning a more accurate $h_s$ and $h_d$ values as Dijkstra's algorithm is calculating the cheapest path back from frontier to each node in the LSS. The algorithm then takes the first action along the path from $s$ to $g'$ before repeating this process. We need to learn $h_s$ values and cache them so that we are able to escape local minima or heuristic depressions that may be encountered during the search due to the static world. In section 4.2.1 we proved that our $h_s$ values will never decrease during the successive searches. This ensures that if using an admissible heuristic, our heuristic will remain admissible, yet become more informed as subsequent search iterations are performed.

More details are found within Koening and Sun's paper (Koenig and Sun, 2009).

Coming up with accurate heuristics for predicting the cost-to-go due to dynamic obstacles is a hard problem that, to our knowledge, has not been addressed in the literature. Thus we use the trivial $h_d$= 0 in our implementation. While this is very weak, we can improve it drastically during the search using our dynamic learning step. This is another key advantage of this technique over the competing methods: because we track dynamic and static costs separately, we can learn $h_d$ values through our $g_d$ costs.

In our implementation we use Algorithm 3 to calculate our $h_d$ values, with one additional tweak. We also use the form of decay over a pose, discussed in section 4.3.4. That is, we store a secondary cache which maps a pose to a list of triples containing the state's time, the planning iteration it was cached in and its $h_d$ value. If we get a hit in our main cache using our state as the key, we use the cached $h_d$ value. Otherwise, if we do not get a hit in our main cache, we strip the time from the state to get its pose, then check our secondary cache. If we get a hit in this secondary cache we use the time stamps of each triple to find the $h_d$ value closest to our state in time. We then use its $h_d$ value and the search iteration it was stored in to determine what the decayed $h_d$ value for our search node should be.

### 4.5.1 Possible Extensions

Several extensions of these techniques are possible:

**1.** One could disable dynamic learning when not near dynamic obstacles as well as in situations where the nodes expanded in the LSS and along the frontier have no dynamic cost associated with them.

**2.** It seems that there should be some way of doing both the dynamic learning and the static learning in one pass. The techniques we've considered include the following:

- Sort on lowest combined $h_s + h_d$. However, this does not guarantee that either the static or the dynamic $h$ portion are the minimum of the given nodes successors. This means we can be learning greatly inaccurate $h_s$ and $h_d$ values.

- Sorting on lowest $h_s$ obviously will not result in the $h_d$ values being visited in the lowest to highest order either.
- It also follows that sorting on lowest $h_d$ values will not yield the $h_s$ values in the correct order.

It is possible that better solutions to this problem exist, but currently we simply perform the static and dynamic learning separately.

# CHAPTER 5

# EXPERIMENTS

We performed an empirical analysis over a number of different instances in our problem domain. For each instance of the problem, we ran our new algorithm as well as other current state-of-the-art algorithms in motion planning and real-time search. A example of an instance is shown in figure 5-1. All real-time algorithms were given expansion bounds to allow them to return a solution within the time bound. Time-Bounded Lattice was set to use the parameters shown in their paper: a max *timebound* of 4 seconds. They did not specify their weights, however, so we used a number of weights as documented below. In addition to our random run instances, we ran on specific hand crafted scenarios which were designed to show desirable behavior in specific situations. The analysis on these scenarios are more visual than cost based, essentially answering the question "is what the robot did in this situation reasonable and intelligent looking?".

All experiments are run on our compute servers which are Dell Optiplex 960's each featuring a Core2 duo E8500 3.16 GHz processor and 8GB of RAM. The simulator, as well as each algorithm, are implemented in Ocaml 3.12. All experiments performed use $T_a$ and $T_p$ of 0.5. With cell discretization of 4cm per grid-cell. They all use a motion model with 16 distinct headings and 4 speeds: 1.5m/s backwards, stopped, 1.5m/s forward and 3.0m/s forward. The expansion limits are denoted in the figures as *lh* for lookahead. For the Time-Bounded Lattice, the timebound is shown as *tb*, and the weights are denoted as $w$. A linear decay technique is used by PLRTA* and the number of steps before a cached $h_d$ value becomes completely decayed is listed as *ds*. In each experiment one machine ran

both the simulator and the planner. Being dual core machines, there should not have been much thrashing.

The opponent model used simply looks at the previous two locations of a dynamic obstacle and linearly interpolates it out eight time-steps into the future, assuming it will maintain its current speed and heading. This is a fairly weak opponent model, which can obviously be improved, yet for the purposes of our experiments it serves well.

The implementation of Time-Bounded Lattice has one modification made to it. As the algorithm is proposed it will terminate when it expands the goal node. Because our experiments run for a fixed amount of time, not until the agent reaches the goal, we had to modify it so it would continue to do search after reaching the goal. Therefore, if the agent is starting on the goal state it will perform one additional expansion and choose the lowest $f$ child to move to. Otherwise, the algorithm operates as proposed.

## 5.1 Random Runs

In each instance of the random runs, there are $n$ opponents, each of which is performing a hand traced path. There is also one "intelligent" bot, which is running the algorithm under test. Each experiment lasts 60 seconds. The algorithm being tested is unaware that the experiment will last 60 seconds and is only given the information specified in section 2.1.2. The intelligent bot's goal at each time step is to take the best looking action, not necessarily the action which will minimize their cost within the 60 second window. The world is 20 meters by 20 in the random runs.

There are a set of 36 pairs of randomly selected start and goal states for each of the $n$ opponents. This gives us 36 instances times $n$ opponents, which in our case we run from 0 to 10 opponents (11 opponent matchings) or 396 different instances to solve per algorithm. Each 396 instances are the same for each algorithm so the only variable in the experiment is the algorithm being tested.

Figure 5-2 through 5-6 show the actual cost incurred by each algorithm over the 11

different opponent matchings when using a 2D Dijkstra heuristic discussed in section 2.5.2. Each box plot is over the 36 instances. Box plots work by displaying: 1) the sample minimum as the horizontal line below the box, 2) the lower quartile as the lower horizontal line forming the box, 3) the median as the line splitting the box, 4) the upper quartile as the top line of the box, 5) the sample maximum as the top line in the plot, 6) Outliers as dots outside of the range of the minimum and maximum. Outliers are simply data points that deviate from the sample greatly.

As you can see, with no opponents in the world, all algorithms do fairly well, the Time-Bounded Lattice technique with a timebound of 4000ms (4s) fairing the best, but not by much. This is because of the accuracy of the heuristic in our test map. Time-Bounded Lattice can quickly switch to relying solely on the heuristic (in this case a 2D Dijkstra) and follow it greedily to the goal. The story changes, however, once the number of dynamic obstacles in the world begins to grow. We can see that PLRTA* consistently and convincingly beats out all other algorithms.

LSS-LRTA* does comparably to PLRTA* until around 4 opponents at which point the two algorithms begin to really separate themselves in terms of performance. I attribute this to PLRTA* being able to utilize much better $h$ values in the form of the partitioned heuristic discussed in section 4.1. PLRTA* can tell much earlier on in the search if a path with yield a dynamic collision due to its ability to learn $h_d$ values properly.

We've also benchmarked the interesting algorithms using the straight line heuristic discussed in section 2.5.2. The results are shown in figure 5-7 through 5-11. It comes as a bit of a surprise that overall the algorithms seem to perform better using the straight line heuristic. This is computed as the straight line distance between the agent and the goal, divided by the maximum forward speed of the agent. Although the Dijkstra heuristic is more informed, the straight line still yields lower costs. There are even fewer collisions for PLRTA* when using this heuristic.

It should be noted that in all of these experiments, no matter the number of dynamic obstacles, collisions for PLRTA* were always outliers in our results. Never did the max

sample, ignoring outliers, cost more than a collision. This means it was a extraordinary condition for the PLRTA* algorithm to take an action resulting in a collision. On the other hand, all of the other algorithms tested had collision sample points within their interquartile range. This means it was not an extraordinary condition for these algorithms to take an action resulting in a collision. This is a very promising result as the opponent model used in all of these experiments is fairly weak.

Figures 5-12 through 5-14 show the number of nodes expanded during an entire experiment, over all 36 instances for the specified number of opponents. As is evident, PLRTA* does a constant amount of work in each search iteration. With a lookahead of only 1000 expansions per search iteration, we are able to do very well relative to the other algorithms. It should be noted that lsslrta, that is the original version of LSS-LRTA* as it is proposed, also does a constant number of expansions each search iteration. As you can see, however, Time-Bounded Lattice must do more and more work per planning iteration as the number of dynamic obstacles scale, leading to non-real-time response times. The median nodes expanded between 6 opponents and 10 opponents rose by around 100,000.

This is also a very positive result, as although Time-Bounded Lattice must perform a great deal more work each time it plans for the next action, we still come up with lower cost plans. It is unclear whether this was the result of time bounded lattice being run over by an opponent while planning due to missing the time window to send the next action to take or if this occurs while it is moving about the world.

From the results, one can see that PLRTA* performs fairly consistently despite the rise in the number of opponents. This speaks well to the method's scalability. Of course, there is a point where the algorithm will need to reduce its *lookahead* further as it will not be able to return an action in time due to the increase of dynamic obstacles in the world. This is because our cost function is not greatly optimized and makes $n$ checks each time a node is generated to determine the cost of a given cell, where $n$ is the number of opponents. Some optimizations may be made to reduce this issue, however, we have not pursued them due to time constraints.

### 5.1.1 Isolating Enhancements

We are able to show the affect of our different enhancements through our experiments. LSS-LRTA*is the base version of the algorithm, with none of our enhancements. In the plots we have shown our performance with all enhancements enabled. As it does not make sense to use heuristic decay without partitioning the heuristics, because the decay only affects $h_d$ values, we did not benchmark that configuration. However, we did benchmark our algorithm while varying the number of planning iterations that must pass before the value is considered completely decayed.

