Winter 2011

Real-time sampling-based motion planning with dynamic obstacles

Kevin Rose
University of New Hampshire, Durham

REAL-TIME SAMPLING-BASED MOTION PLANNING WITH DYNAMIC OBSTACLES

BY

Kevin Rose

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.

[illegible]

Thesis director, Wheeler Ruml,
Assistant Professor of Computer Science

[illegible]

Philip Hatcher,
Professor of Computer Science

[illegible]

Radim Bartoš,
Associate Professor of Computer Science

[illegible]

Michel Charpentier,
Associate Professor of Computer Science

December 8, 2011

Date

ACKNOWLEDGMENTS

I would like to sincerely thank my thesis advisor, Professor Wheeler Ruml, for all the advice and insight he has given me throughout my years at the University of New Hampshire. I would also like to thank Jarad Cannon who worked closely with me throughout this entire process. In addition, I would like to thank the members of the UNH AI Research group for their insightful comments and questions along the way. Lastly, I would like to thank my friends and family for their support through the many sleepless nights involved with preparing this thesis. This research was funded in part by the DAPRA CSSG program (grant D11AP00242) and the NSF RI program (grant IIS-0812141).

CONTENTS

LIST OF FIGURES

ABSTRACT

REAL-TIME SAMPLING-BASED MOTION PLANNING WITH DYNAMIC OBSTACLES

by

Kevin Rose
University of New Hampshire, December, 2011

Autonomous robots are increasingly becoming incorporated in everyday human activities, and this trend does not show any signs of slowing down. One task that autonomous robots will need to reliably perform among humans and other dynamic objects is motion planning. That is, to reliably navigate a robot to a desired pose as quickly as possible while minimizing the probability of colliding with other objects. This involves not only planning around the predicted future trajectories of dynamic obstacles, but doing so in a real-time manner so that the robot can remain reactive to its surroundings. Current methods do not directly address this problem. This thesis proposes a new real-time planning algorithm called real-time R* (RTR*). RTR* is based on the R* search algorithm that couples random sampling with heuristic search and has been shown to work well in several different robotics domains. Several modifications needed to transform R* into a real-time algorithm are described. Additional modifications that were developed specifically for this problem domain are also detailed. An empirical evaluation is given comparing RTR* with several state-of-the-art motion planning and real-time search algorithms. RTR* shows promising performance and improves on R*, however it underperforms the current state-of-the-art. Several enhancements are discussed that could improve the behavior of RTR*.

CHAPTER 1

INTRODUCTION

Autonomous robots are increasingly becoming incorporated in everyday human activities, and this trend does not show any signs of slowing down. One task that autonomous robots will need to reliably perform among humans and other dynamic objects is motion planning. That is, to reliably navigate a robot to a desired pose as quickly as possible while minimizing the probability of colliding with other objects. This involves not only planning around the predicted future trajectories of dynamic obstacles, but doing so in a real-time manner so that the robot can remain reactive to its surroundings. Current methods do not directly address this problem. This thesis proposes a new real-time planning algorithm called real-time R* (RTR*). RTR* is based on the R* search algorithm that couples random sampling with heuristic search and has been shown to work well in several different robotics domains. Several modifications needed to transform R* into a real-time algorithm are described. Additional modifications that were developed specifically for this problem domain are also detailed. An empirical evaluation is given comparing RTR* with several state-of-the-art motion planning and real-time search algorithms. RTR* shows promising performance and improves on R*, however it underperforms the current state-of-the-art. Several enhancements are discussed that could improve the behavior of RTR*.

1.1 Problem Description

The type of problem that is specifically dealt with in this thesis is the problem of motion planning for a wheeled, differential drive ground vehicle from its current pose to some goal pose while minimizing the probability of collision with static and moving obstacles. This is a reasonable and most likely necessary task that autonomous robots interacting in close proximity to humans and other moving obstacles must be able to accomplish. There has been much research devoted to robot motion planning, but little that specifically addresses the problem at hand. The robot must be able to navigate through moving obstacles, taking into account their past observed states, their likely future states, and the uncertainty associated with these predictions, all in a real-time system that is capable of adjusting to its surroundings several times per second. In addition, the plans returned must be feasible with respect to the kinematic and dynamic constraints of the mobile robot.

This thesis addresses the problem of successfully navigating a robot amongst moving and static obstacles from an initial world state to some goal state. The trajectories and future positions of the dynamic obstacles are unknown to the robot. The robot must be able to react to changes in the trajectories of dynamic obstacles in real-time. It must also be able to plan around the estimated future locations of obstacles. The robot must be able to not only arrive at the goal, but stay out of harm's way once it is there. This could mean moving back off the goal to allow an obstacle to pass. While there are many aspects that go into making a functional robot, such as controlling actuators, sensing, tracking, etc., this thesis is mainly concerned with the high-level planning algorithms used to dictate the robot's movement. The specific methods that the robot uses to gather information needed to plan and to then execute the plan are abstracted away in this work. It is assumed that the robot has some way of obtaining all the necessary information for planning. The planning algorithms discussed here will use this information to generate plans made up of sequences of actions that are feasible according to the physical model of the robot being used.

Although only one specific type of robot is considered in this thesis, this work may be extended to many other types of vehicles, given a sufficient model of their geometric properties and their kinematic and dynamic properties and constraints. There are also many ways to plan trajectories for mobile robots. This thesis either constrains or modifies the techniques used to only include ones that generate dynamically feasible trajectories.

That is, trajectories that the robot can execute from its current pose without violating any of its physical or dynamic constraints. This avoids the post-processing that must be done after planning to turn possibly infeasible trajectories into feasible ones.

1.2 Solving Methods

The general path planning problem of moving a polyhedron through Euclidean space while avoiding polyhedral obstacles has been shown to be PSPACE-hard [33]. Despite this complexity, several heuristic and sampling based approaches have been developed that can reliably solve these problems. This thesis will look at a mixture of two different approaches commonly used for motion planning in robotics and AI. One way that one might possibly solve this problem is through search. That is, take the robot's initial state and search over all the possible sequences of actions that the robot can take until a sequence is found that arrives at the goal. This method has been successfully used to do path planning for several robots [5, 6, 39]. Another popular approach is to use randomized sampling over the possible actions that the robot can take until a path is constructed that reaches the goal. Several randomized approaches have seen wide adoption in the robot planning community recently and have shown to be able to solve challenging problems [20, 22, 15].

With respect to this application, both types of approaches should be real-time. That is, there is a fixed amount of time that is available for planning, after which the planner must return an action to execute. The robot interleaves planning and execution as it moves throughout the world. The time alloted to planning must be short enough that the robot can respond quickly to any moving obstacles that pose a threat to it and long enough that it has time to select reasonable actions to take. Due to the hard time constraints imposed, the planning algorithms used must be modified to handle the likely scenario that they are unable to plan a complete path from the robot's current configuration to the goal in a single planning iteration. This thesis will discuss several real-time search and randomized algorithms currently used in robotics and AI, and present a hybrid real-time algorithm that uses techniques from both search and random sampling.

1.3 Outline

The rest of this thesis will proceed as follows:

- Chapter 2 will discuss the specifics of the domain that is being planned over. It will explain how states and actions are represented, as well as how moving obstacles are dealt with. It will also discuss some of the specific properties of the resulting search space that make this a difficult problem to solve.

- Chapter 3 will discuss the simulator that was built to evaluate the performance of different planning algorithms on this domain. The execution of the simulator and the specific evaluation criteria recorded will be discussed.

- Chapter 4 will provide an introduction to and overview of relevant past work in this field. Several different methods that have been used to solve the robot motion planning problem will be discussed as well as why they may or may not be suited for this domain.

- Chapter 5 will present a new algorithm, Real-time R* (RTR*) that incorporates ideas from search and random sampling into a real-time algorithm suitable for this problem. The changes needed to turn the original R* algorithm into a real-time algorithm will be discussed, as well as enhancements made specifically for this problem. This chapter will also include results from an empirical evaluation of RTR* and several other state-of-the-art algorithms.

- Chapter 6 will discuss insights made and conclusions drawn from the experiments done to evaluate the different algorithms tested. This chapter will also include several ideas that could be incorporated into RTR* to possibly improve performance.

CHAPTER 2

PROBLEM DOMAIN

In order to define the planning problem, we must first define the domain that is being planned over. The problem of planning kinodynamically feasible trajectories for a robot among moving and stationary obstacles consists of several parts that will be discussed in this chapter. The state space that is being planned over must be defined along with the specific variables that make up a robot's state. The different actions available to the robot and their specification must also be defined. The representation used for static and dynamic obstacles will be given along with the techniques used to predict the future positions of dynamic obstacles. Lastly, the cost function that is used to evaluate different paths through the state space will be defined. In reality, the problem of motion planning is continuous. The solution methods used in this thesis rely on discrete graph search, so the state and action space must be discretized to make the problem manageable. It is assumed that the locations of static obstacles are known perfectly. The current positions of the dynamic obstacles can also be observed perfectly, although the future positions of the dynamic obstacles are unknown. Lastly, it is assumed that the actions that the robot can take are completely deterministic. When executing an action from a given state, there can only be one resultant state.

2.1 State Space

The planning algorithms used in this thesis plan over the possible states that the robot may be in. At the very least, the robot's state must consist of its x and y location. Since the actions that a robot executes must be feasible, more state parameters may be needed.

In this thesis, a wheeled differential drive robot is used. To assure that the kinematic constraints of the robot are not violated, the robot's heading, θ, is incorporated into the robot's state. To assure that the robot's dynamic constraints are not violated, the speed of the robot, v, is also specified in the robot's state. Lastly, as the locations of the moving obstacles in the world are dependent on time, the time, t, must part of the robot's state. Since these variables are all continuous, they are discretized so that an infinite number of states need not be explored. It should be noted that it is possible to do motion planning with moving obstacles without including time in the state space representation. Dynamic obstacles are treated as static obstacles and are enlarged according to their current direction of movement. While this makes the planning problem easier, it also restricts the possible solutions the robot can produce, resulting in inefficient plans or even causing the planner to be unable to find a solution to the problem.

2.1.1 State Representation

The state of a robot is the 5-tuple $\langle x, y, \theta, v, t \rangle$ which consists of the robot's location, heading, speed, and the current time, respectively. In this thesis, the robot's rotational velocity is ignored in the interest of reducing the complexity of the problem. If the rotational velocity of the robot is very important to generating feasible actions, then it can be added to the state representation, at the expense of increasing the size of the state space. Since the planning must be done over a five dimensional state space, the number of possible states to explore is much greater than would be required in a lower dimensional space, $\langle x, y \rangle$ for example. This makes the problem of planning much more difficult since the size of the state space is exponential in the number of dimensions. The advantage of representing state to this degree is that plans can be generated that are guaranteed to be dynamically feasible for the particular robot. That is, they will not violate any of the robot's motion constraints. For example, planning over a state space consisting of just x and y location makes the assumption that the robot can turn in place and can accelerate instantaneously, which is probably not the case. The inclusion of time allows for plans that respect the predicted future locations of obstacles.

The state space is discretized considerably to cut down on the number of unique states that can be generated. In the experiments used in this thesis, the robot is limited to having four possible speeds, maximum forward, medium forward, stopped, and maximum reverse. The robot's heading is limited to one of 16 possible headings (22.5° intervals). For position, the robot's location is specified as centimeters. Lastly, time is discretized by setting the duration of all actions used to some constant number of milliseconds, t_a. These discretizations are still fairly accurate, but greatly reduce the state and action space needed to solve problems. Also, from an implementation perspective, the discretization allows the states and transitions between states to use integers and integer arithmetic, which is much more efficient than floating point arithmetic on most processors.

2.2 Action Space

The action space consists of the different actions that are available to the robot. In reality, the number of possible actions that are available to the robot is infinite, as they are over a continuous space. For planning, the actions used are discretized as in [26]. When searching, the possible actions that a robot can take at each state are constrained to a predefined set of short, dynamically feasible actions called motion primitives. The number of motion primitives that a robot is allowed to execute from a given state determines the branching factor of the corresponding node in the search graph, since it determines the number of successor states that there can be. Increasing the number of motions can result in better paths at the expense of more planning that must be done. The actual semantics of a motion primitive with duration t_a can be described mathematically by the function:

$$f(s,t) \rightarrow s', \quad 0 <= t <= t_a \tag{2.1}$$

where $s = \langle x_0, y_0, \theta_0, v_0, t_0 \rangle$ is the initial state of the robot and $s' = \langle x', y', \theta', v', t' \rangle$ is the instantaneous state of the robot while performing the motion primitive at time t', where $t' \leq t_a$. This function fully describes the robot's state for the duration of the motion primitive.

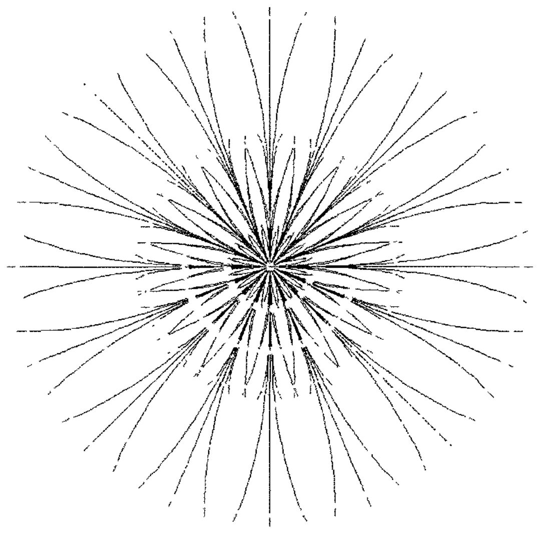


Figure 2-1: A complete set of motion primitives used for planning. There are three possible speeds and 16 possible headings. The yellow circles represent the end state of each motion primitive.