We ran all of these random runs with different decay settings to isolate the effect of the decay step. We used decay steps of 1, 2, 4 and $\infty$. A decay steps setting of $\infty$ essentially results in no decay, that is once a $h_d$ value is cached for a state, it never decreases. The results were clear: the decay setting did not have an effect on either the planned or actual cost of the plans found. This was a surprising result. We instrumented the code and thus, we know the values were being decayed with any setting other than $\infty$. However, upon further reflection, it is reasonable that these changes did not have an effect on this set of problems. This is because there are no situations in which the planner *must* go through a portion of the graph previously thought to be of high cost. This is because there are many paths to all the goal configurations and thus, the planner was never in a situation that would force it to decay its values to find a path to the goal. Also, because of the sheer size of our state space, there are a large number of very cheap paths to the goal. It appears this technique will be most useful when 1) there are a small number of paths to the goal, 2) these paths to the goal are blocked by dynamic obstacles driving up the cost of the paths 3) the dynamic obstacles then move away from the area previously thought to have high cost, allowing the decay technique to quickly lower these values back down, thus, allowing the agent to proceed to the goal.

This means that the partitioned heuristic learning was responsible for the increase in performance over that of LSS-LRTA*. This is a result strongly supporting the use of

partitioned heuristics and partitioned learning in this domain.

## 5.2 Hand Crafted Scenarios

We ran our handcrafted scenarios on a number of the algorithms discussed in this thesis. The scenarios we used are shown in figure 5-15, numbered from 1 to 6 starting in the upper right corner and going left to right in each row. Each scenario lasts only 30 seconds. The static environments tested in the scenarios are smaller than those of the random runs. Figures 5.2 and 5.2 show the results of all these runs.

These figures show not only the actual cost incurred and the number of expansions performed in the runs, but also a qualitative assessment of how "intelligent" each agent looks while acting in each specific scenario. This is qualified with three different assessments: good, ok and bad. As we can see in figure 5.2, not all those plans that have low cost are necessarily determined to look good. As a human observer, it is hard to always understand why the agent is behaving in a certain way. For example, in Scenario 1, the Time-Bounded Lattice algorithm freezes numerous times, as it takes too long to compute the action to take. Even once unimpeded paths to the goal are present, it sometimes takes multiple planning phases to pass before an action to take is returned. Also, PLRTA* oscillates back and forth between plans while moving to the goal, giving it a look of indecisiveness.

In Scenario 2, the Time-Bounded Lattice finds the long path around the static obstacle and reaches the goal fairly quickly, although it does freeze a few times along the way. LSS-LRTA* never makes it around the static obstacle and instead moves indecisively around the starting area. PLRTA* finds the path around the static obstacle and reaches the goal quickly, yet struggles in trying to arrange itself perfectly on the goal state.

In Scenario 3, the Time-Bounded Lattice agent fails to move off of the goal, even though a dynamic obstacle was known to be coming towards it. This is again because of the fact that Time-Bounded Lattice was designed to run until it expands the goal during the search. Thus, it was not entirely clear how to convert this into an algorithm which plans beyond

the goal. We stated earlier that, we simply expand one node if the agent begins on the goal state and move to the child with the lowest $f$. This is clearly not enough lookahead for the agent to escape and as such, it decides to continue sitting on the goal. Clearly, this is not a desirable result.

PLRTA* really shines in Scenario 3, as it waits on the goal as long as it can before moving out of the way, letting the opponent pass, and then returning back to the goal. LSS-LRTA* moves out of the way on this scenario as well, yet never returns to the goal afterward.

Overall, we've tried to summarize the performance in these scenarios by looking at the accumulated totals. Figure 5.2 shows these. Obviously, the cost of the Time-Bounded Lattice's performance in Scenario 3 skews these results. Ignoring them, however, you can see they did not fair all that better than PLRTA* or LSS-LRTA*. Also of note, is the significant amount of additional work Time-Bounded Lattice has to perform in terms of nodes expanded to achieve these costs. The Time-Bounded Lattice with a weight of 1.0 does nearly 10 times as many expansions as PLRTA*, even though it only does one expansion per planning iteration once it reaches the goal.

The overall performance qualitatively, is shown in figure 5.2. These assessments are also made through human judgement. The Time-Bounded Lattice agents works well in most cases yet cannot deal with the situation of needing to leave their goal location. This resulted in a collision in Scenario 3. LSS-LRTA* performs the worst overall despite never colliding with any obstacles. This is because it made a large number of seemingly unintelligent moves is most scenarios. PLRTA* performs the best, never colliding with dynamic obstacles, and coming up with reasonable looking plans.

This is a positive result as even though PLRTA* is only doing a limited amount of lookahead search, it is still able to react well to the dynamic obstacles around it and find intelligent looking plans to reach the goal.

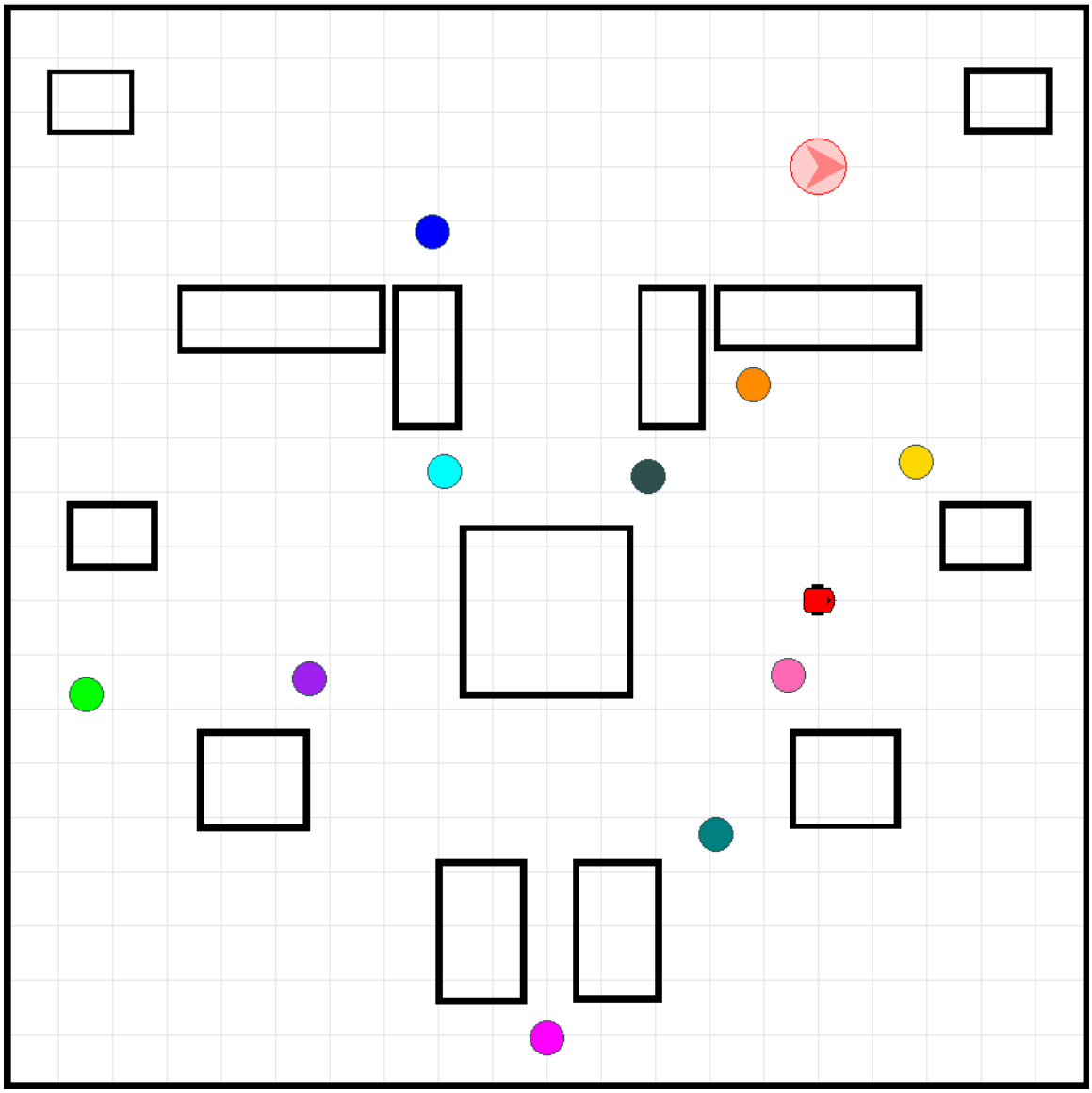

Figure 5-1: Example instance with 10 opponents. The goal area and heading are denoted by the red circle and arrow. The robot running the algorithm under testing is the red bot with wheels.

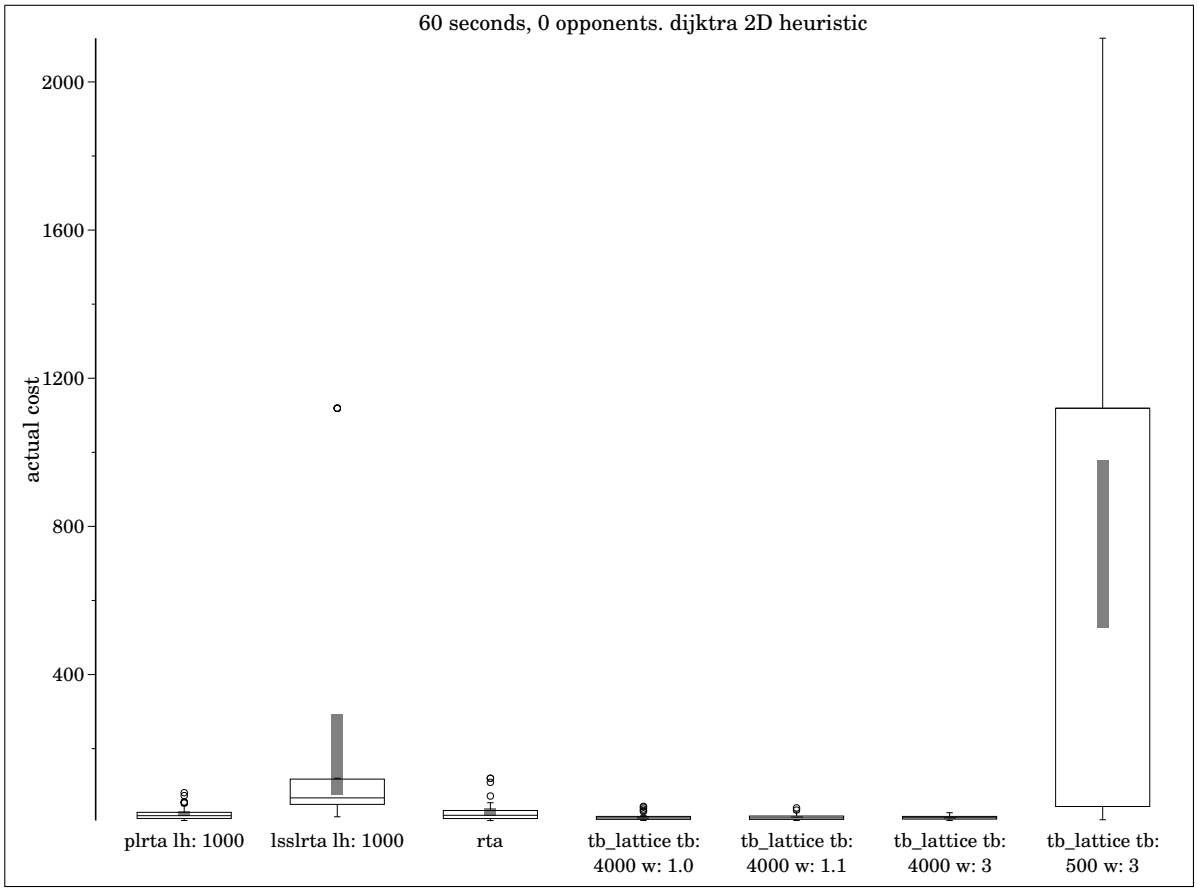


Figure 5-2: Actual cost incurred per algorithm with 0 opponents in the world over 36 different start/goal pairings

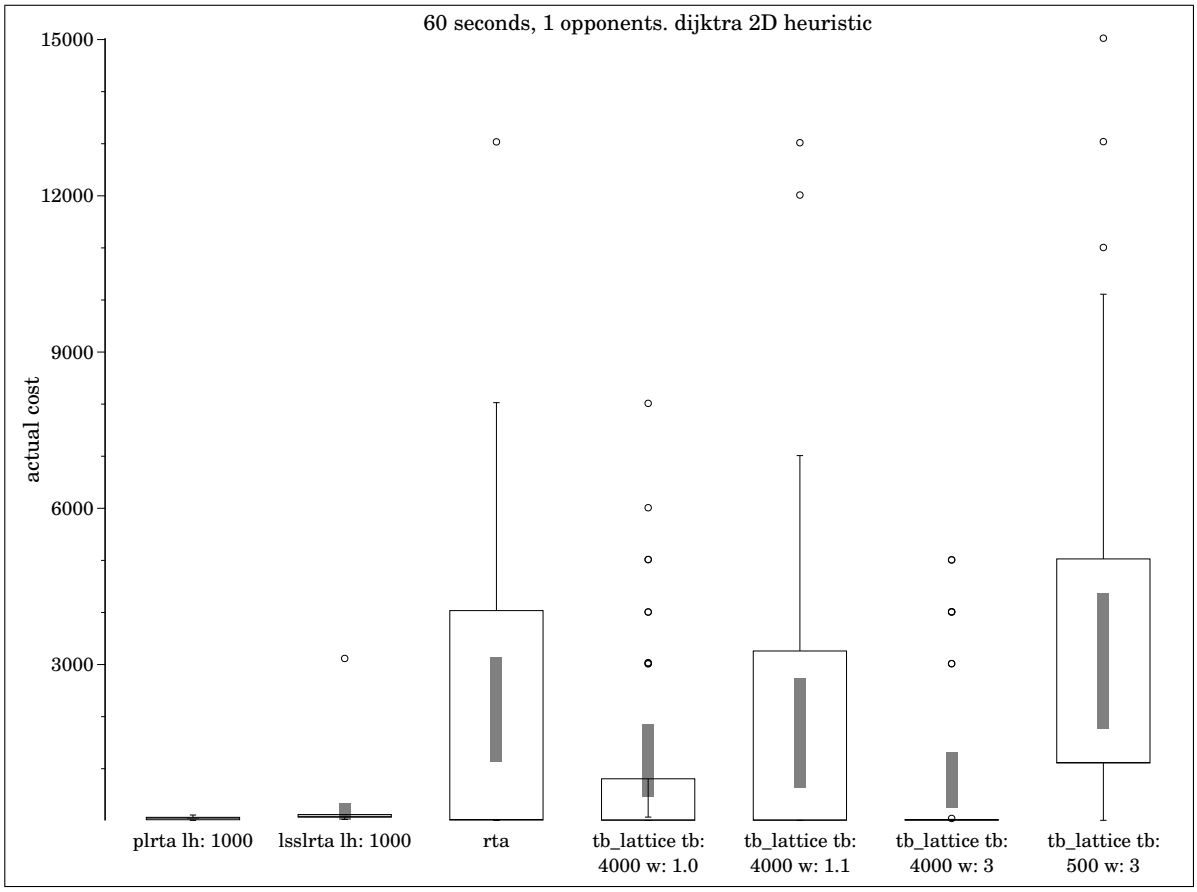


Figure 5-3: Actual cost incurred per algorithm with 1 opponents in the world over 36 different start/goal pairings

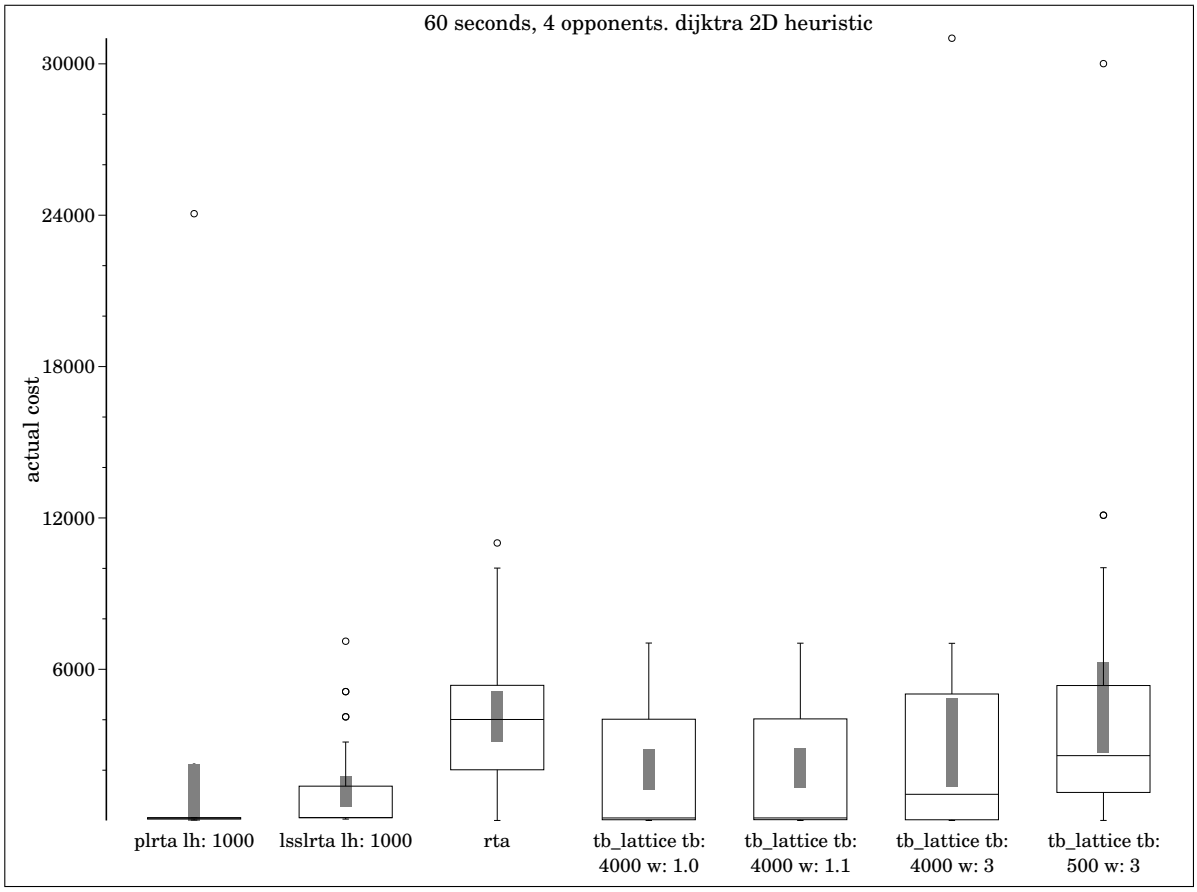


Figure 5-4: Actual cost incurred per algorithm with 4 opponents in the world over 36 different start/goal pairings