The implementation of motion primitives that is used does not specify the function $f(s,t)$ exactly. To allow for greater flexibility of the function, and to also handle cases where $f(s,t)$ can only be approximated, a motion primitive is specified as a collection of points approximately representing f over the range $[0,\ t_a]$.

2.2.1 Generating Motion Primitives

This section will discuss how to generate physically accurate motion primitives for a differential drive robot. A differential drive vehicle changes direction by altering the relative speeds of of its two wheels. To turn left, it sets the speed of its left wheel slower than that of its right wheel, and the opposite to turn right. The geometric model of a two wheeled differential drive robot with axle length b is shown in Figure 2-2. The possible headings and speeds that the robot can take when planning are constrained to be part of a finite set. The

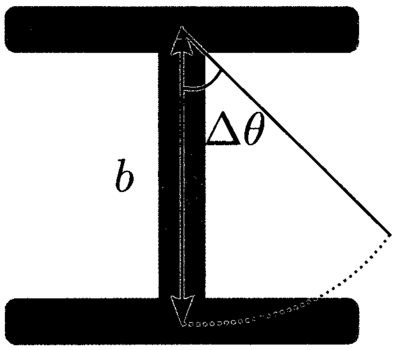


Figure 2-2: A two wheeled differential drive model where the axle length is b and heading is changed by $\Delta\theta$.

duration of each action is specified as t_a. When generating motion primitives, we must be able to specify the starting state of the robot and the robot's intermediate state at any point in time during the action so that the robot motion can be simulated and collision checking can be performed. We must also be able to specify the end state speed and heading that we wish the robot to be in after executing the action. In order to generate a physically and dynamically accurate motion primitive for a differential drive robot, we must be given the length of the robot's axle, b, the desired duration of the motion primitive, t_a, the initial and final speeds of the robot, v_0 and v_f, and the initial and final headings of the robot, θ_0 and θ_f. The starting location is assumed to be $(0,0)$, this way the motion primitive can be treated as an offset to any location. The goal is then to generate a motion primitive lasting for t_a seconds that changes speed by $\Delta v = v_f - v_0$ and changes heading by $\Delta\theta = \theta_f - \theta_0$. The equations of motion of the differential drive robot are as follows:

$$v = \frac{v_r + v_l}{2} \tag{2.2}$$

$$a = \frac{a_r + a_l}{2} \tag{2.3}$$

$$\omega = \frac{v_r - v_l}{b} \tag{2.4}$$

$$\alpha = \frac{a_r - a_l}{b} \tag{2.5}$$

where v is speed, a is translational acceleration, ω is rotational velocity, α is rotational acceleration, v_r and v_l are the right and left wheel velocities and a_r and a_l are the right and left wheel accelerations. The equations to calculate the translational and rotational acceleration are:

$$a = \frac{\Delta v}{t_a} \tag{2.6}$$

$$\alpha = \frac{2\,(\Delta\theta - \omega\, t_a)}{{t_a}^2} \tag{2.7}$$

where a and α are assumed to be constant throughout the motion primitive. We can now specify the robot's heading and speed as functions of time:

$$\theta(t) = \theta_0 + \omega t + \frac{1}{2}\alpha t^2 \tag{2.8}$$

$$v(t) = v_0 + at \tag{2.9}$$

To calculate the x and y position of the robot throughout the motion, we first must specify the x and y derivatives with respect to time:

$$\frac{dx}{dt} = (v_0 + at)\cos\left(\theta_0 + \omega t + \frac{1}{2}\alpha t^2\right) \tag{2.10}$$

$$\frac{dy}{dt} = (v_0 + at)\sin\left(\theta_0 + \omega t + \frac{1}{2}\alpha t^2\right) \tag{2.11}$$

These equations can't be integrated exactly to give the equations of position, $x(t)$ and $y(t)$, so they must be integrated numerically. For this, the GNU Scientific Library was used. We now have the equations $x(t)$, $y(t)$, $\theta(t)$, and $v(t)$, allowing us to fully specify the robot's state at any period in time throughout the motion primitive. To check that a motion primitive does no violate any speed or acceleration constraints of the robot, we must also be able to calculate the right and left wheel velocities and the right and left wheel accelerations. We can solve for the right and left wheel velocities, v_r and v_l, by simultaneously solving equations (2.2) and (2.4). This yields:

$$v_r = v + \frac{b\,\omega}{2} \tag{2.12}$$

$$v_l = 2v - v_r \tag{2.13}$$

We can also calculate the right and left wheel accelerations needed to achieve the desired change in heading, $\Delta\theta$, and change in speed, Δv. This can be achieved using equations (2.3), (2.5), (2.6), and (2.7) and solving for a_r and a_l. This yields:

$$a_r = \frac{\Delta v - \omega_0\, b}{t_a} + \frac{b\Delta\theta}{{t_a}^2} \tag{2.14}$$

$$a_l = 2\frac{\Delta v}{t_a} - a_r \tag{2.15}$$

The previous four equations allow the speed and acceleration of each wheel to be calculated to ensure that the motion primitive does not violate any constraints that the specific robot may have. We have now shown the equations of motion necessary to generate motion primitives for differential drive robots that are consistent with the physical model of the robot and to check that a given motion falls within the robot's capabilities.

2.3 Static Obstacles

The world may contain an arbitrary number of static obstacles. These are obstacles that cannot move, such as walls in an indoor environment, or trees and boulders in an outdoor environment. It as assumed that the locations and sizes of static obstacles are known a priori to the robot. While this may not be a valid assumption for a real robot, since planning is done in real-time, updates to the static obstacle map can be done between planning iterations, allowing the robot to react to changes in the static obstacle map in the next planning cycle.

Rather than store the exact position and shape of each static obstacle, a boolean grid representation of the map is stored instead. Each cell can either be an obstacle or free space. Prior to planning, all obstacles are expanded by the radius of the robot so that the robot can be represented as a single point in the static obstacle grid. This improves efficiency because computing whether or not the robot is in a blocked cell only requires calculating the grid cell that the robot's position falls in and looking up its value in the static obstacle matrix. In this thesis, the static obstacle grid is also used for duplicate checking during

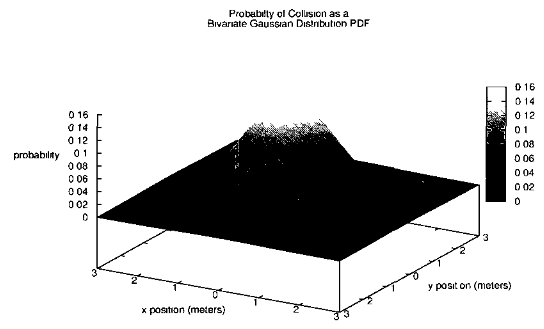


Figure 2-3: An example bivariate Gaussian probability density function.

planning. Two states that fall in the same static obstacle grid cell are treated as duplicates as long as they also have the same heading, speed, and time. The static obstacle grid does not need to be constrained to be of boolean type. Alternatively, the static obstacle grid could be treated as a real-valued cost map instead of a boolean map to capture different degrees of traversability. For example, a road could have a low cost associated with it and a rocky hillside could have very high cost associated with it. In this thesis, the boolean representation is used.

2.4 Dynamic Obstacles

A dynamic obstacle is any obstacle that is capable of moving. It is assumed that the robot can observe the current location of all dynamic obstacles, but it knows nothing about their future trajectories. Following [21], we represent the uncertainty associated with the movements of a dynamic obstacle as bivariate Gaussian probability distributions over x and y location. These distributions will logically represent the probability of an obstacle being at a given location at a certain time. Figure 2-3 shows a plot of the probability density function of the standard bivariate Gaussian distribution. The value of the z axis represents the probability that the dynamic obstacle is located at exactly some x and y position. Since the function is a probability, the volume under the curve sums to one. These distributions

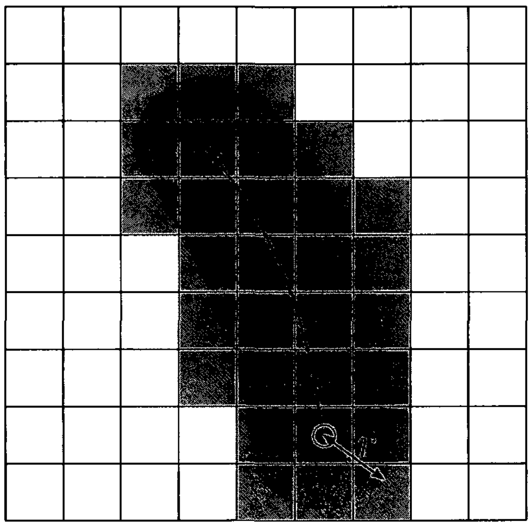


Figure 2-4: The cells used to approximate the cost of executing a motion primitive.

can be generated using past observations of the dynamic obstacle's position [21]. This will be discussed in more detail in the next section.

A bivariate Gaussian distribution over x and y can be specified by the five parameters: μ_x, μ_y, σ_x, σ_y, and ρ. These correspond to the mean x value, the mean y value, the standard deviation in x, the standard deviation in y, and the correlation of x and y, respectively. The probability density function (PDF) of the bivariate Gaussian distribution is defined as:

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left[-\frac{z}{2(1-\rho^2)}\right] \tag{2.16}$$

where

$$z \equiv \frac{(x-\mu_x)^2}{\sigma_x^2} - \frac{2\rho\,(x-\mu_x)\,(y-\mu_y)}{\sigma_x\sigma_y} + \frac{(y-\mu_y)^2}{\sigma_y^2} \tag{2.17}$$

The probability of the robot colliding with an obstacle is equal to the integral of (2.16) over the area of the robot expanded by the area of the obstacle. Assuming that the robot and the obstacle are circular with radii r_1 and r_2 respectively, the probability that the robot is in collision with the moving obstacle is:

$$F(x,y) = \int\limits_{y-r}^{y+r} \int\limits_{x-r}^{x+r} f(x,y)\,dx\,dy \tag{2.18}$$

where $r = r_1 + r_2$. Since this equation does not have a closed form solution, it must be approximated, as is the approach in [21]. The technique used here is to precompute, for each motion primitive used, all the grid cells that are within the radius r of any point along the path of the motion primitive. Figure 2-4 shows a motion primitive in red. It is expanded by the radius r shown by the gray area. The blue cells are those that intersect with the gray area and are the ones used to approximate the integral. The integral is approximated by computing the PDF using (2.16) at the center of each of the blue cells and then calculating the sum. That is to say, the approximated probability of colliding with a dynamic obstacle while executing a motion primitive is approximated as:

$$P(col) \approx \sum_{cells} f(x_c, y_c) \tag{2.19}$$

where *cells* are all the cells touched by the expanded motion primitive and x_c and y_c equal the center of a given cell. This approximation will be less than the true integral but was found to be accurate enough to capture the costs of dynamic obstacles during path planning. It should be noted that in reality, the integrals of the Gaussian distributions should be computed across space *and* time. Here, the Gaussians used to compute cost are assumed to remain stationary throughout the duration of the motion primitive. This assumption should be fine for motion primitives with short duration and greatly reduces the complexity of the cost computation.

2.4.1 Opponent Modeling

In order to plan around the predicted future locations of moving obstacles, the robot must create a time parameterized probability distribution of where it thinks the opponent will be. This can be done in several ways. This work uses a simple and straightforward approach that is easy to calculate, albeit not the most accurate. The opponent's previous and current position are used to determine its velocity. This is used to estimate the mean x and y positions at future time steps. The standard deviation of the Gaussian is increased exponentially as the future time increases to account for increased unpredictability further into the future. Further information regarding the method used for dynamic obstacle modeling and its implementation can be found in [3]. While this is a simple and easy to use method, it is by no means the best way of predicting future trajectories. More advanced estimators could be used such as an Extended Kalman Filter [38], or a particle filter [9]. Also, the opponent modeling could be modified to take into account the static obstacle map to exclude areas of the map where the dynamic obstacle could never be from the probability distribution. This would result in more accurate predictions.

2.5 Cost Function

The quality of each path returned by the planner is determined by a cost function, which should be minimized. The cost of an action is broken up into two components, the cost of time passing, C_{time}, and the cost of a collision, C_{col}. This is to represent the dual goals of the robot: getting to the goal as quickly as possible and avoiding collisions. The total cost of a path is simply the sum of the costs of each action taken along the path. Costs are defined to be non-negative, with the minimum possible cost a robot can accrue during an action being zero. The agent does not accrue cost penalties for time passing while on the goal. When not on the goal, the agent accrues cost C_{time} for each time step that passes. The value of C_{time} is relatively small. The agent will also always accrue cost based on the probability of its current action resulting in a collision. The collision cost C_{col} is relatively high compared to C_{time}. The values of C_{time} and C_{col} can be set to achieve desirable behavior from the robot. The larger C_{time} is compared to C_{col}, the more risks the robot will take to get to the goal sooner. In the experiments in this thesis, C_{time} is set to 5 and C_{col} is set to 1000. Given an action that transitions between two states, the total cost of the action, C, is defined as:

$$C \equiv P(col) \cdot C_{col} + status \cdot C_{time} \tag{2.20}$$

where $P(col)$ is the probability of a collision occurring with any of the dynamic obstacles. *status* is either one if the robot is not on the goal or zero otherwise. The probability $P(col)$

is computed assuming that the events of not colliding are independent, as in [21]. If this is the case,

$$P(col) = 1 - P(\overline{col}) = 1 - \prod_{i=0}^{k} (1 - P(col)_i) \tag{2.21}$$

where $P(col)_i$ is the probability of colliding with the ith dynamic obstacle out of a total of k dynamic obstacles present. An algorithm that attempts to minimize the cost C will be attempting to reach the goal as quickly as possible while also trying to avoid obstacles.