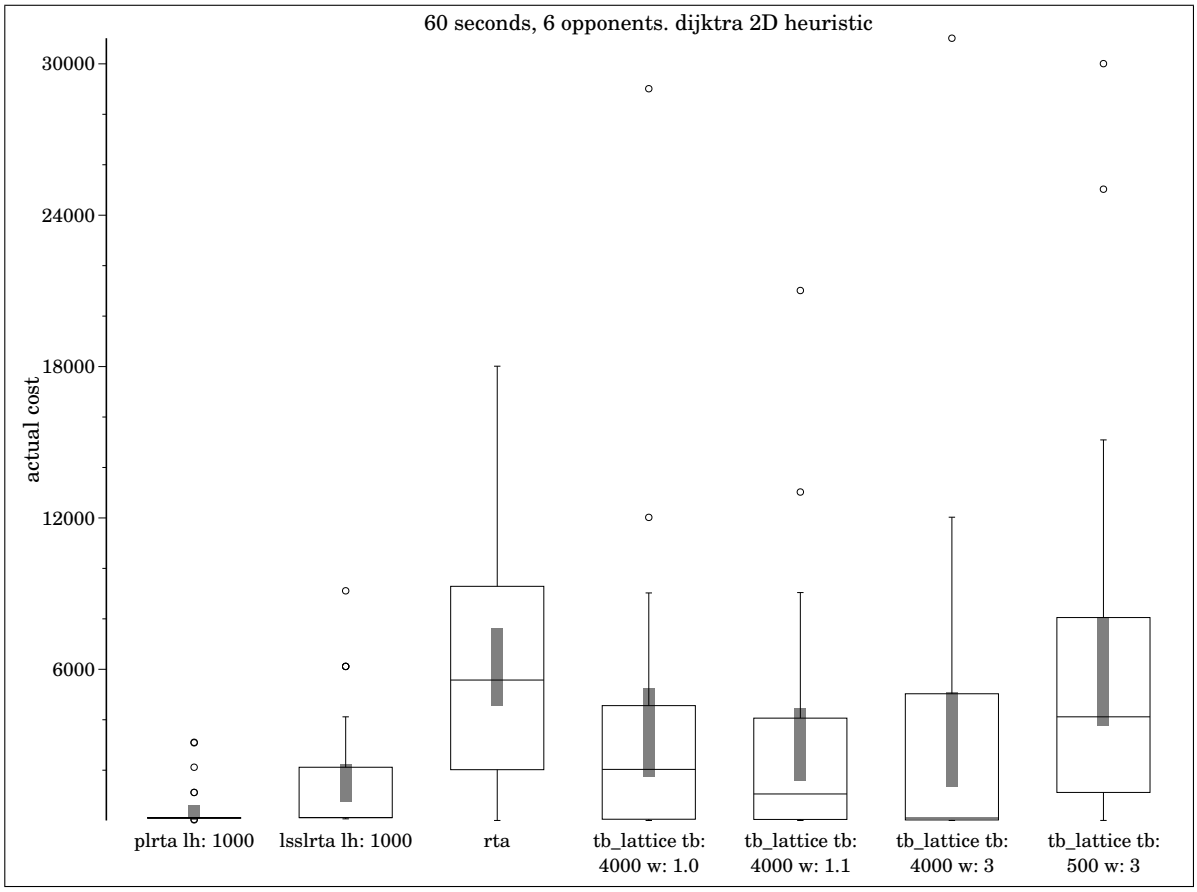


Figure 5-5: Actual cost incurred per algorithm with 6 opponents in the world over 36 different start/goal pairings

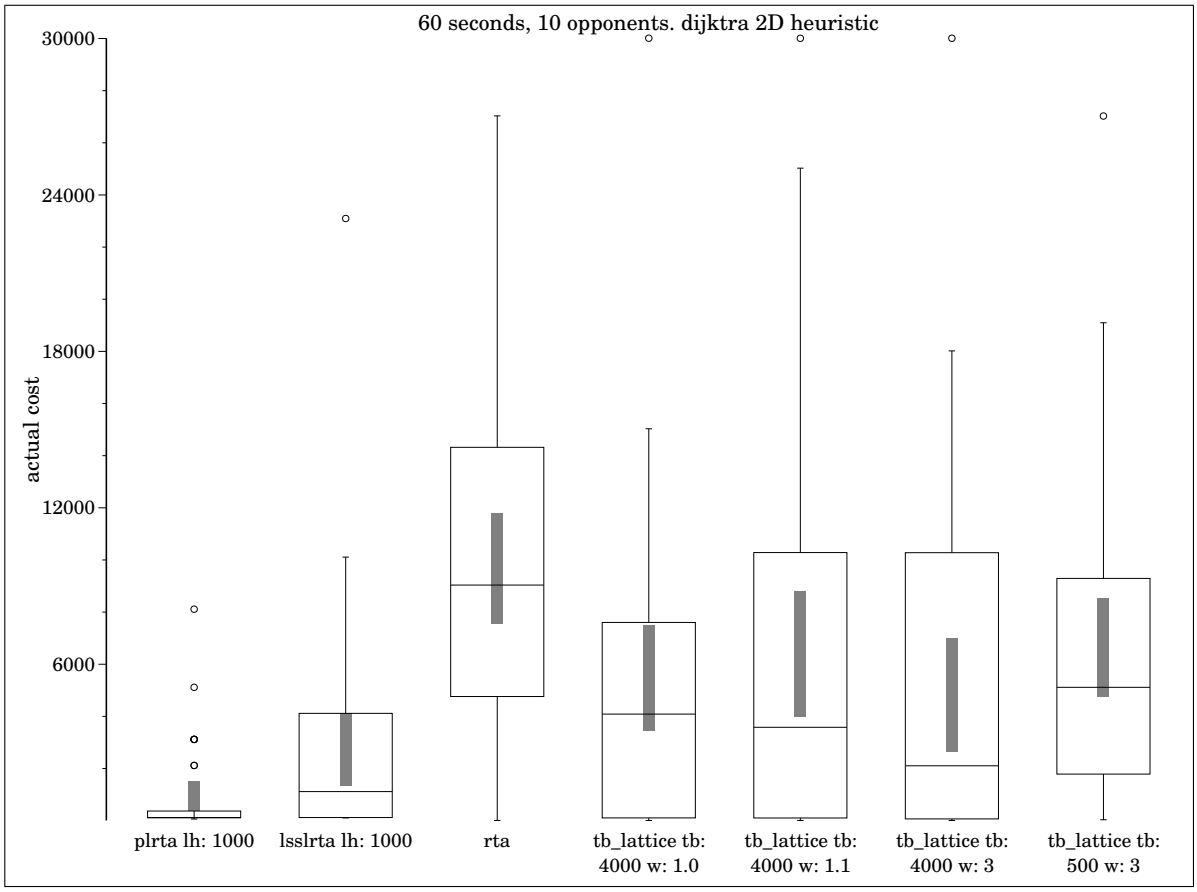


Figure 5-6: Actual cost incurred per algorithm with 10 opponents in the world over 36 different start/goal pairings

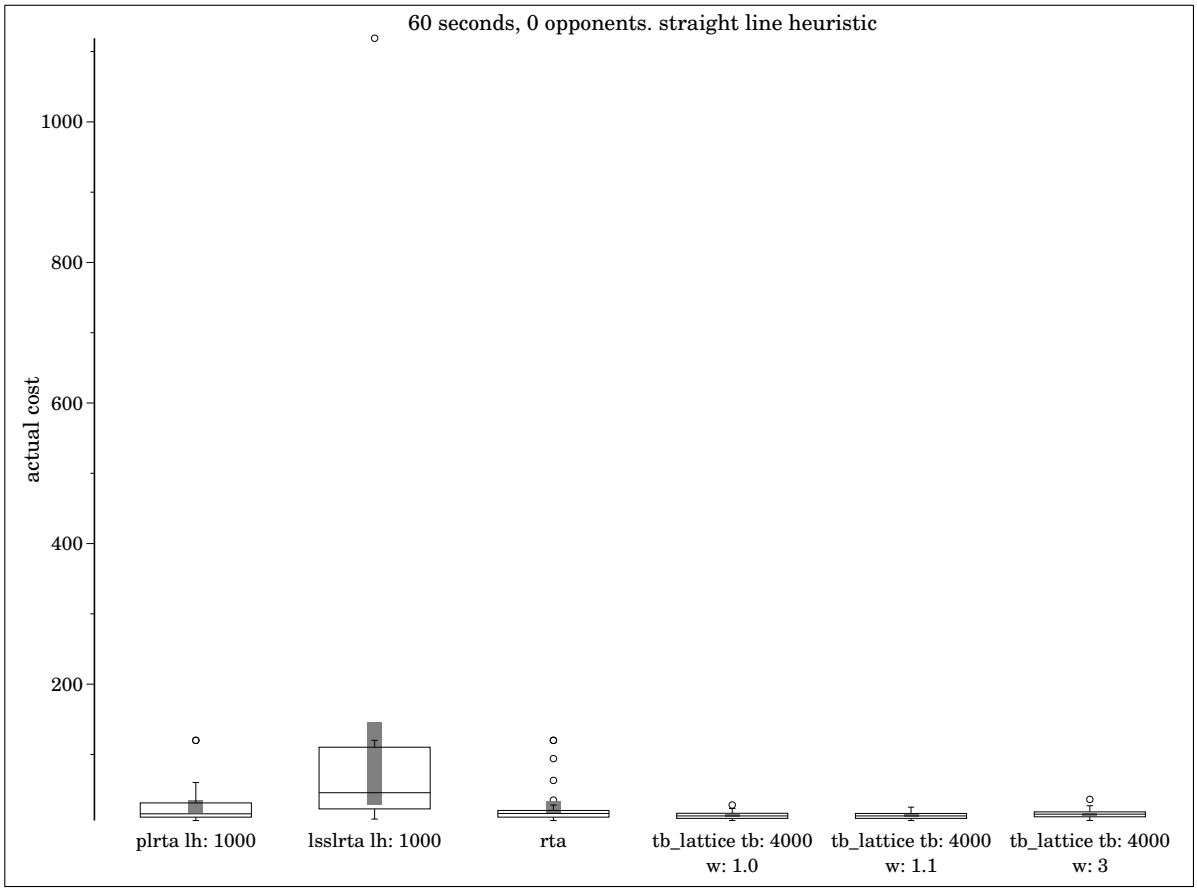


Figure 5-7: Actual cost incurred per algorithm with 0 opponents in the world over 36 different start/goal pairings