2.6 The Planning Problem

Now that all the elements that define the space to be planned over have been discussed, we can define the actual planning problem faced by the robot agent. A planning instance $\mathcal{P}$ is defined as the tuple $\mathcal{P} = \{S, s_{start}, G, A, \alpha, O, D\}$ where:

- S is the set of states, where a state is the tuple $\langle x, y, \theta, v, t \rangle$ corresponding to location, heading, speed, and the time.
- s_{start} is the starting state, $s_{start} \in S$.
- G is the set of goal states, where each state $g \in G$ is the tuple $\langle x, y, \theta, v, \rangle$. Goal states are underspecified states in S because it is unknown when the robot will arrive there, hence they do not include the parameter for time.
- A is the set of all possible actions available to the robot, i.e. the set of motion primitives. An action is a function $a : S \rightarrow S$ that maps states to states. Each action has a duration of t_a.
- α is a function $\alpha : S \rightarrow Q,\ Q \subseteq A$ that maps states in S to a subset of actions in A that can be applied from that state.
- O is the set of static obstacles. Their locations are known perfectly and do not change. They are represented as cells of some fixed size and may be either free or blocked.

- D is the set of dynamic obstacles. Each dynamic obstacle in the set is represented as a function $d : t \to \mathcal{N}$ from time to a bivariate Gaussian distribution of the object's location. These distributions can change as the robot acquires more observations of an obstacle.

In addition, the amount of time allowed for each planning phase is defined as t_p. This value must be less than or equal to the duration of the motion primitives, t_a, so that the robot will always have the next action to execute by the time it completes its current action. A real-time planning algorithm must always return an action $a \in \alpha(s_{start})$ for a given problem $\mathcal{P}$ within the planning time-bound t_p.

2.6.1 Search Space

The search for solutions to the robot motion planning problem occurs over the space of possible states that the robot may be in. These states form the vertices of the graph that is searched over. The state space is five dimensional. Since time is one of the dimensions, it is also infinite, unless a bound on the maximum time value allowed is used. The space of actions available to the robot determines the edges in the graph. These edges are directed and may (and probably do) form cycles. Each edge in the graph has a corresponding weight that designates its cost. This weight is made up of two parts. The first part is the cost contributed by the cost of time passing, C_{time}. This cost is based solely off of whether or not the robot is in the goal configuration, and as such it is a *static* cost. It will not change over the life of the planning problem for a given edge. The second part is the portion of the cost contributed by the probability of colliding with dynamic obstacles, C_{col}. This is based on the predicted future positions of the dynamic obstacles and the location and time over which the action takes place. Since these predictions are often inaccurate and are updated often, these costs will fluctuate up and down across planning iterations. Since the values of the edge weights of the graph can change over time, the graph is called *dynamic*. As we will see in later chapters, the inclusion of time in the state space and having dynamic edge weights in the graph, although necessary to properly represent dynamic obstacles, poses a major problem to most algorithms previously developed for robot motion planning.

2.6.2 Initialization

When the planning system is initialized, it is given the static obstacle map, O, the motion primitives, A and α, the set of goal states, G, and the time allowed per planning cycle, t_p. The planner must also know the robot's radius and the radii of the dynamic obstacles in the world, both of which are assumed to be circular. This allows it to expand the static obstacle map by the robot's radius so that the robot can be treated as a single point object. It also allows it to precompute offsets for the cells that are touched by each motion primitive. This allows for fast collision checking and cost computation. It should be noted that although the simulator used makes sure each planner returns an action within the hard time constraint t_p, the real-time planners actually use a node expansion limit to achieve real-time performance. This expansion limit is tuned for each algorithm.

2.6.3 Planning

The planner will only be initialized once. During the subsequent planning cycles, the planner will be given the current state of the robot, s_{start}, and the time-parameterized Gaussian distributions representing the predicted trajectories of the dynamic obstacles, D. The static obstacles, motion primitives, and the set of goal states that were provided for initialization will persist unchanged throughout all the planning phases. Each time the planner is called, it must return an action $a \in \alpha(s_{start})$ to execute within the time constraint, t_p. The robot will execute the action and call the planner again to get the next action to take. The robot calls the planner while it is executing its current action, giving the planner the end state of its current action to use as the start state to plan from. This allows for a seamless plan-act cycle since the planner plans from the expected state of the robot after it finishes its current action. If it happens that the robot does not reach this state due to a collision or other event, then the work done by the planner during that iteration is thrown out and

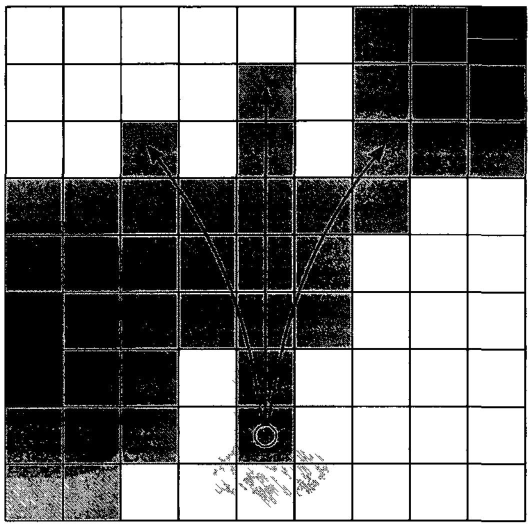


Figure 2-5: A robot and three possible motion primitives to execute. The cells traversed by the motion primitive are purple. There are also two static obstacles (black) that have been expanded (grey) by the robot's radius (light green).

the planner is sent the robot's current state as the new start state.

When a planner expands a node in the search space, it will generally proceed in the following manner. Figure 2-5 shows a robot in green with three possible motion primitives to execute in red. The point location of the robot is represented by the small green circle, whereas the actual area of the robot is shown by the larger, light green circle. The grid used to represent static obstacles has been overlaid. There are two static obstacles colored black that have been expanded (in gray) by the area of the robot (the area of the robot is shown in green expanding out of the obstacles for convenience). The blue squares designate the cells that are traversed during the execution of the motion primitives. They are the cells that must be checked for static obstacles. In this example none of the motion primitives comes into contact with a static obstacle. Since each motion primitive is valid, the cost will be computed for each one and the successor states generated by the planner. The planner will then proceed to expand other nodes.

2.6.4 Implementation and Optimizations

This section will discuss actual implementation choices and optimizations used that were found to be useful when creating the domain. As mentioned previously, the state variables of the robot and its actions are discretized and represented as integer types. This allows for considerably more nodes to be visited during search since integer arithmetic is much faster than floating point arithmetic.

The static obstacle map is expanded by the radius of the robot upon initialization to allow for fast static collision lookups, consisting of a single lookup into a two dimensional array. The cells that are touched by each motion primitive are also precomputed and stored as offsets. This way, these offsets can just be applied from the current cell of the robot to get all of the cells that need to be checked for a static collision for a given motion primitive.

The same technique used for static collision checking is applied for computing the costs of dynamic collisions, except here all the cells within a certain radius of the motion primitive are precomputed. Two more optimizations were used with respect to computing cost. First, since many more grid cells must be looked at when computing the cost of an action versus just checking it for a static collision, a larger grid cell resolution is used for this than for the static obstacle grid. Secondly, since evaluating equation (2.16) is computationally expensive, it is just approximated. The function is evaluated over a number of points out to three standard deviations in either direction and the values are stored in a lookup table. The closest value for a given query is then retrieved from the table.

CHAPTER 3

SIMULATOR

Due to the expensive costs and the technical difficulties of operating a real robot, a simulator was built to evaluate the effectiveness of different algorithms in the domain described. The simulator was built to allow for the possibility of a number of agents, each running a different planning algorithm, to be tested. It also allows for moving obstacles following predetermined paths to be simulated. Assuming the robot planning algorithms are deterministic, then the output from simulator runs will also be deterministic[1]. That is because the simulator runs in *simulation-time* as opposed to true *real-time*. This allows experiments to be re-run without getting different results. The simulation-time aims to be as close to real-time as possible, given the processing power of the system. Note that while the simulator is run in simulation-time, the planning algorithms are still given a hard time limit to adhere to that is in real-time, not simulation-time.

The reason a separate simulator was built instead of using one of the open source robot simulators already available [8, 16] was to have tight integration between the simulator and the planning domain, and also to allow for the simulator to be deterministic. The display of the simulator that was developed is shown in Figure 3-1. The faint gray grid lines are each separated by one meter. The static obstacles are shown in black. The dynamic obstacles are shown as circles and the robot running the planning algorithm is red with black wheels.

[1]Due to nondeterminism in the network and CPU scheduling, the output of the simulator is actually not deterministic but in this case it will be because network lag or CPU scheduling caused a real-time planner to fail to return an action in time and this will be caught and reported as an error by the simulator

The goal location of this robot is the opaque red circle. The triangle in the circle specifies the goal heading of the robot. The circles extending away from all of the robots show the predicted future locations of each robot up to eight time steps in the future.

3.1 Simulator Modules

The complete simulator package consists of two programs to be run as separate processes, allowing it to span multiple machine boundaries. The design of the simulator is shown in Figure 3-2. Each separate process is enclosed in a blue box. Separate threads (besides the main thread) for each process are enclosed in red boxes. The physical machine boundary is shown by the dotted line. The direction of information flow between modules of the same process or between processes is shown by directed arrows. The reason the simulator framework is divided into two (or more) separate processes is because it may be necessary to run the simulation and the motion planning programs on different computers, as they are CPU intensive processes. The first program is the simulator. It is responsible for rendering the robots at each time step, performing collision checking, performing opponent modeling, keeping track of statistics, and sending and receiving planning information from each robot agent. The second program is the agent program. For each robot, there is an agent program running as a separate process. The agent program is responsible for reading in the robot's state and the opponent models for the current time step being planned for, doing the planning, and sending the action to take back to the simulator. The agent arrives at its action choice by invoking the designated planning algorithm that is being used for the robot.

3.1.1 Renderer

The renderer module is responsible for drawing the world and the robots on screen. It also draws other helpful information such as the goals for each agent, the current plan for each agent and the predicted future locations of each robot. Besides drawing the scene, the renderer is also responsible for doing collision checking. If a collision is detected, the velocity of the robot is set to zero and the position set to the location just before the collision would have happened, so that the physical areas covered by the robot and the obstacle are not overlapping. Lastly the renderer must update certain statistics for each robot, such as the amount of time spent on the goal, number of collisions, and so on. The renderer is run in a separate thread that runs a certain number of times per second based on the desired frames per second. The renderer reads from the world model, reads and writes to the agent model, and reads from the opponent model. Thus, synchronization is used with the agent and opponent model for thread-safety. Since the world model is only read from and remains constant once it is created, thread safety mechanisms are not required when accessing it.

3.1.2 World Model

The world model defines the size of the world as well as the static obstacle grid. Since the size of the world and the static obstacles are assumed to not change during the simulation, the state of the world model remains constant once created. The world module also defines other superficial constants that are used by the renderer, such as the colors used to draw the static obstacles and the floor of the world.

3.1.3 Agent Model

The agent module contains all the information that pertains to a single robot or dynamic obstacle. This includes the robot's name, the robot's geometry, the robot's current state, the robot's predicted future locations, the robot's current and next actions, and the host name of the computer the robot's planner is running on. The agent model also specifies the type of planner that the robot is running, i.e. real-time or not real-time. This is so the manager knows whether or not to treat a late action as an error or not. Lastly, the agent model contains the status of the current move, i.e a collision, on the goal, or not on the goal. This allows the simulator to keep track of costs correctly.

3.1.4 Opponent Model

The opponent model stores the current and past locations of each agent and predicts their future locations from that information. This prediction returns a function from time to a bivariate Gaussian probability distribution, as specified in section 2.4.1. The prediction algorithm is performed every time step, prior to sending the current start states of each robot to the planners. The 95% confidence intervals of the predicted future locations of all dynamic objects out to eight time steps are shown as empty circles in Figure 3-1.

3.1.5 Manager

The manager module is what communicates with each robot's planner. It sends the start states and dynamic obstacle predictions to each planner and then reads in their action and updates the agent model. It is also responsible for enforcing time constraints on the real-time planners. The manager and the planners adhere to a communication protocol that is specified in detail in Appendix B.

3.1.6 Planner

The planner is what communicates with the manager module of the simulator program. The planner is a separate program that may be run on a different physical machine than the simulator. The reason for this is that planning may be a very CPU intensive operation and multiple planners could slow down the simulator as well as the other planners. For experiments, each planner should also have access to its own machine so that correct timing results can be recorded. The communication between planner and simulator is done over the SSH protocol. This allows the simulator to spawn planner processes on different machines without the need for a dedicated server to be set up, as the SSH server is already filling this role. When the planner is created, it reads in the initial world state, the goal state, and the name of the planning algorithm that it should use. It then loops over reading the robot's initial state and the dynamic obstacle predictions, invoking the correct planning algorithm, and returning the next action, until the simulation is finished. The planner also keeps track of certain statistics for the specific planning algorithm used, such as the number of nodes expanded, and the expected cost.

3.2 Simulator Statistics

The simulator keeps track of certain statistics for each robot so the planning algorithms used may be evaluated. These include the number of collisions with static obstacles, the number of collisions with dynamic obstacles, the total cost that each planning algorithm expected to accrue, and the total cost each robot actually accrued.

3.3 Simulator Execution

A high level view of the flow of execution of the simulator and the planner processes is shown in Figure 3-3. Once the simulator is started, it first reads in a configuration file, either from a physical file or from the standard input stream. This file specifies everything the simulator needs to know to run the simulation. It contains certain global properties such as the size of the map, the image to use to create the static obstacles, the planning time allowed, and length of the simulation. It also contains, for each robot, information such as the motion model used by the robot, the planning algorithm used, the host computer to run the planner on, and so forth. An example configuration file is shown in Appendix C. The simulator then initializes the world and agent models in accordance to the configuration file. The main thread of the simulator then creates the renderer thread and the manager thread and waits for their termination. From here, the renderer will draw the initial world state and then wait on the manager at a synchronization point. The manager will initialize the planner program for each agent and send the planners certain initialization information, such as the planning algorithm to use, the motion model to use, the static obstacle map, and the radius of the robot. When both the renderer and the manager are ready, the simulator proceeds. The render will simply perform collision checking and then display the world for the current time step at regular intervals and then wait on the manager before proceeding to the next time step. The manager proceeds by updating the opponent models and future trajectories for each robot, and then sending this information along with the start state of the robot to each of the planners. After the planner has computed an action to take, it is sent back to the manager, which reads it in, updates the agent models, and synchronizes with the render. This process continues for a user specified number of cycles. The manager will then notify each planner that the simulation is over and read in statistics sent from each of the planners before terminating. The main thread then wakes up and outputs all global and agent-specific statistics before exiting.

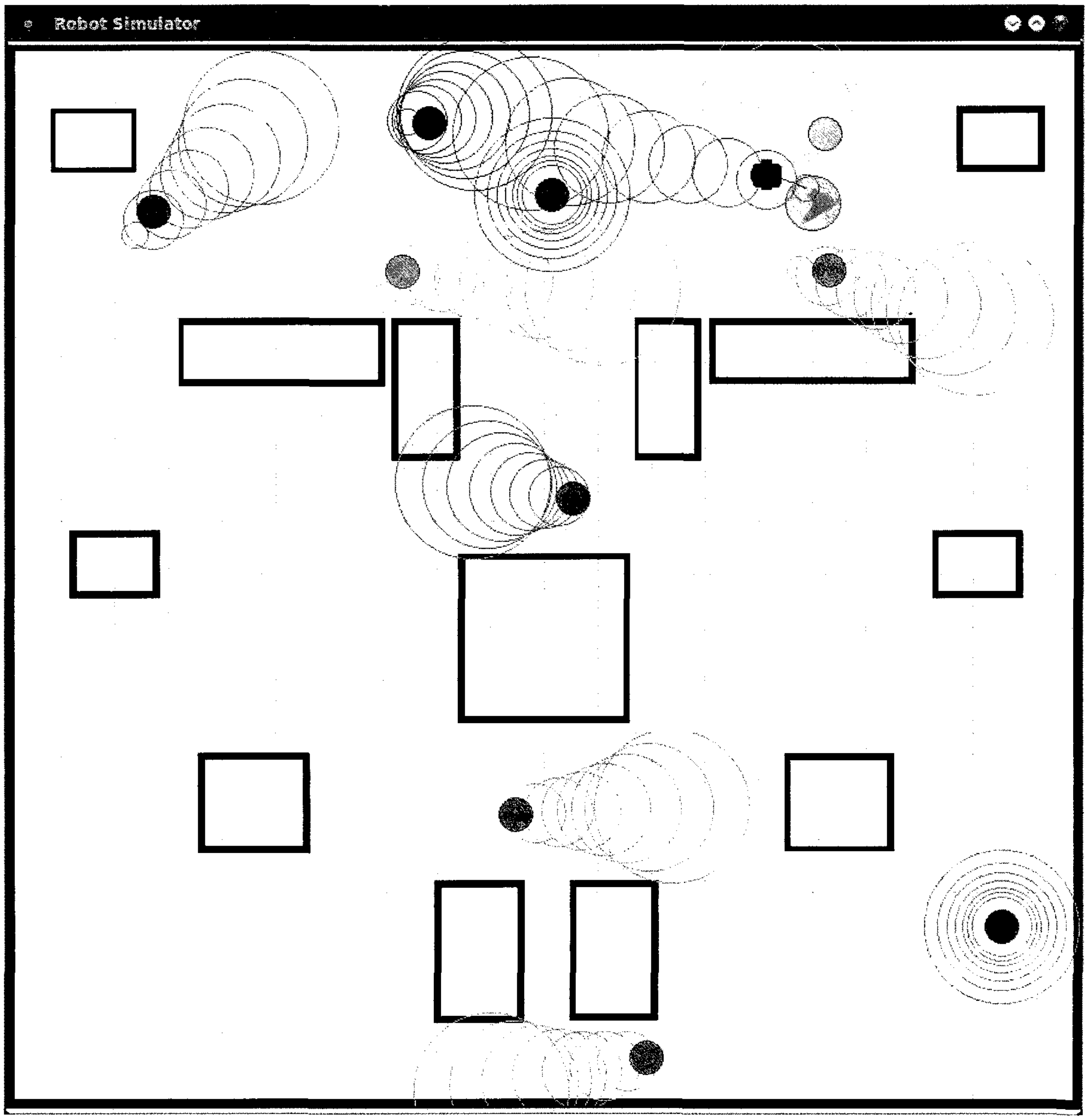


Figure 3-1: The robot simulator being run with 10 dynamic obstacles (circles) and one real-time planning robot (red). Future predicted robot locations are shown by empty circles.

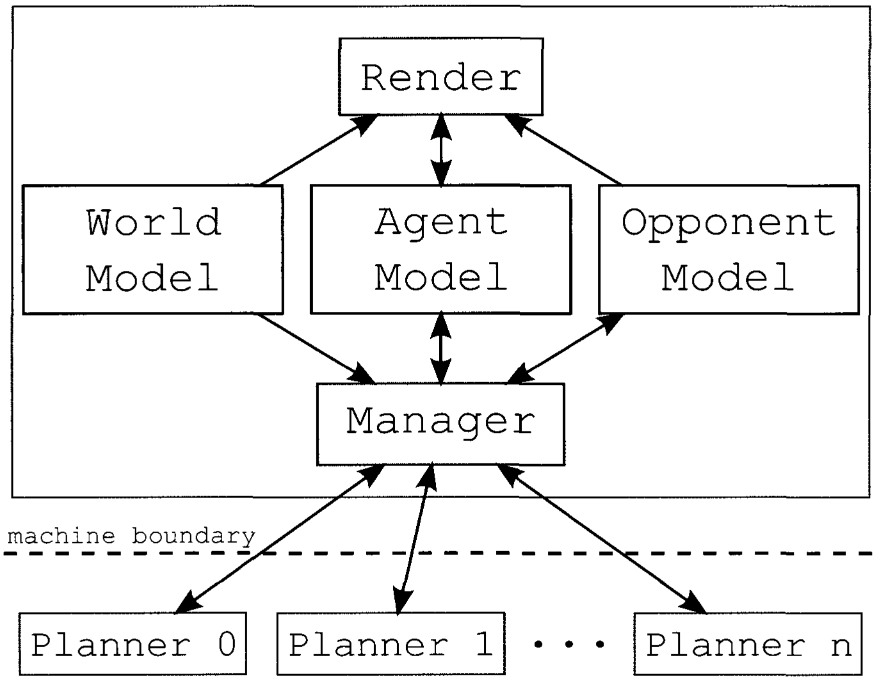


Figure 3-2: The major modules of the robot simulator. Separate processes are shown in blue and separate threads in red.

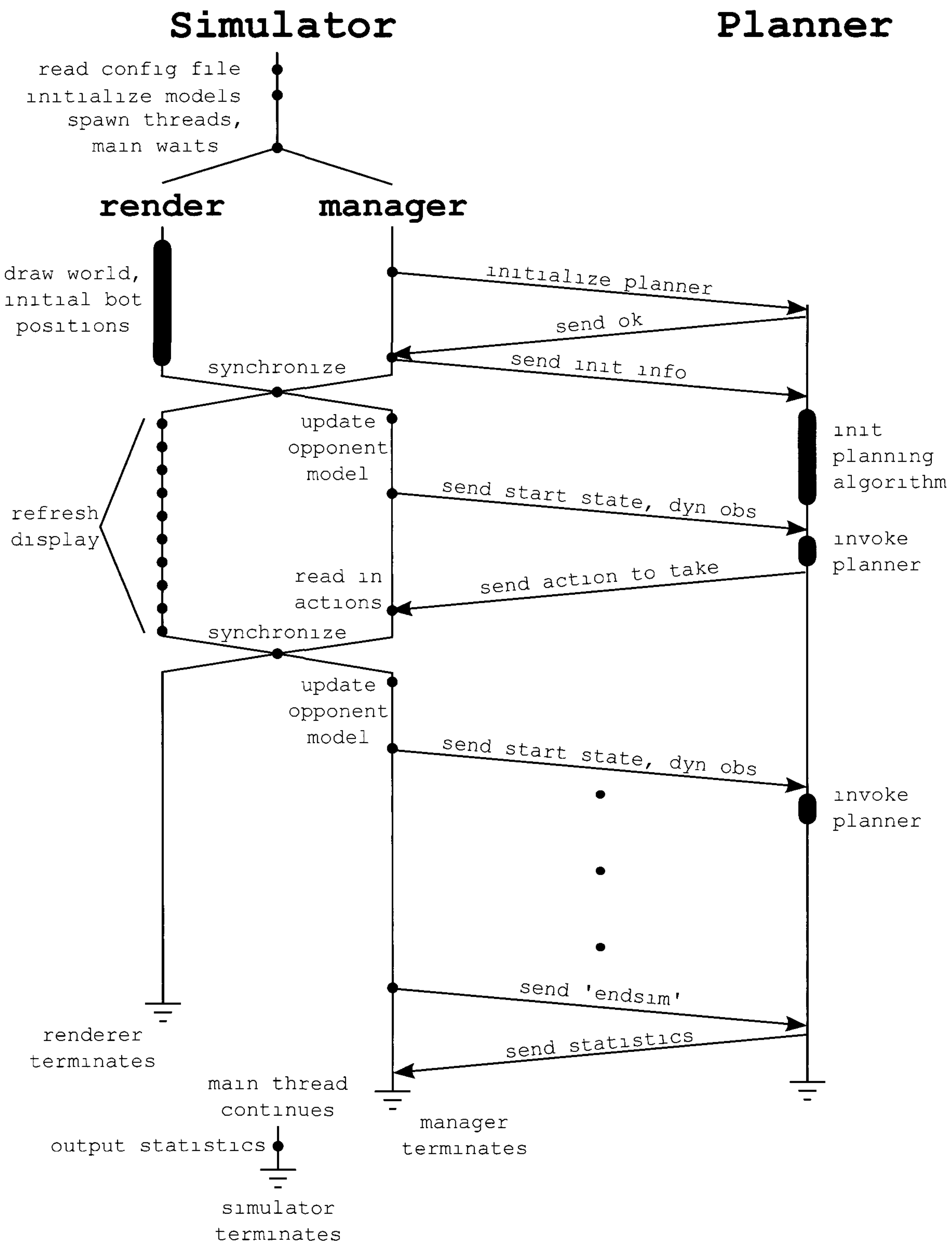


Figure 3-3 The flow of execution for the simulator and planner programs

CHAPTER 4

PREVIOUS WORK

Many techniques have been developed to solve motion planning problems in robotics. This chapter will discuss the major ideas and algorithms that pertain to this problem. Four different classes of algorithms that have been successfully used for robot path planning will be discussed. These classes are: heuristic search, real-time heuristic search, randomized sampling, and reactive methods. The most current and relevant algorithms from each of these four areas will be discussed. Some of these algorithms will be included in the empirical evaluation presented in the next chapter, and some will be discarded for reasons that will be stated.

4.1 Heuristic Search

The problem of planning a trajectory for a mobile robot can be solved using search as follows. Let S be the set of all possible states that the robot can be in and A be the set of all possible actions the robot can take. Let $s \in S$ be the current state of the robot and $\alpha : S \rightarrow A$ be a function that maps states to applicable actions for that state. Let G be the set of states considered to be a goal. A basic search algorithm could plan a path from start to goal by performing a uniform cost search [30] where the start node is the robot's current state and the children of a node are all the states that can be reached by executing one of the actions available to the robot from that state. A node is said to be *expanded* once it is visited and its successors are generated. Once a node is expanded that constitutes a goal, the search is terminated and the parent pointers can be followed back to the start node

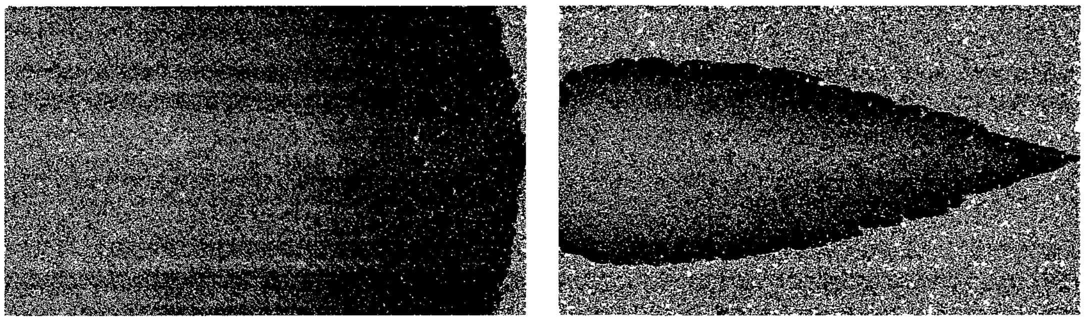


Figure 4-1: On the left, the states explored by a uniform cost search to find a path from the left side of the map to the right side. On the right, the states explored by A* for the same problem. Images courtesy of Jordan Thayer [36].

to obtain a complete path from start to goal. The previously described search method is *complete* in that it will find a solution if one exists or report that there is no solution. It is also *optimal* since the cost of the solution path returned is minimal. Unfortunately, it visits many nodes that are not along the solution path that is eventually found, making it a completely impractical real-time search algorithm for all but the most trivial problems.