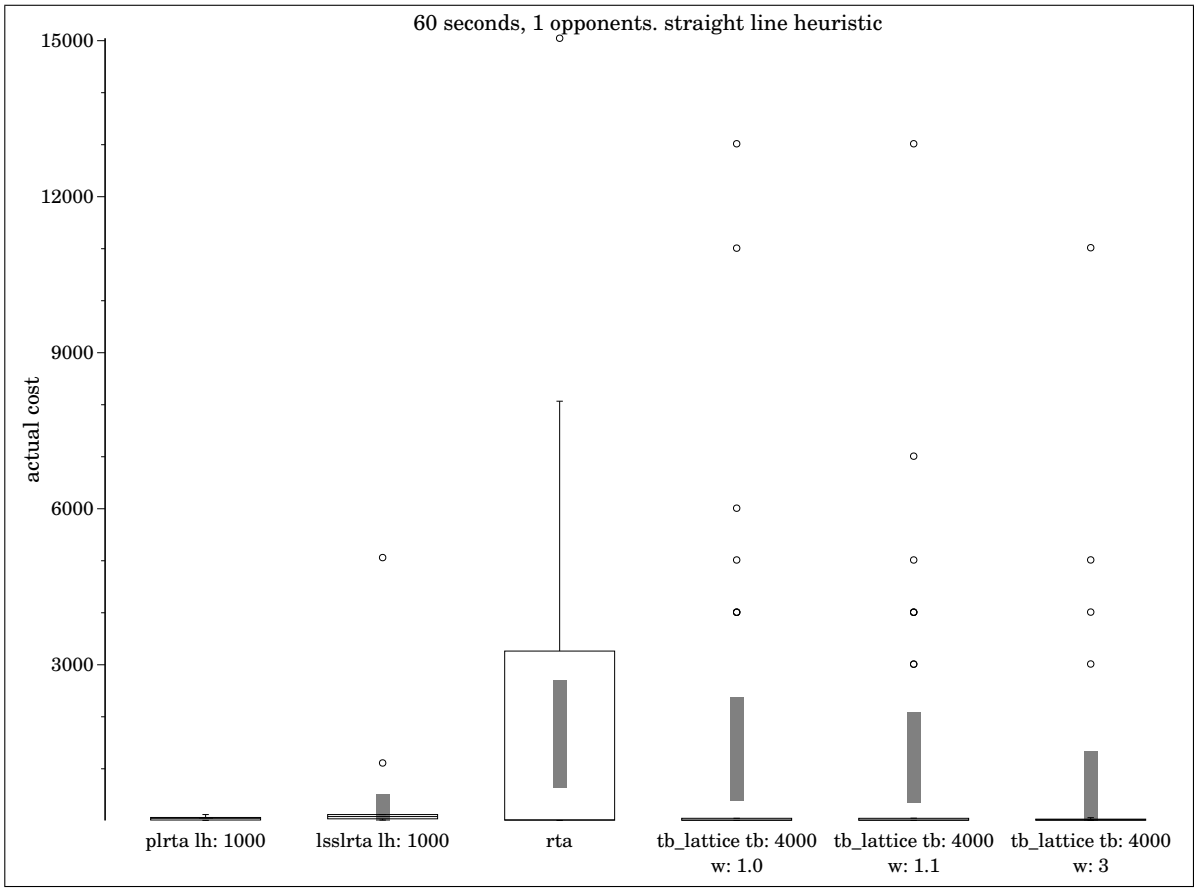


Figure 5-8: Actual cost incurred per algorithm with 1 opponents in the world over 36 different start/goal pairings

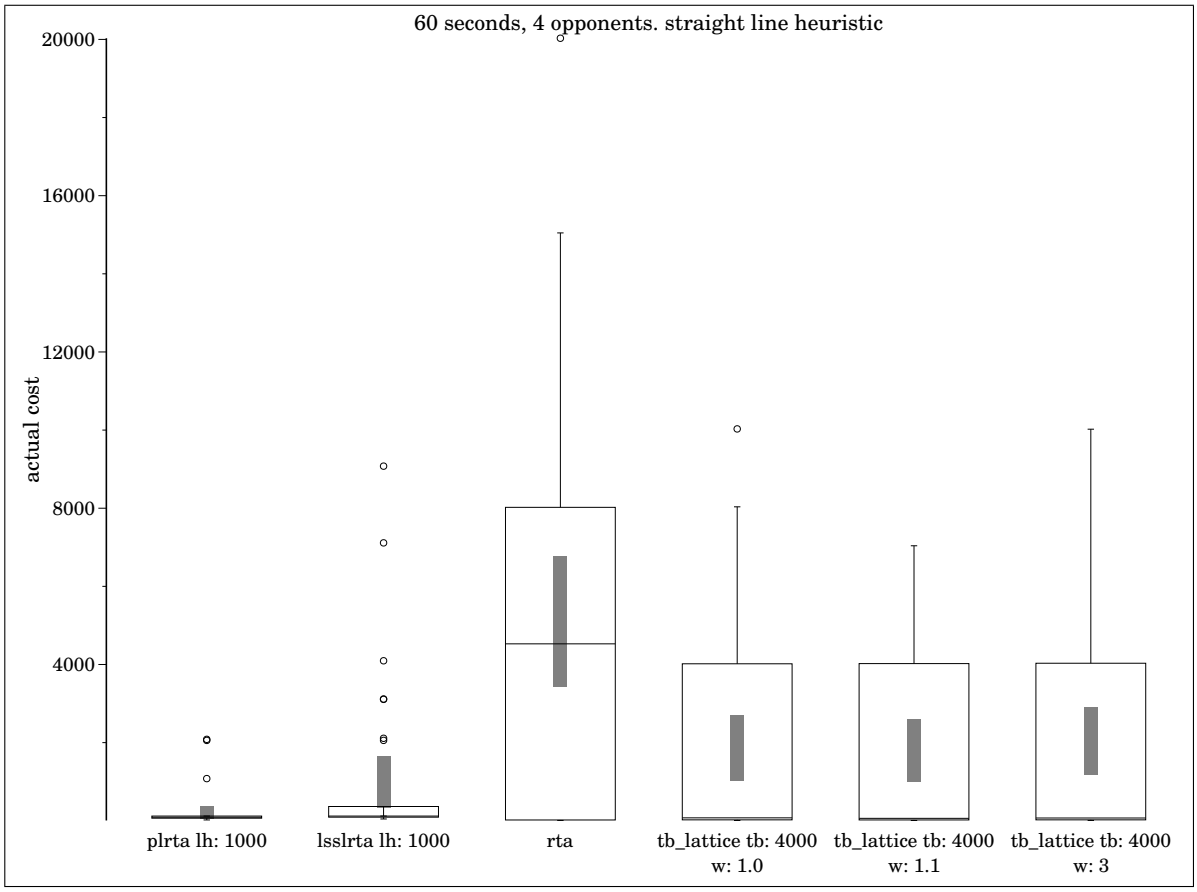


Figure 5-9: Actual cost incurred per algorithm with 4 opponents in the world over 36 different start/goal pairings

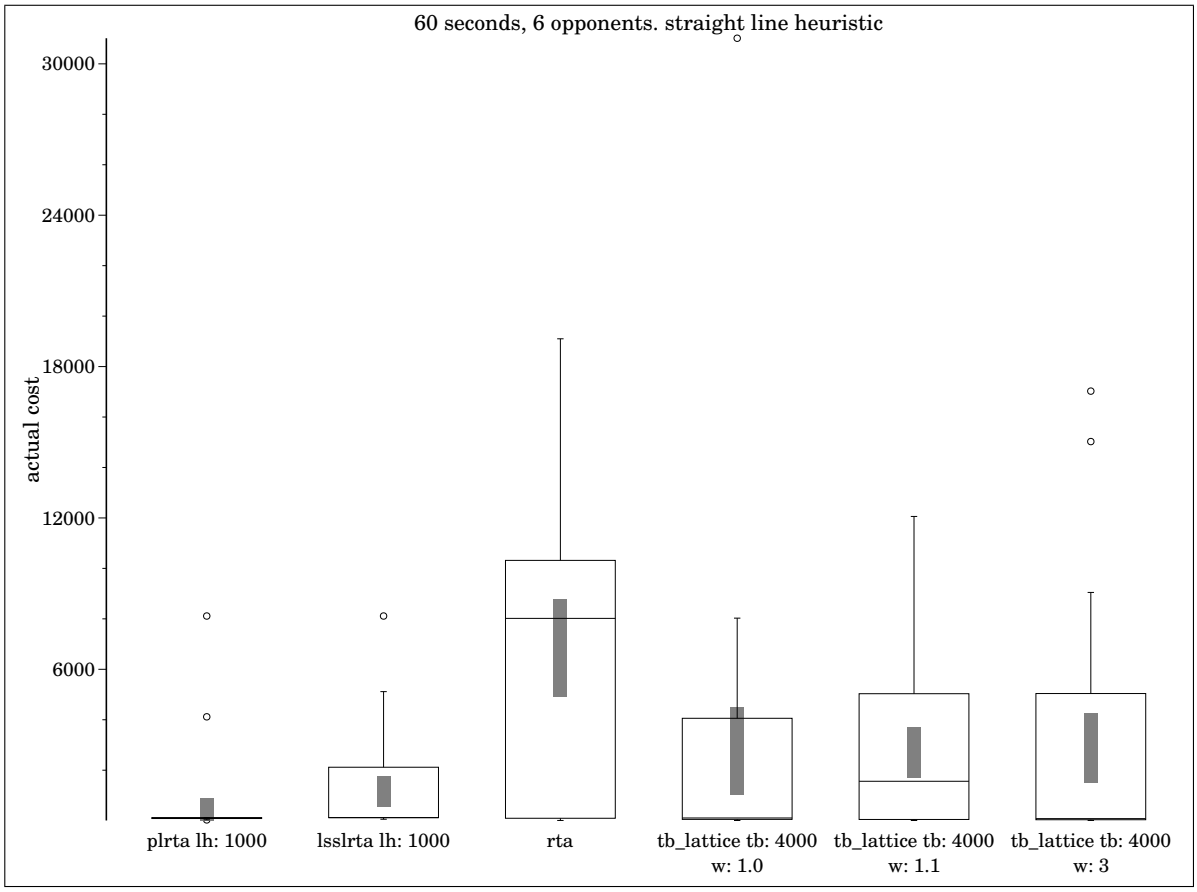


Figure 5-10: Actual cost incurred per algorithm with 6 opponents in the world over 36 different start/goal pairings