One way to drastically reduce the number of nodes that need to be expanded to find a solution is through the use of a *heuristic*. A heuristic function is a function that returns an estimate of the cheapest cost path from any node in the search space to the goal. Heuristic functions can often be created by relaxing the problem constraints. These functions can be used to steer the search in a direction more likely to lead to the goal, eliminating the need to expand many of the nodes visited by uniform cost search. An example of the savings realized by using a heuristic during search can be seen in Figure 4-1. Figure 4-1a on the left shows the nodes visited by uniform cost search to find a path from the left side of the map to the right, avoiding obstacles shown in black. Figure 4-1b on the right shows the nodes visited by a heuristic search algorithm called A*. As can be seen, many fewer nodes need to be visited, resulting in a drastic decrease in search time, even though both algorithms find optimal solutions.

4.1.1 A* Search

Perhaps the most well known heuristic search algorithm is the A* algorithm [10]. It is guaranteed to find optimal solutions if the heuristic it is using is *admissible.* A heuristic is admissible if it never overestimates the cheapest cost of getting to the goal from any state. For example, the heuristic used in Figure 4-1 is the straight line distance from a node to the goal, ignoring obstacles. This is obviously admissible since it is the shortest possible distance between two points. It has been show that A* visits the minimum number of nodes necessary to return a provably optimal solution for a given admissible heuristic function [10]. During the search, A* keeps a priority queue of nodes that have been generated but not yet visited. This priority queue is referred to as the *open list.* The priority of a node n is the function $f(n) = h(n)+g(n)$. Here $h(n)$ is the heuristic function, i.e. an estimate of the cost of the cheapest path from n to the goal. $g(n)$ is the cost of the path from the start state to n. This is not an estimate, but is known exactly since all the nodes along this path will have already been expanded by the search in order to generate n. A* works by continually removing the best node on the open list and generating its children. The h and g values of the child node are computed and each child is added to the open list. Once a goal node has been removed from the open list and the f values of all the other nodes on open are greater than or equal to the cost of the goal node, the search may terminate, having found the optimal solution (assuming the heuristic used was admissible). For domains that have many duplicates, such as the robot motion planning domain discussed, A* also maintains a separate list called the *closed list* which is often implemented as hash table. This list is necessary to perform duplicate checking and to make sure that A* has computed the cheapest cost path to nodes that it expands.

The ideas from A* search will be the basis for many of the algorithms discussed in this chapter. The reason why A* isn't good enough is that it is an optimal search algorithm. Since the problems pertaining to this thesis are PSPACE-HARD [33], A* will take time and space exponential in the problem size to find a solution. Since A* must not only find a solution, but also prove its optimality by expanding all nodes that could possibly lead to cheaper solutions, it will almost certainly not satisfy the real-time constraints necessary for the robot motion planning problem that is the basis of this thesis. The weighted A* algorithm [32] is exactly the same as A*, however it weights the heuristic function used. The priority of a node is computed as: $f'(n) = g(n) + w\, h(n)$ where w is the weight being used. The higher the weight is, the more greedy the search will be on h. For admissible heuristics, weighted A* is guaranteed to find solutions of cost no worse than w times the optimal solution. While this often drastically reduces the number of nodes that are visited, for hard problems weighted A* will still almost certainly not meet the real-time constraints.

4.1.2 Heuristics

There are several different heuristics that have been shown to work well with the robot motion planning domain. These include the simple straight line distance heuristic, the static 2D Dijkstra heuristic [26], and the motion constrained free-space dynamic heuristic [26]. These heuristics are all admissible as they never overestimate the cost of reaching the goal from any state, assuming there is only one goal state. If multiple goal states are allowed, certain precautions must be taken that will be discussed. Unfortunately, these heuristics were developed for domains with no dynamic obstacles and they do not take the costs associated with dynamic obstacles into account. This leads to a search space with large heuristic local minima.

Static 2D Dijkstra Heuristic

One useful heuristic which captures the static obstacles, but not the robot's motion constraints, is the static two dimensional heuristic calculated using Dijkstra's algorithm [4]. The idea is to take the static obstacle matrix describing the world and compute and store the shortest paths from each cell to the goal cell by using Dijkstra's algorithm. The robot's heading, speed, and any other state variables are ignored, and the planning is done only

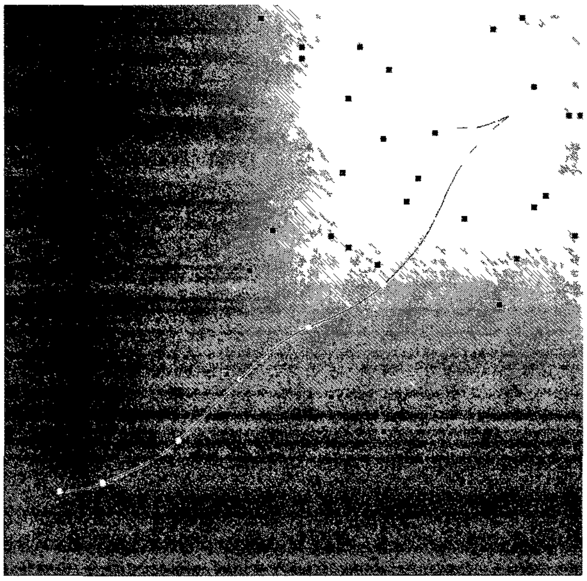


Figure 4-2: The heuristic values of the 2D static heuristic calculated by Dijkstra's algorithm and the resulting A* plan for a differential drive robot using motion primitives. White areas correspond to lower cost.

between 2D grid cells. A visual representation of the costs computed by this heuristic is shown in Figure 4-2. The dark red dots are static obstacles and have maximal cost. For the Dijkstra search, movement is allowed to any adjacent grid cells in eight directions. Moving in the cardinal directions costs 1 and moving along the diagonals costs $\sqrt{2}$. Upon completion, the weights computed by Dijkstra's algorithm for each cell must be converted to time steps, since this is what determines the cost accrued from C_{time} (see section 2.5). The length of a time step is the duration of a single motion primitive. The conversion is done by multiplying the costs by the size of a grid cell and dividing by the maximum distance that the robot can travel during a single time step, in order to keep the heuristic admissible. This must further be divided by a factor to take into account the difference between the angles of movement allowed by eight-way grid movement ($45\,^{\circ}$) versus the headings allowed for the robot state based on the discretization used.

Theorem 1 *Let the smallest possible heading the robot can have be θ, where $\theta \leq 45\,^{\circ}$. Let*

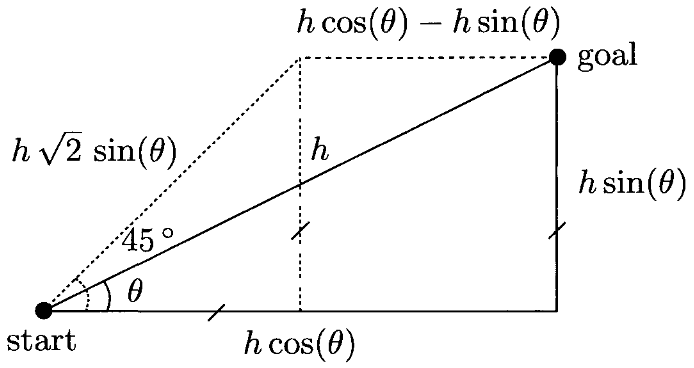


Figure 4-3: The relationship of shortest paths resulting from different heading discretizations.

r be the length of the side of a single static grid cell and d be the maximum distance that the robot can travel in a single action. Then, the weights reported by a Dijkstra search of the static obstacle grid can be made admissible by multiplying each cell weight by $r/(dc)$, where c is computed as:

$$\left(\sqrt{2}-1\right)\sin(\theta)+\cos(\theta)$$

Finally the cost should be multiplied by the user defined cost constant C_{time} to capture the user specified cost of time passing while not on the goal.

Proof: By multiplying the weight associated with each square cell by the cell resolution and dividing by the maximum distance the robot can travel in one action, the weights reported by the Dijkstra algorithm are converted into distance, and then into an admissible estimate of time steps. We must now find the ratio of the shortest path that can be found using the eight-way grid movement, versus the shortest path that can be found using the motion primitives, for an arbitrary start and goal position. Figure 4-3 shows the shortest paths possible between two points for an eight-way movement and for motion primitives. The robot starts in the lower left hand corner and needs to move to the upper right hand corner. In the best case for the motion primitives, the robot will be facing in the direction of its goal and will just be able to drive straight at maximum speed until it has arrived. This path is denoted by the hypotenuse of the triangle which has a distance of h. The shortest path that an eight-way movement robot can take is to travel at 45 ° and then across to the goal. This is denoted by the dotted line and has a length of $h\sqrt{2}\sin(\theta) + (h\cos(\theta) - h\sin(\theta))$. The ratio of the shortest path found using eight-way movement versus the shortest path that can be found using the motion primitives can then be computed:

$$\frac{\text{eight-way}}{\text{motion primitives}} = \frac{h\sqrt{2}\sin(\theta) + (h\cos(\theta) - h\sin(\theta)}{h} \tag{4.1}$$

$$= \sqrt{2}\sin(\theta) + \cos(\theta) - \sin(\theta) \tag{4.2}$$

$$= \left(\sqrt{2} - 1\right)\sin(\theta) + \cos(\theta) \tag{4.3}$$

Dividing by this value scales the weights of each cell, which are based on eight-way movement, to be admissible for the heading discretization used by the motion primitives. □

For a motion primitive discretization that allows 16 possible headings, θ is equal to 22.5 °. The ratio computed using (4.3) is approximately 1.08, matching the value reported in [26] for the same heading discretization. In the event that the possible goal states fall within several static grid cells, each of these cells must be given a starting weight of zero in the Dijkstra search. This is necessary to keep the heuristic admissible since the h value of a goal state must equal zero. This heuristic can be precomputed and stored once the static obstacle map and the goal configuration are known.

Motion Constrained Free-space Heuristic

While the two dimensional static heuristic is able to capture distance from the goal and the static obstacles, it knows nothing about the motion model used by the robot. Another heuristic can be used to capture exactly this. The idea behind this heuristic is to do a uniform cost search out from the goal to some depth limit using the motion primitives available to the robot. As each node is generated, it is added to a lookup table, along with its depth in the tree. To compute the heuristic of a node, it is queried from the lookup table. If it is found, its depth from the uniform cost search is returned. If it is not found, one plus the depth limit of the uniform cost search is returned, since the node must be at least that many actions away from the goal. This heuristic is admissible, since it is the least number of actions required to get the robot from some starting state to the goal state, assuming there are no obstacles in the way. If there are obstacles in the way, this value can only increase. In the event that the goal state is not a single state but a set of states, the uniform cost search must be seeded with each of these goal states to remain admissible. This heuristic can be precomputed for the specific motion model being used.

Two heuristics have now been described, one that takes into account the static obstacles in the world and the distance to the goal state, and another that takes into account the motion model used by the robot. Instead of only using one of these heuristics, it has been shown that taking the max of them can offer great performance boosts [26]. More advanced heuristic search algorithms for motion planning will now be discussed.

4.1.3 D* Lite

The D* Lite algorithm [17] is an incremental search algorithm based on the Focussed Dynamic A* algorithm [35]. It is called an incremental search algorithm because on successive searching iterations, it is able to reuse work from previous iterations, greatly speeding up planning. The D* Lite algorithm is able to search over dynamic graphs, where the edge weights of the graph may increase or decrease across search iterations. In the domains for which it was originally designed, the D* Lite algorithm is able to handle replanning for a moving robot. It does this by reversing the start state and the goal state and reversing the edges of the search graph so that the start state remains constant across all search iterations. This allows the g values of all the nodes explored in the search space to remain constant with respect to the start node. It is not obvious how to do this in a domain where time is part of the state, since it is unknown what time the robot will actually arrive at the goal node. D* Lite is not included in the empirical evaluation for this reason.

4.1.4 Time-Bounded Lattice

The Time-Bounded Lattice algorithm (TBL) [21] is designed for a domain very similar to the one used here. The state representations and dynamic obstacle representations presented earlier are based on this. The domain presented in TBL consists of a motion primitive constructed lattice including time, with Gaussian distributions for dynamic obstacles. Planning in the high dimensionality space of the motion primitives is only done until a time when the distributions of the dynamic obstacles are sufficiently spread out, T_b^{max}. The planning then continues in a two dimensional grid space to the goal, taking only static obstacles into account.

Limitations of this approach are that obstacles will seem to "disappear" to the planner after the time bound cutoff is reached. Although in the paper a way to choose T_b^{max} on-line is suggested, a hard-coded time bound of 4 seconds is used in their evaluation because it was too expensive to do planning in 6D for longer. Also, the planning is done with weighted A* and is not real-time. In some instances of their evaluation, plans took as long as 10 seconds to be computed. This would not be acceptable for a fast moving vehicle navigating amongst other fast-moving dynamic obstacles, such as automobiles.