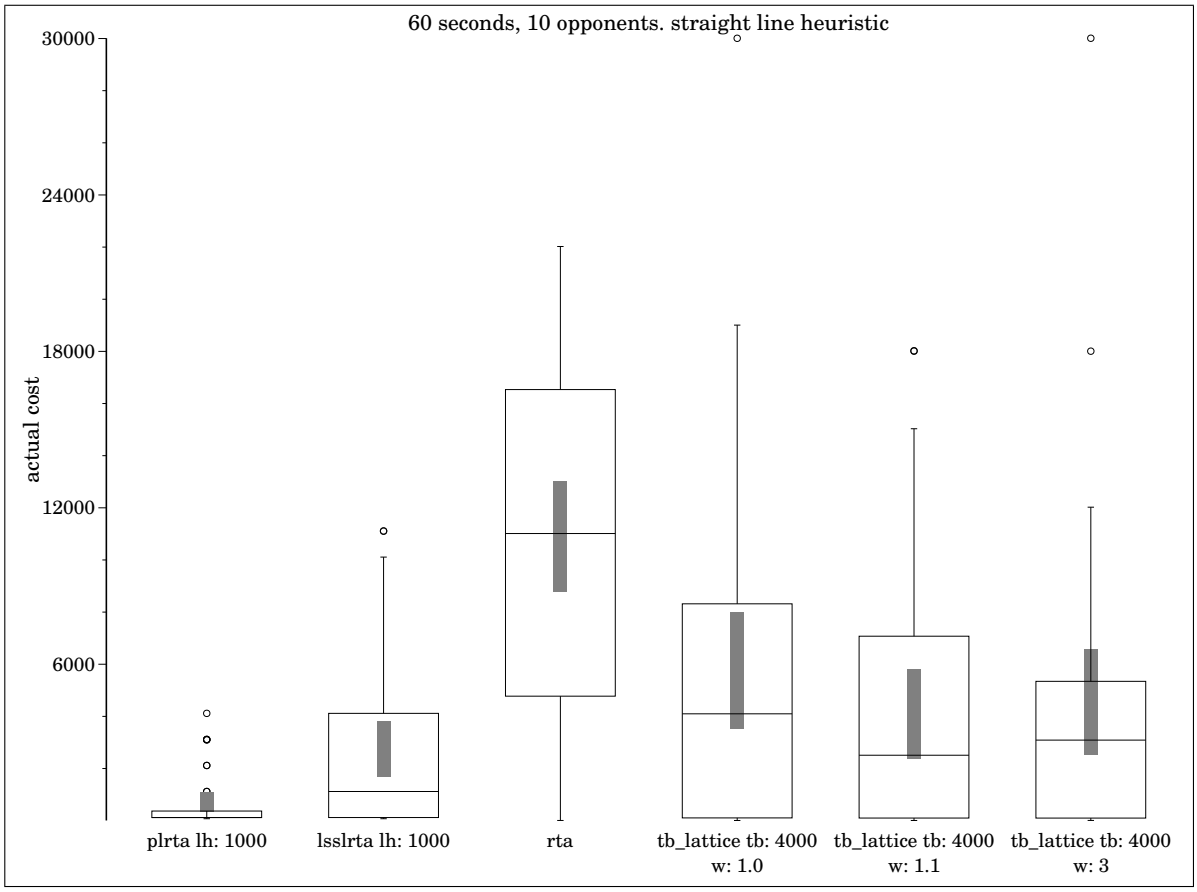


Figure 5-11: Actual cost incurred per algorithm with 10 opponents in the world over 36 different start/goal pairings

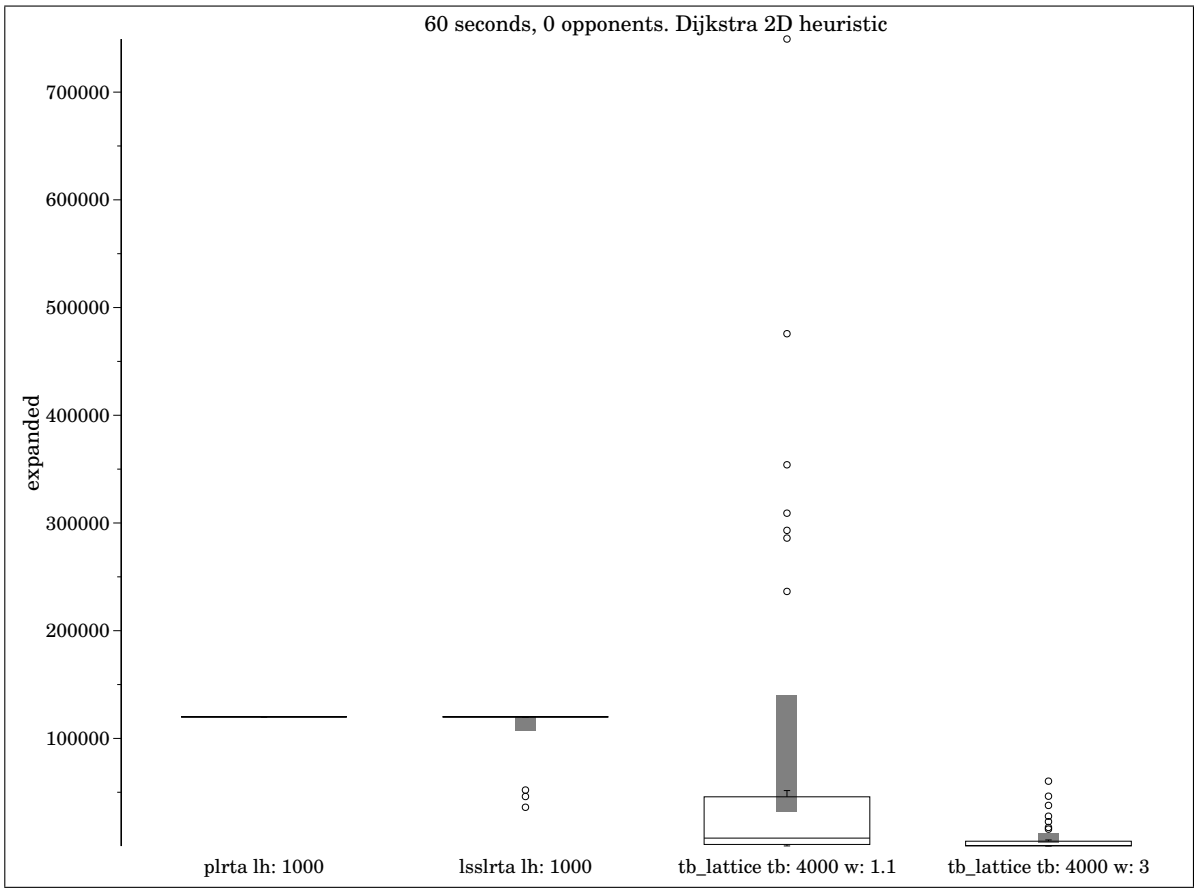


Figure 5-12: Number of nodes expanded during each iteration of the 36 different start/goal pairings with 0 opponents

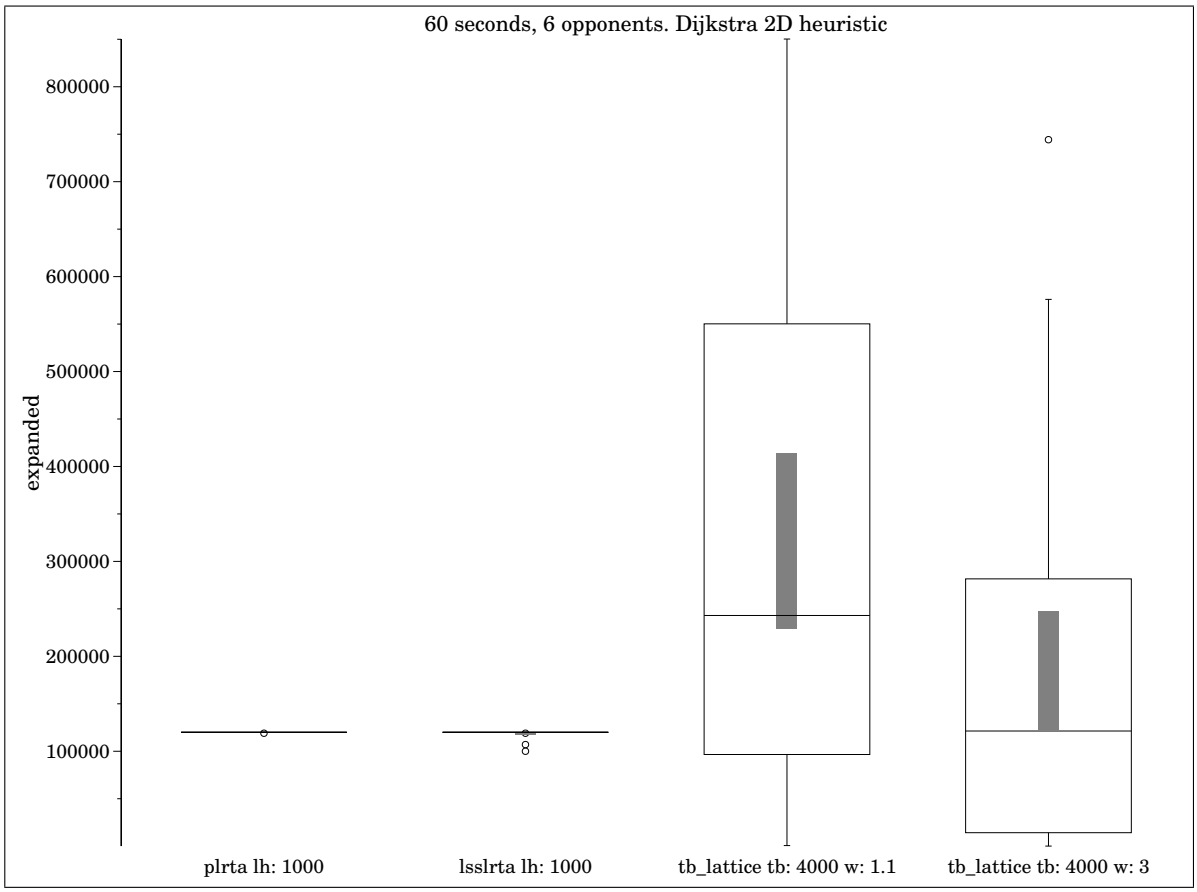


Figure 5-13: Number of nodes expanded during each iteration of the 36 different start/goal pairings with 6 opponents