4.1.5 Safe Interval Path Planning

The Safe Interval Path Planning algorithm (SIPP) [31] allows for bounding the number distinct time steps seen by the search. It is a method for reducing the size of the search space by not searching over distinct values of time but instead distinct *safe intervals.* A safe interval for a given location is the period of time such that there is no dynamic obstacle in the location for the entire interval, however there is an obstacle in the location at one time step before the interval and one time step after the interval. A* search is used to generate plans over the state space discretized by time intervals and has been shown to visit much fewer nodes. There are two assumptions that this algorithm relies on that may or may not be true for a given robot. First, it is assumed that the robot is capable of waiting in place for an arbitrary amount of time. This may not be the case if the robot requires movement to remain in a stable state, such as a motorcycle or an airplane. Second, it is assumed that the acceleration of the robot is negligible, i.e. robot can speed up or slow down instantaneously. Using this assumption was acceptable for the slow moving PR2 robot that this approach

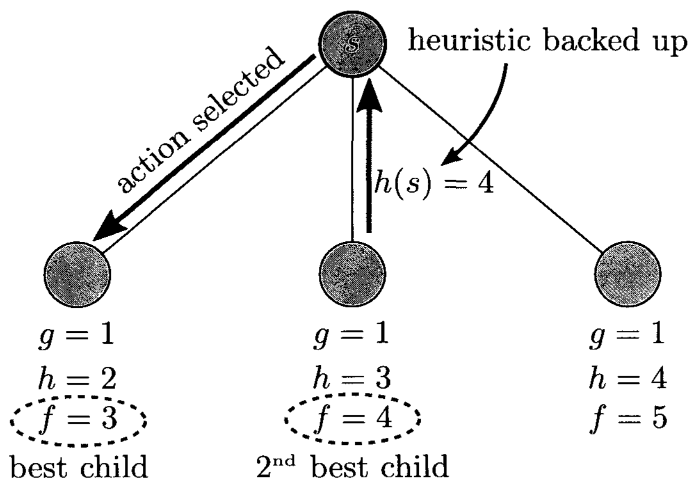


Figure 4-4: The child chosen and the heuristic function backed up for a single planning cycle of RTA*.

was tested on, however it is almost certainly not the case for any robot moving at moderate to high speeds, e.g. automobiles, hovercraft, etc. Since SIPP is not constrained to generate dynamically feasible paths, it is not considered in the empirical evaluation.

4.2 Real-time Heuristic Search

The search algorithms discussed so far, while taking advantage of heuristic information, and employing other clever techniques to reduce search, are not considered real-time algorithms. A real-time algorithm must be able to return the next action to take within a constant amount of time that is not dependent on problem size or difficulty. This section will discuss several real-time heuristic search algorithms. These algorithms must use specialized techniques to avoid getting stuck in local minima, since there is often not enough time available to plan a complete path from the start to the goal. The algorithm must return the best action to take based on the limited lookahead that it has.

4.2.1 Real-time A*

The real-time A* algorithm (RTA*) [19] was developed assuming that that a complete path to the goal can not be planned for during a single planning iteration. It takes steps to ensure that the agent makes the locally optimal decision, given the information it has learned from previous actions and its local lookahead. It does this by doing a limited lookahead under each successor of the start node s. For simplicity sake, assume that all it does is expand the start node and compute the f values of each successor. This is shown in Figure 4-4. The action towards the child with the lowest f value is selected to return. The f value of the second best child of s is cached as the new heuristic value for the node s as it represents the best the search algorithm can do if it returns to state s. This heuristic caching theoretically allows the search to eventually escape local minima and find a path to the goal state, even if no heuristic is used [19]. The reason that heuristic caching is needed is so the robot will not get stuck in an infinite cycle moving between some set of states. As RTA* was not developed for graphs that incorporate time as part of a state, it is not expected to perform well in this domain, and empirical results will confirm this. Since RTA* only learns heuristic values for the start state on each planning iteration, all learning is effectively useless since the current state of the robot will be in the past during all future planning iterations. For the RTA* implementation used in this thesis, a depth-first search is used to compute the lookahead under each successor of the start node.

4.2.2 Real-time D*

Real-time D* (RTD*) [1] is a bidirectional search algorithm that combines a local search outward from the robot's starting position with a global search backward from the goal. The intuition is that the robot can use the local search to select actions in real-time. The backwards search can run across iterations and reuse information to eventually converge on a complete solution.

This method relies on an incremental search algorithm called Anytime D* [27] to plan

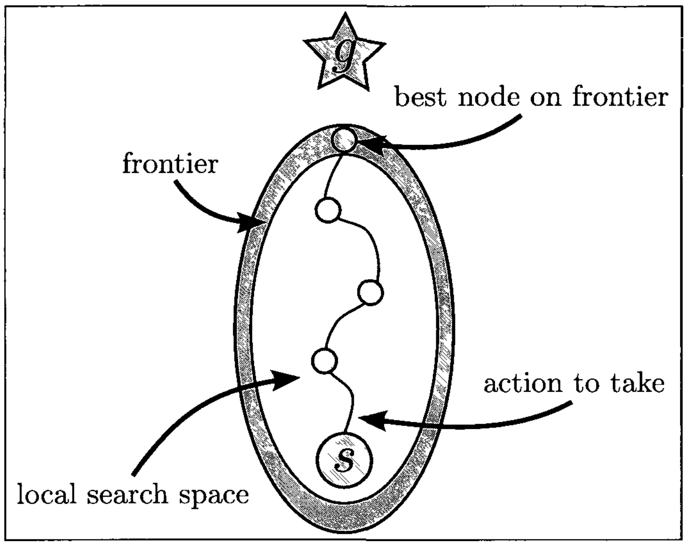


Figure 4-5: The local search space and frontier of an iteration of LSS-LRTA*. The best node on the frontier and the corresponding action to take are shown.

backwards from the goal to the start state. This allows the search to eventually converge on an optimal solution. However, in a domain where the state description includes time, it is unclear how to search backwards from the goal in this manner. This is because it is impossible to know exactly when the robot will arrive at the goal, so the exact state can't be fully specified. RTD* is not included in the empirical evaluation for this reason.

4.2.3 LSS-LRTA*

Real-time search algorithms must be able to issue actions incrementally, and within very short time windows. Thus, they often have to select an action to do even though a complete path from the start state to the goal state has not been found. Local Search Space Learning Real-time A* (LSS-LRTA*) [17] is a modern real-time heuristic search algorithm that incorporates learning of heuristic values interleaved with searching. This algorithm searches forward from the start position of the robot to the goal. The search phase consists of searching using A* until a node expansion limit is reached. It then selects the node with the lowest f value on the frontier of unexpanded nodes (i.e. the open list) and returns the first action along the path to this node. It then performs Dijkstra's algorithm from the nodes on the frontier to the nodes in the local search space, learning updated heuristic values for these nodes. Similar to the cached h technique of RTA*, but more efficient, this allows LSS-LRTA* to avoid getting trapped in cycles and avoid the local minima caused by not being able to plan complete paths to the goal. Figure 4-5 shows a visual depiction of the frontier and local search space of a LSS-LRTA* search. The problem with this search algorithm is that because time is constantly progressing, the heuristic values learned for nodes could quickly become out of date. This will cause the learning step to not be useful.

4.2.4 Partitioned Learning Real-time A*

The Partitioned Learning Real-time A* (PLRTA*) [3] is an algorithm based off of LSS-LRTA* that was specifically designed to handle the dynamic costs associated with the domain used in this thesis. Its main contribution is to realize that the costs associated with the domain are of two types: *static* and *dynamic*. Static costs are the costs accrued by the robot as time passes while not in the goal state. These costs are only dependent on whether or not the robot is on a goal, and as such, do not change and are independent of the time value associated with each state. Dynamic costs are the costs associated with the probability of colliding with a dynamic obstacle. These costs are always changing as the estimates of future dynamic obstacle trajectories are recalculated. Also, these costs are dependent on the time value associated with states.

LSS-LRTA* has the problem of cached h values becoming out of date as time progresses, limiting the usefulness of the heuristic learning in the first place. The PLRTA* algorithm solves this problem by caching both the static costs of a state and the dynamic costs of a state as two separate entities. Since the static costs do not depend on the time of a state, they can be generalized across time, allowing the algorithm to learn useful and persistent static h values. Since the dynamic h values do depend on time and can change, a heuristic decay technique is used to discount these values as time progresses. This technique seems to be very useful for adapting real-time search for this domain as the results will show.

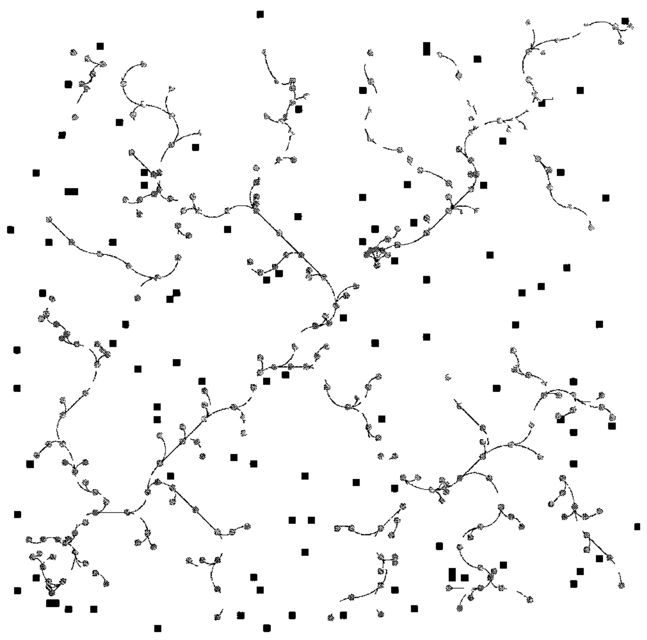


Figure 4-6: An example RRT made up of motion primitives where the start is the bottom left and the goal is the upper right.

4.3 Randomized Sampling

So far, only methods dealing with heuristic search and real-time heuristic search have been discussed. These methods explore all possible sequences of actions that the robot can take from the start state, using heuristics to avoid having to explore some paths the won't contribute to a solution. As the dimensionality of the state space and the branching factor of the nodes in the state space increases, search can become infeasible as there are just too many states to explore. This problem can be solved by sacrificing the completeness of the algorithm and the optimality guarantees. Randomized sampling techniques can be used to greatly reduce the number of states that need be explored. Most randomized techniques, while not complete, are *probabilistically complete* in that as the number of samples increases to infinity, the probability of finding a solution if one exists goes to one.

4.3.1 Rapidly-exploring Random Trees

Rapidly-Exploring Random Tree (RRT) [24] is an extremely popular planning technique in the robotics community and has successfully been used to solve very hard problems, especially those with nonholonomic and kinodynamic constraints. The RRT algorithm works by growing a tree outwards from an initial state. The tree is grown randomly, but is heavily biased towards unexplored regions of the state space. An example RRT made up of motion primitives is shown in Figure 4-6. While RRTs are sometimes able to solve very hard problems quickly, their main drawback is that no guarantees are made on solution quality. Also, given more time, the solution returned by the RRT algorithm will not converge to optimal. In fact it has been proven that the best path returned by RRT almost always converges to a non-optimal one [12].

The basic RRT algorithm proceeds as follows. Initially, the only node in the tree is the start node. RRT then picks a sample state, x_{rand}, from the search space at random. The tree is searched for the nearest neighbor to x_{rand}, denoted x_{near}. This node is then expanded using the motion primitives to generate its successors. The successor that is closest to x_{rand} is then added to the tree and the algorithm continues.

Since RRT is not a real-time algorithm, a version similar to RRT, but real-time is used for evaluation in this thesis. In this algorithm, the RRT is limited to sampling a constant number of states. After this, the tree is searched for the node with the lowest h value. The action along the path to this node is returned.

4.3.2 Metric Adaptive RRT

For domains where cost isn't directly proportional to distance, the RRT algorithm has a major flaw: it does not incorporate cost into the nearest neighbor search. Besides the fact that the RRT algorithm makes no guarantees on solution quality to begin with, not incorporating cost into the tree generation procedure can result in paths that blatantly intersect with high cost areas of the map. The Metric Adaptive RRT (MA-RRT) algorithm

[25] was developed to address this issue. The MA-RRT algorithm also incorporates relevant features from the ERRT algorithm [2], a "real-time" version of the RRT algorithm. While it doesn't meet the definition of a real-time algorithm as described in this thesis, since it cannot return the action to take in constant time, the ERRT algorithm does make some attempt to use information from previous search iterations to improve the paths found on the current iteration.

The MA-RRT algorithm has the same general structure as the basic RRT algorithm, however it redefines the distance function used when computing the nearest neighbor of a node. The distance between two nodes is defined as:

$$dist_{cost}(n, n') = g(n) + w \cdot dist_e(n, n') \tag{4.4}$$

Where $g(n)$ is the cost of the path from the start node to node n, just as in heuristic search algorithms. The function $dist_e$ returns the Euclidean distance between the two nodes. The parameter w is a weight that is updated adaptively during the tree generation to bias the growth of the tree. A small w value causes the tree to become very bushy, as nodes are preferred that are close to the root. If w is set too high, the tree will not be as sensitive to the costs. While this cost function formulation allows the growth of the tree to depend on the underlying costs, it does have the disadvantage that the cost of every node in the tree must be computed to find the nearest neighbor each time the tree is to be extended. This is in contrast to the normal RRT algorithm that can use efficient data structures such as KD-Trees to reduce the complexity of this operation. Lastly, MA-RRT uses the node caching mechanism of [2] to bias the tree growth towards places where successful paths were found in previous iterations. When a path to the goal is found, all the nodes along the path are added to a cache of bounded size. During the next iteration, the growth of the tree is biased towards these nodes a certain percentage of the time. The MA-RRT algorithm is not included in the empirical evaluation however it would be interesting to see how it compares to the other algorithms tested.