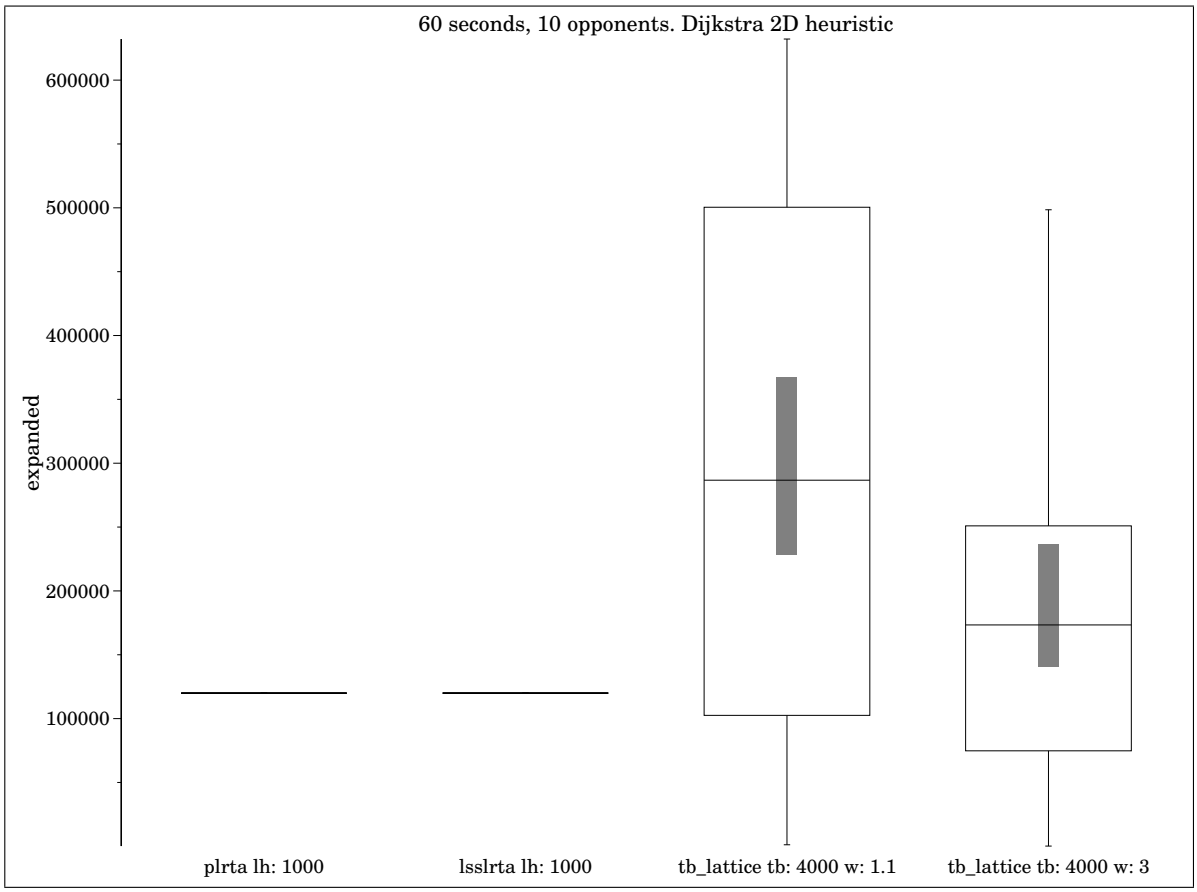


Figure 5-14: Number of nodes expanded during each iteration of the 36 different start/goal pairings

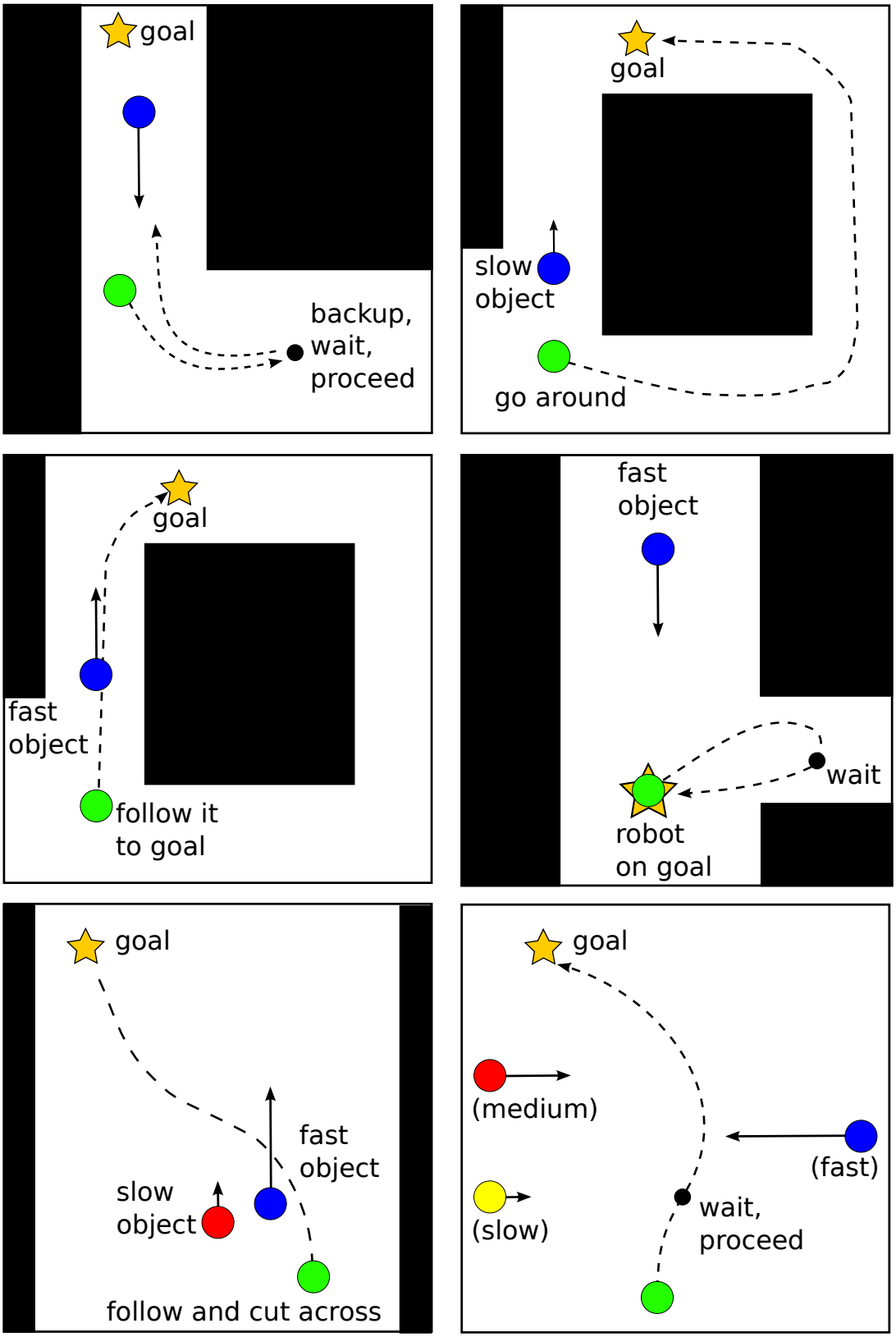


Figure 5-15: Hand crafted scenarios

Figure 5-16: Results of Scenarios 1 - 3

| Scenario 1 | | | |
|---|---|---|---|
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 29 | 272349 | ok |
| tb_lattice tb:4000 w:1.1 | 30 | 290356 | ok |
| plrta lh: 1000 | 37 | 60000 | ok |
| lsslrta lh: 1000 | 60 | 60000 | bad |
| Scenario 2 | | | |
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 30 | 1420197 | good |
| tb_lattice tb:4000 w:1.1 | 41 | 137395 | ok |
| plrta lh: 1000 | 60 | 60000 | ok |
| lsslrta lh: 1000 | 60 | 60000 | bad |
| Scenario 3 | | | |
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 60 | 797715 | bad |
| tb_lattice tb:4000 w:1.1 | 60 | 669209 | bad |
| plrta lh: 1000 | 32 | 60000 | good |
| lsslrta lh: 1000 | 60 | 54003 | bad |

Figure 5-17: Results of Scenarios 4 - 6

| Scenario 4 | | | |
|---|---|---|---|
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 3000 | 60 | bad |
| tb_lattice tb:4000 w:1.1 | 3000 | 60 | bad |
| plrta lh: 1000 | 26 | 60000 | good |
| lsslrta lh: 1000 | 54 | 60000 | ok |
| Scenario 5 | | | |
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 22 | 115827 | good |
| tb_lattice tb:4000 w:1.1 | 22 | 123933 | good |
| plrta lh: 1000 | 60 | 60000 | bad |
| lsslrta lh: 1000 | 60 | 60000 | bad |
| Scenario 6 | | | |
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 60 | 651882 | bad |
| tb_lattice tb:4000 w:1.1 | 60 | 600860 | bad |
| plrta lh: 1000 | 41 | 60000 | ok |
| lsslrta lh: 1000 | 46 | 60000 | ok |

Figure 5-18: Totals of all over all the Scenarios

| Totals | | | |
|---|---|---|---|
| Algorithm | Actual Cost | Nodes Expanded | Look |
| tb_lattice tb:4000 w:1.0 | 3201 | 3258030 | ok |
| tb_lattice tb:4000 w:1.1 | 3213 | 1821813 | ok |
| plrta lh: 1000 | 256 | 360000 | good |
| lsslrta lh: 1000 | 340 | 354003 | bad |

# CHAPTER 6

# CONCLUSION

In this thesis, we have introduced three new techniques to address some of the issues in the current state-of-the-art algorithms in robot motion planning. These techniques are a partitioned heuristic, heuristic decay and a garbage collection technique for dealing with unnecessary states. We also introduced a new algorithm, Partitioned Learning Real-Time Search (PLRTA*), which we believe to be the new state-of-the-art in real-time algorithms that must deal with dynamic obstacles.

PLRTA* is based on LSS-LRTA*, yet improves it markedly by using all of our new techniques introduced in this thesis. We extensively benchmarked our algorithm in the domain of real-time robot motion planning with dynamic obstacles and compared its results to the current state-of-the-art real-time and non-real-time algorithms. In these experiments, we showed that PLRTA* outperforms the current state-of-the-art substantially in terms of minimizing cost when there are larger numbers of dynamic obstacles in the world. Because we adopt a real-time technique, it is also shown that we do a constant amount of work during each planning phase to determine the next action to take, whereas the non-real-time techniques must scale the amount of work they do with the number of dynamic obstacles in the world.