4.3.3 RRT*

The RRT* algorithm [12] is a random tree approach similar to RRT, except that given more time, the solution that it returns will converge to optimal with high probability. The major difference between the RRT and RRT* algorithms is that while the RRT approach will only randomly grow the tree by one vertex at a time, the RRT* approach will first attempt to grow the tree by one vertex, v, and if that extension is determined to be successful, it will grow the tree further by adding several more vertices at once that are within some distance Δ of v. As the search progresses, Δ is decreased. In general, the solutions generated by RRT* are of much better quality than RRT, especially as time allowed to solve the problem increases. The drawback is that RRT* is more computationally intensive than RRT. The RRT* algorithm is not included in the empirical evaluation. This is because the RRT* algorithms was originally designed for holonomic robots and it is unclear how to implement it for nonholonomic robots.

4.3.4 Probabilistic Roadmaps

Probabilistic roadmaps (PRM) [14] are another random sampling approach that is similar to RRT. The main difference is that while RRT generates a random tree during the actual search, PRM initially generates a set of states throughout the entire search space. It uses local search techniques to connect states that are in the same vicinity of each other. This generates a sparse graph of the state space. PRM then searches over this sparse graph for a solution. Probabilistic roadmaps are mainly used to solve multi-query problems, where many paths need to be found throughout the same environment. The disadvantages of PRM is that they can take a long time to compute and must be updated every time the world changes, due to improved sensor readings for example. Also, it may be hard to connect close nodes with local searches during the generation phase. For these reasons, PRM is left out of the evaluation and RRT is instead preferred.

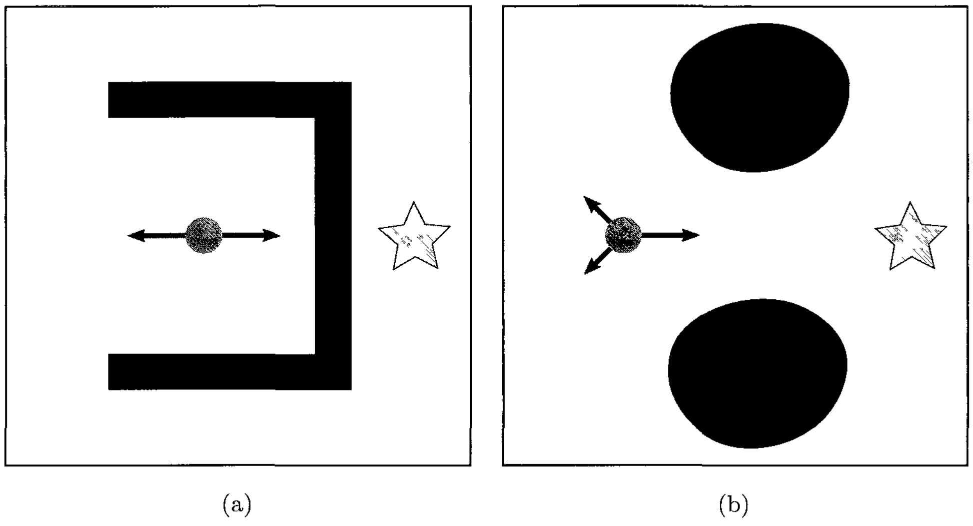


Figure 4-7: Two possible potential minimums. The repulsive forces shown in red and the attractive forces in blue.

4.4 Reactive Methods

Another type of algorithm that is commonly used for robot motion planning doesn't do any kind of search to find a solution. Instead, the plane of the state space can be represented as a differentiable function whose gradient slopes away from obstacles and towards the goal. The robot chooses actions to follow this gradient. While these methods suffer from many drawbacks [18] including getting stuck in local minima, they are easy to compute and could be useful for real-time motion planning.

4.4.1 Potential Fields

The potential field approach to path planning [34] works by transforming the plane of the world into a potential field. The goal is represented by attractive forces (low potential) on the robot and obstacles are repulsive forces (high potential). The robot is treated as a point in this potential field, always moving to lower its potential. At each planning iteration, the potential function can be updated to reflect the locations of the dynamic obstacles. The potential function can also be smeared to indicate the predicted trajectory of each dynamic obstacle. The robot then computes the vector of the total force acted on it by the repulsive potentials of the obstacles and the attractive potential of the goal. It chooses an action to take that most closely matches the direction and magnitude of this force.

The main drawback of potential field type algorithms is that they are prone to the problem of local minima in the potential function. This could cause the robot to never reach goal. Dynamic obstacles aren't necessary to reach this condition, as it can be caused by close proximity of static obstacles, such as in narrow hallways. Figure 4-7 shows an example of two possible scenarios where a robot could become stuck due local minima in the potential function. Notice, as in Figure 4-7b that the obstacles need not be concave to cause a local minima. Techniques have been described to transform the potential function into one with only one global minimum, known as a navigation function [7]. Unfortunately it is not clear how to perform these techniques in a real-time manner. It is also not clear what effects dynamic obstacles would have on these techniques. Potential field methods are not included in the empirical evaluations discussed in the next chapter.

CHAPTER 5

REAL-TIME R*

In this chapter, a new real-time algorithm for solving the robot motion planning problem with moving obstacles is presented. The real-time R* (RTR*) algorithm is a real-time adaptation of the R* algorithm [29]. The R* algorithm combines ideas from best-first heuristic search and random sampling to create a hybrid algorithm. In this chapter, the changes made to create RTR* will be discussed, namely how to deal with not having enough time to plan a complete path to the goal, and how to select the next action to return. Modifications made to R* specifically for the robot motion planning domain will also be discussed. Before RTR* can be introduced, the original R* algorithm must first be described.

5.1 R* Search

One of the difficulties of planning in high dimensional state spaces is that the size of the state space increases exponentially with each additional dimension. The addition of the time dimension to states considered in the robot planning domain yields an infinite state space. One popular approach that has been taken to solve hard high dimensional planning problems is to use randomized techniques to greatly reduce the number of states explored to find a solution [23, 12, 14, 11]. The R* search algorithm is one such algorithm. It attempts to quickly solve problems in high dimensional state spaces and avoid heuristic local minima by using random sampling paired with search. R* performs an interleaved two level weighted A* search [32] where the higher level states are generated randomly and sparsely over the state space and low level searches are performed in the original state/action

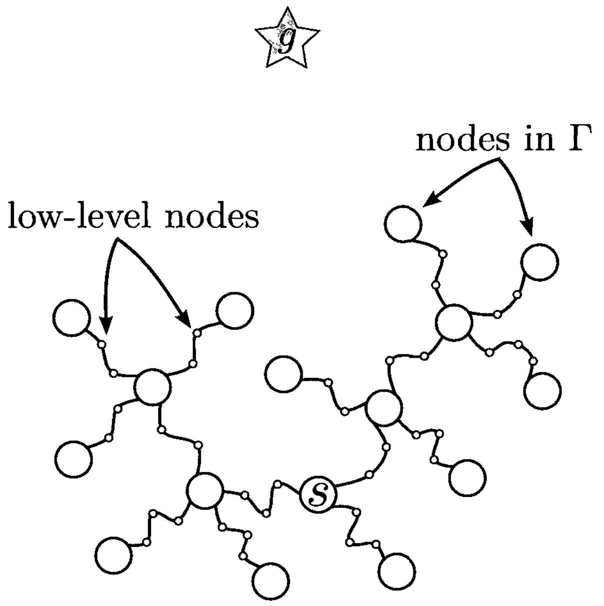


Figure 5-1: The nodes in Γ (large, white) and the nodes in the low level state space (small, yellow) that are explored by an R* search with $k = 3$.

space to connect these higher level states. The two tiered approach taken by R* has the advantage of splitting the problem up into smaller, easier to solve subproblems, while not forfeiting the accuracy of the actions provided at the low level search space.

At the top level, instead of expanding the direct predecessors of a state s, R* selects a random set of k states that are within some distance Δ of s. This behavior constructs a sparse graph, Γ, which is searched for a solution. The edges computed between nodes in Γ represent actual paths in the underlying state space. When tasked with determining the cost between two nodes s and s' in Γ, R* does a weighted A* search from s to s' in the underlying state space. The cost of the solution that is returned is used as the edge cost in the sparse graph. A depiction of an R* search with $k = 3$ is shown in Figure 5-1. The nodes in both the sparse graph and along the low level paths are shown. Additionally, if the weighted A* search does not find a solution within some node expansion limit, it will give up on its attempt to connect the two sparse states and R* will instead focus the search elsewhere. In this way, R* solves the planning problem by carrying out searches that are

```
rstar_search()
1. loop until a goal is found or the entire graph has been visited:
2.     select the best unexpanded state $s \in \Gamma$
3.     if the edge betweem $parent(s)$ and $s$ has not been computed then
4.         try to compute the path
5.         if path computation failed then label $s$ as AVOID
6.         else
7.             update $g(s)$ based on $g(parent(s))$ and the cost of the path found
8.             if $g(s) > w \cdot h(s_{start}, s)$ then label $s$ as AVOID
9.     else
10.        let $SUCCS(s)$ be $k$ randomly generated states that are a distance $\Delta$ from $s$
11.        if the goal state is within $\Delta$ of $s$ then add it to $SUCCS(s)$
12.        foreach state $s' \in SUCCS(s)$, add $s'$ and the edge $s \to s'$ to $\Gamma$
```

Figure 5-2: High level R* pseudocode

not only much smaller than the original problem, but also easy to solve.

Since weighted A* is used, R* is guaranteed to return solutions with cost no worse than w times optimal in the high level graph for the heuristic weight w being used [29]. In their evaluation, [29] show that R* can solve motion planning problems on 6 and 20 degree of freedom robotic arms with very high success rates and reasonably low solution costs compared to ARA* [28] and RRT.

High level pseudocode of the R* algorithm is shown in figure 5-2. The main loop of the R* algorithm is similar to a best first search such as A*. First, the best node on the open list is removed (line 2). The ordering function for the open list first prefers nodes that haven't been labeled AVOID. It then prefers nodes with lower f value, with ties broken on lower h value. In R*, there are two types of nodes in Γ that can be popped off the open list. The node can either already have a low level path computed between its parent and it, or not. If a path hasn't been found, R* uses a bounded weighted A* search to find one. If the search succeeds, then the g cost of the node is updated to reflect the cost of the path that was found (line 7) and the search continues. If a path is not found, then the node expansion limit was reached, indicating that this subproblem may be hard to compute. In this case, the weighted A* search will return the cost of the best node on the frontier. This cost is used to update the g value of the node with a better estimate of the true path cost. The second case is that the node did already have a path to it. In this case, the node is just expanded (lines 9-12). This involves randomly generating k successors that are a distance Δ away (line 10). The goal state is also added to the list of successors if it is within this distance (line 11). These nodes and edges are then added to the sparse graph Γ and the search continues (line 12). More detailed pseudocode of the R* algorithm is given in Appendix A.

It is worth mentioning that in R*, the f values of the nodes in Γ are not strictly made up of h and g as in A*. While it is still conceptually used for the same purpose, it may be composed of several values. Let n and n' be two nodes in Γ where n is the parent of n'. The f value of n' will always contain the g value of n and the h value of n' to the goal. R* also includes an estimate of the cost of the low level path between n and n'. Initially, this is just the heuristic value between the two nodes. Note, that the heuristic used must be capable of estimating the cost of the path between any two nodes. Once R* attempts compute the low level path between n and n', this value will be updated (line 7). If a path is found, it will be the cost of the path. If a path is not found, due to the expansion limit, then the f value of the best low level node will be returned and this will be used as the new estimate of the cost of the path between n and n'.

5.2 Real-time R*

Real-time heuristic search algorithms deal with problem of not being able to plan complete paths to the goal by using information gathered from previous search iterations to escape from heuristic local minima and find a path to the goal. In the case of the robot planning domain where there is a large search spaces and high branching factors, the lookahead performed by traditional A* style real-time search, such as LSS-LRTA*, may not by able to see far enough into the future to make informed decisions about what action to take. One way to increase the depth of the A* lookahead used would be to reduce the branching factor of the search space by limiting the number of actions available to the robot. This however reduces the quality of plans returned, since fewer actions may be used, possibly even making the problem unsolvable. The size of the state space may also be reduced by increasing the size of the discretization used, for example in the size of the static obstacle or cost grid. This also has the adverse effect of reducing the quality of the plans returned and again possibly making the problem unsolvable. R* is able to deal with high dimensional state spaces by splitting the problem up into smaller, easier to solve subproblems. This does not reduce the size of the action set available to the robot, nor does it increase the size of the problem discretization.

While R* has been shown to perform well in many hard domains, it is not a real-time algorithm. R* must find complete paths to the goal on every search, and the time that this takes is not bounded to be real-time. This section will describe the changes needed to transform R* search into a real-time search, as well as other enhancements that were found to be useful in the robot planning domain.