As far as we know, we are the first to feature partitioned heuristics for tracking the dynamic and static costs in the world separately and to use a novel decaying technique to both generalize heuristic estimates over poses in the world independent of time, as well as for maintaining correctness.

## 6.1 Future Work

This section overviews some possible future work to improve the techniques introduced in this thesis.

### 6.1.1 More Efficient Partitioned Learning

Our current technique of partitioned learning is to first sort the open list from the $A^*$ search on lowest $h_s$ and backup learned $h_s$ values in a Dijkstra like manner. We then resort the original $A^*$ list on lowest $h_d$ and backup the learned $h_d$ values in the same way. It seems as though there must be a more efficient way to perform these operations. This would lead to more time for the A* search if discovered which could lead to even better performance.

### 6.1.2 Non A*-based Lookahead Searches

PLRTA* uses an A* lookahead to determine its Local Search Space. Because we are not looking for optimal solutions, it seems as though there may be better ways to form an LSS during the search portion of the planning stage.

### 6.1.3 More Principled Decay Techniques

The decay technique was shown to have no effect on our search in the instances we tested due to the large number of alternative paths that can be taken in our domain. This technique still seems as though it may be useful if used in other ways. Further work could be done in investigating how to better utilize the decay technique, possibly not only generalizing a $h_d$ value over pose, but maybe even more generally, such as an $x, y$ position or some radius around an $x, y$ position.

Also, we currently use a simple linear decay technique to reduce our cached $h_d$ values down to 0 before they are thrown out and removed from the cache. Other obvious techniques for performing the decay include using an exponential decay rate and dynamically varying the amount of decay for a given cached state depending on how predictably the dynamic

obstacle which caused the dynamic cost is moving. If it is moving erratically we may want to decay the cached value more quickly than if it is moving predictably, as the value is likely to become inaccurate much quicker. This would require additional tracking per cached $h_d$ value and will add additional time overhead.

### 6.1.4 Inadmissible $g$ Values

Due to our cost function which changes through time, we have inadmissible $g$ values in our domain. As far as we can tell, inadmissible $g$ values have not been explored in the literature. This would suggest that this problem may be an entirely new type of graph search problem. The technique I've devised, separates out the admissible from the inadmissible portions of the $g$ value, allowing us to do search while maintaining provable properties of completeness. Although, the technique is simple, it is easy to understand and works well in practice. More research must be performed to really understand what affect inadmissble $g$ vlues have on our state space.

# APPENDICES

# APPENDIX A

# Communication Protocol

All messages are sent in ASCII and must be terminated with a newline character ('\n').

## A.1 Initialization:

### A.1.1 Agent to Simulator

- **hello**: This is the first command sent to the coordinator. This is used to check communication channels.
- **ready**: This is sent as a response to the *init* command from the coordinator.

### A.1.2 Simulator to Agent

- **init name time move-cost collision-cost radius map-res motion-prim-file algorithm alg-params domain-params gx gy goal-deltas rows cols static-obstacles**: This is sent as an initialization command.
  - *init* the string "init".
  - *name* String. This is a space delimited string representing the name of the robot being controlled.
  - *time* Float in seconds. The amount of time given for each planning cycle.
  - *move-cost* Float. The cost of moving in the world.
  - *collision-cost* Float. The cost of colliding with an obstacle.

- *radius* Float in meters. The radius of the robot.
- *map-res* Float in meters/pixel. representing the resolution of each cell of the map.
- *motion-prim-file* This a path to the motion primitive file the planner will use.
- *algorithm* A string name of the algorithm to be used for planning.
- *alg-params* Key Value string pairs separated by spaces and terminated by a newline of algorithm specific parameters.
- *domain-params* A series of parameters for the domain as a series of strings terminated by a newline.
- *goalx goaly goalh goalv goalw* x,y in meters. h in degrees. w in degrees per second. All floats. The goal location for the robot.
- *goal-deltas* The deltas allowed around the goal to still be considered on the goal. These are in terms of a radius a difference in degrees and a difference in rotational velocity all as float.
- *rows* Int. The number of rows in the world grid.
- *cols* Int. The number of columns in the world grid.
- *static-obstacles* are the locations of the static obstacles in the world. They have been expanded by the corresponding robots radius already and are supplied as a $rows * cols$ length string of ones and zeroes. There are no spaces between the ones and zeroes.

## A.2 Operation:

### A.2.1 Simulator to Agent

- **state time goal num-dyn-obstacles dyn-obstacles**: This message tells the agent what the state they are currently in, the projected trajectories of the dynamic obstacles

and the goal location. Note: for the state and goal part of the message, the x and y values are in meters. The heading is in degrees. Speed is in m/s and rotational speed is deg/s.

- *state* is made up of five string labels each followed by a float value for that label, i.e. "x 5.6 y 7.65 h .002 v 1.0 w 1.2". Note: w is still sent even though we do not use it. Just read it and ignore it.
- *time* this is the simulation time. It is made up of the string "time" followed by a float representing the time in seconds. I.e. "time 3.5".
- *goal* is made up of five string labels each followed by a float value for that label, i.e. "x 5.6 y 7.65 h .002 v 1.0 w 1.2". Note: w is still sent even though we do not use it. Just read it and ignore it.
- *num-dyn-obstacles* Is made up of a label followed by and int i.e. "num-dyn-obstacles 4".
- *dyn-obstacles* A series of *num-dyn-obstacles* dynamic obstacles. None of these fields have labels and are each space delimited. They are sent as follows:
  - *radius* Float in meters.
  - *time-delta* Int in milliseconds. The time that each Gaussian is valid for.
  - *base* This is a series of five floats each with a space character between them. They are in the following order. x,y,stddevx,stddevy,r. Where x,y is the center of the gaussian. stddevx and stddevy are the standard deviation in the x and y coordinates, and r is the correlation.
  - *deltas* This is a series of five floats each with a space character between them. They are in the following order. $x_d, y_d, stddevx_d, stddevy_d, r_d$. These are the *deltas* for each respective field. x,y are the difference between each step in the gaussians. that is if the values of a gaussian at time $i$ were $x_i, y_i, sddevx_i, sddevy_i, r_i$ the values at the next time step would be: $x_i + x_d, y_i + y_d, sddevx_i + stddevx_d, sddevy_i + stddevy_d, r_i + r_d$

- **endsim**: The string "endsim". This message signals the end of the simulation, the agents should then exit. No other messages will be sent or handled after this is sent.

### A.2.2 Agent to Simulator

- **action**: This is sent back to the controller.
  - *action* A serialized version of the motion primitives.

# APPENDIX B

# Configuration File Specification

Shown here is the schema for our configuration file. Each field is required the the specific order show. The type is specified after the field name, as well as an example value.

```
bitmap string simulator/models/bitmaps/empty.pnm
world-x float 30.0
world-y float 30.0
world-z float 5.0
px_res float 45.0
cost_res float 4.0
framerate float 15.0
floor-color int 0xFFFFFF
obstacle-color int 0x000000
sim-iterations int 50
plan-time float 0.4
action-time float 0.5
move-cost float 1.0
collision-cost float 1000.0
goal-delta-radius float 0.5
goal-delta-v float 0.0
goal-delta-h float 0.0
goal-delta-w float 0.0
```

```
num-robots int 1


name string bot_0
host string localhost
alg-type string realtime
motion-prim-file string /home/path/to/motion/primitives
algorithm string lsslrta*
alg-params string lookahead 20
domain-params string sh dijkstra
command string  /robot_simulator/agent.Unix
rgb int 0xff0000
radius float 0.3
height float 2.0
start string diff_drive_state 2.0 5.0 0.0 0.0 0.0
goal string diff_drive_state 14.0 15.0 0.0 0.0 0.0
```

# APPENDIX C

# Division of Labor

The work performed to construct the simulator to run our experiments was done as a joint effort between Kevin Rose and I. Kevin dealt mainly with the underlying search domain for the problem, including the motion model and other domain specific features. He also implemented the graphical front-end for our simulator. I dealt mainly with the actual running of the simulation: tracking the state of the world and statistics, as well as the communication between the simulator and the planners.

All other work presented in this thesis are the result of my research, including all of the algorithms demonstrated. I also implemented all of the algorithms which I test in Chapter 5.

# BIBLIOGRAPHY


Bond, D.; Widger, N.; Ruml, W.; and Sun, X. 2010. Real-Time Search in Dynamic Worlds.

Hart, P.; Nilsson, N.; and 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.

Koenig, S., and Likhachev, M. 2002. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, 476–483. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Koenig, S., and Likhachev, M. 2006. Real-Time Adaptive A*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, 281–288. ACM.

Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

Kopriva, S.; Sislak, D.; Pavlicek, D.; and Pechoucek, M. 2010. Iterative accelerated A* path planning. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, 1201–1206. IEEE.

Korf, R. 1990. Real-time heuristic search. *Artificial intelligence* 42(2-3):189–211.

Kushleyev, A., and Likhachev, M. 2009. Time-bounded lattice for efficient planning in dynamic environments. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, 1662–1668. IEEE.

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

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems* 16.

Phillips, M., and Likhachev, M. 2011. SIPP: Safe Interval Path Planning for Dynamic Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Rose, K. 2011. Real-Time Sampling-Based Motion Planning with Dynamic Obstacles. In *M.S. Thesis, U.N.H. 2011*.

Šišlák, D.; Volf, P.; and Pěchouček, M. 2009. Accelerated A* path planning. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 1133–1134. International Foundation for Autonomous Agents and Multiagent Systems.

Snape, J.; Guy, S.; and van den Berg, J. 2010. Independent navigation of multiple robots and virtual agents. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 1645–1646. International Foundation for Autonomous Agents and Multiagent Systems.

Urmson, C.; Anhalt, J.; Bagnell, D.; Baker, C.; Bittner, R.; Clark, M.; Dolan, J.; Duggins, D.; Galatali, T.; Geyer, C.; et al. 2008. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics* 25(8):425–466.

Van Den Berg, J.; Stilman, M.; Kuffner, J.; Lin, M.; and Manocha, D. 2009. Path planning among movable obstacles: a probabilistically complete approach. *Algorithmic Foundation of Robotics VIII* 599–614.