5.2.1 Limiting Expansions

The most important quality of any real-time search algorithm is that it actually be able to perform in a consistent, real-time manner. This means always returning the next action to take before the time bound has expired. RTR* uses the approach taken by many real-time search frameworks [19, 17], of limiting the number of node expansions to some constant number. In RTR* there are two types of node expansions that happen during the search. Nodes in the sparse graph are expanded to generate a set of random successors. Nodes in the low level state space are expanded by weighted A* while computing a path between two nodes in the sparse graph. The former occurs relatively infrequently but takes more CPU time to set up the weighted A* search. The latter occurs much more frequently but each expansion takes little CPU processing comparatively. RTR* counts each low level expansion as one and each high level expansion as thirty to account for this difference, but this should be tuned based on the machine and the implementation being used. Once the expansion limit is reached, the best action to execute is returned as described below. Lastly, different from R*, RTR* does not terminate when a goal state is found. It only terminates when the expansion limit has been reached. This is because it isn't sufficient to just reach the goal state. In domains with moving obstacles, it may be necessary to move off the goal state at some future time to get out of the way of an obstacle.

5.2.2 Action Selection

After an iteration of RTR*, an action to perform must be selected. As in other real-time searches, RTR* picks the first action along the most promising looking path that has been generated. In most traditional A* style real-time searches, this corresponds to the best node on the open list. This approach cannot be taken directly in RTR* because the nodes on the open list are nodes in the sparse graph and they may not actually have a low level path to them. In order to prefer nodes that have complete paths to them, the ranking function used when selecting the best node on the open list is changed slightly. Nodes with complete paths to them are preferred first. Of these, nodes with smaller f' values are preferred. If there are no nodes in the sparse graph with complete paths to them, then nodes with partial paths to them are selected. These are nodes where the weighted A* search failed to find a complete path due to the node expansion limit. Instead the best node on open at the end of the weighted A* search is stored with the sparse node. Of these, nodes with lower f' values are once again preferred.

5.2.3 Geometrically Growing Expansion Limits

In the original R* algorithm, if the path to a node is not found due to the node expansion limit imposed, that node is labeled as AVOID and it is inserted back onto the open list. If the node is ever popped off of the open list again, another attempt is made at computing the path, this time with no node expansion limit. The weighted A* search is allowed to run until a solution is found. This subproblem could be very hard to solve, requiring an immense amount of search time. This will have very detrimental effects on a real-time search algorithm (or any algorithm for that matter). Instead of allowing the low level search to run with an unbounded limit of expansions to compute a path to a node labeled AVOID, the RTR* algorithm uses a geometrically increasing node expansion limit. Each time a search fails due to the expansion limit, the limit is doubled for that node the next time it is removed from the open list. In this way, the RTR* will not focus all of its effort on computing paths to hard to solve subproblems unless completely necessary, and even then, the paths to the easier of these hard problems will be computed first. Since the expansion limit for computing the path to a node is doubled each time, the total amount of extra searching that may need to be done is bounded by a constant factor in the worse case. In practice it should actually cause the search to expand much fewer nodes.

Theorem 2 *The total number of extra node expansions that must be done by R* because of doubling the expansion limit of a sparse node instead of solving the problem outright is bounded by a constant factor.*

Proof: Suppose there is a state s and its successor state s' in the sparse graph Γ. Suppose that R* must compute the path between s and s' to reach the goal. Let the number of low level nodes that must be expanded by weighed A* to compute this path be n. In the worse case, the series of weighted A* searches that uses a node expansion limit that doubles will expand $n-1$ nodes on its second to last iteration before expanding n nodes on its last iteration. In addition to these last two searches, the total number of nodes expanded by

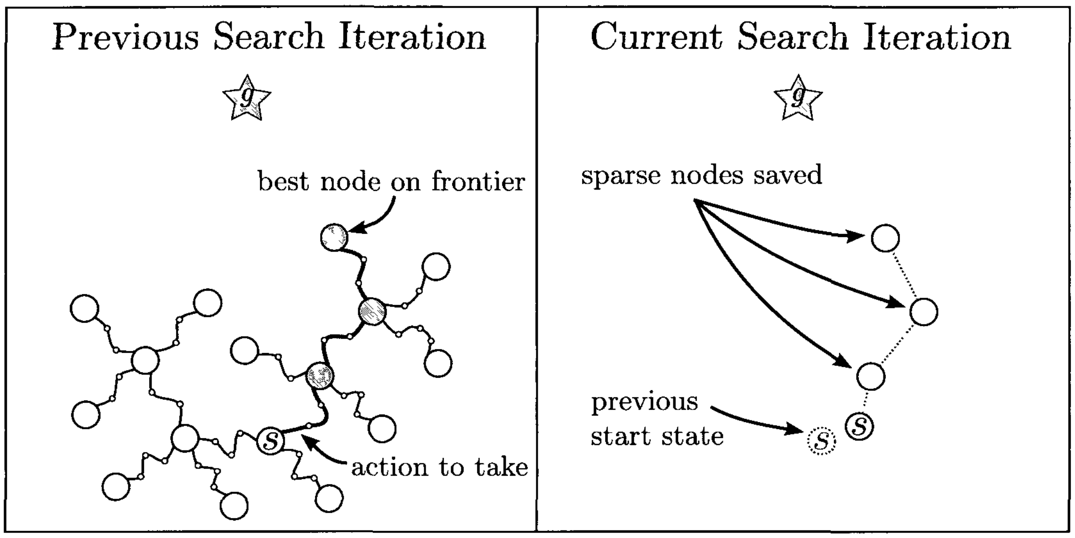


Figure 5-3: Path saving across iterations of RTR*.

weighted A* on all previous iterations will be:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... \approx n$$

So in total, $n + (n - 1) + n \approx 3n$ nodes will be expanded. Since a weighted A* search that does not use an expansion limit will visit n nodes, the overhead of using the doubling technique is bounded by a constant factor of approximately three in the worse case. Now suppose that there are k of these paths that must be computed in the whole problem. The number of nodes expanded using doubling will be $3nk$ in the worse case, versus nk when not using doubling, so the constant factor remains the same. □

5.2.4 Path Reuse

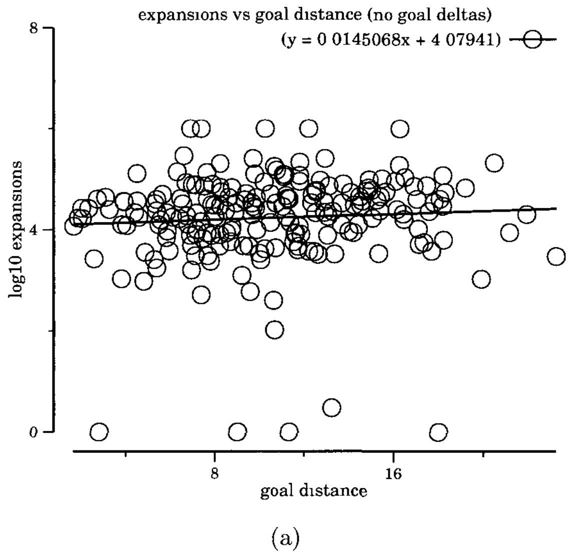
At the completion of a RTR* search, the best node on the frontier is determined and the action along the path to that node is returned. Since this is the best path that was found during the search, it seems reasonable to save this information to be used the next time the RTR* search is run. The only issue is that in the robot domain the search graph is dynamic due to the unpredictability of the moving obstacles. This means that the costs of the edges in the search graph can change between search iterations, so the cost of this path may become inaccurate. So instead of saving the low level path, which may have inaccurate costs, RTR* instead only saves the nodes in the sparse graph that are on the best path found. This allows the RTR* search to seed the sparse graph Γ with nodes that turned out to be good on the previous iteration. If the costs of the graph have not changed much, then these nodes will most likely still be favorable. If the graph has changed, for example a dynamic obstacle has unpredictably moved into the trajectory of the previous path, then RTR* will recompute a better path, or perhaps not even use those edges in the sparse graph at all. Figure 5-3 shows an example of how the path saving mechanism works across iterations. On the left, RTR* has reached its expansion limit and calculated the best node on the frontier and the corresponding action to take along the path (shown in red). The sparse nodes that exist along the path (green) are added to the initial sparse graph in the next iteration of searching, shown on the left. The dashed lines between the nodes on the left show the edges in the sparse graph and indicate that a low level path between them has not yet been computed.

5.2.5 Making Easily Solvable Subproblems

One of the key insights of the R* algorithm is that dividing the original problem up into many smaller subproblems is generally easier to solve than solving the original. In this section, it is shown that many of the assumptions made about generating random successors in the original R* algorithm do not hold in the robot motion planning domain.

Relaxing Goal Conditions

During the node expand process, R* generates successors by randomly sampling the state space at some specified distance Δ away from the node being expanded. R* as originally presented does not specify a certain distance metric, although Euclidean distance is often used. The goal of the R* top level expand process is to generate smaller, easier to solve subproblems. In certain domains, such as robot motion planning, shorter distance does not necessarily correspond to easier to solve problems. Due to the dynamic constraints of the



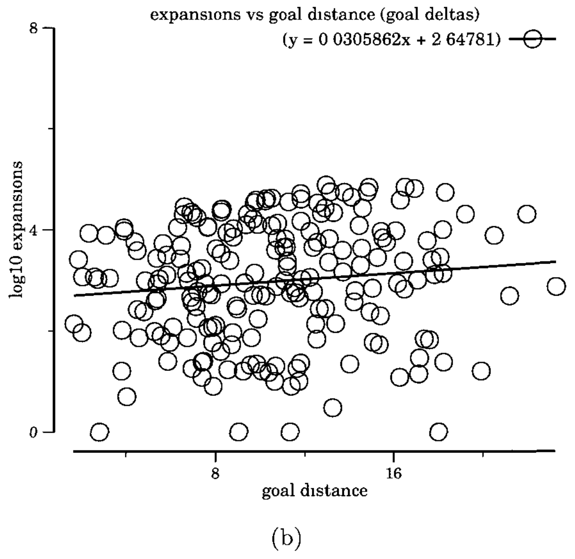


Figure 5-4: The log of the number of nodes needed by weighted A* to solve problems problems with and without a goal radius allowed. The x axis is the distance between the start and goal locations.

vehicle, it could actually be quite difficult to move to a state that is only a small Euclidean distance away. Consider the example of a car trying to parallel park. Even though the car needs to only physically move its location by a couple meters, the maneuver required to do this is quite complex. It was found that requiring RTR* to plan paths to the exact nodes in the sparse graph was prohibiting the search from exploring further into the search space. The reason is that although the start and goal nodes of these subproblems were closer together, it was often still very hard to maneuver the robot precisely onto a given state. Figure 5-4a shows the log of the number of nodes taken by weighted A* with a weight of 3 to compute the solutions to problems with random start and goal states. The map used is only 200 by 200 grid cells and there are no static or dynamic obstacles. Despite these ideal conditions, these problems are still quite difficult to solve, with only a slight correlation between the distance from the start to the goal node and how many node expansions are required to solve the problem. To make these problems easier, the goal condition used for the low level weighted A* searches was relaxed. Any state within some distance d of the actual goal state and with any heading and any speed is considered a goal. Figure 5-4b shows the results of allowing any state within 0.5 meters of the goal state still be considered a goal, provided that it still has the correct heading and speed. The number of nodes taken to solve these problems is on average more than an order of magnitude less. This results in considerable savings since RTR* must solve many of these small subproblems per planning iteration.

5.3 Parameter Tuning

The R*, and consequently the RTR*, algorithms both have several parameters that can be adjusted to get differing performance. The first of these parameters is the number of random successors to generate, k, for each sparse node expansion. Setting k high will result in a bushier search tree and better solutions since the sparse graph will be more likely to include nodes that lie on a better solution path. High values of k also have the adverse side effect of slowing the rate at which the search explores deeper into the search space. This is especially noticeable in real-time search where an expansion limit is used. The shape of the look-ahead will be broader or narrower depending on higher or lower values of k. The second parameter is the distance metric Δ that specifies how far away the random successors of a state should be. The value of Δ should be high enough to allow the search to spread out, but not too high that the subproblems become too hard to solve. The third parameter is the weight w used by the weighted A* that is used to connect states in the sparse graph with paths made up of the low level actions. The higher w is set, the more greedy the weighted A* search becomes, relying more and more on solely the heuristic to guide search. Higher values will also cause weighted A* to return more and more suboptimal solutions, but may drastically cut down on the number of node expansions needed by each weighted A* search. The fourth parameter is the node expansion limit that is the initial number of nodes that the weighted A* search can expand before giving up and labeling the node as AVOID. This value should be set high enough that the weighted A* search has chance at solving the easier problems of the domain. It shouldn't be so high that the RTR* search spends too much of its time trying to solve hard subproblems. Lastly, the goal radius parameter defines how close a state has to be to the goal state before it is considered a goal. Setting this too low will cause subproblems to be too difficult to solve, while setting it too high will cause RTR* to not be able to explore as deeply into the search space.

5.4 Results

The RTR* algorithm was evaluated using the simulator described in Chapter 3. In these experiments, the map shown in Figure 3-1 of Chapter 3 was used. The experiment was run with many different start and goal positions for the robot, 36 in total. The number of dynamic obstacles was also varied from zero up to ten opponents. The paths that the opponents follow are arbitrary paths traced by a human with a pointer device and then stored for reuse. The heuristic used for cost to go to the goal was the 2D Dijkstra heuristic. Since RTR* also needs a heuristic from any arbitrary node to any other arbitrary node, the straight line heuristic was used for this. The size of the map was 500 by 500 cells, corresponding to a 20 by 20 meter map. The cost of a time step passing while not on the goal was 5 and the cost of a collision was 1000.

The other algorithms that were also tested are PLRTA* [3], LSS-LRTA* [17], RTA* [19], Time-Bounded Lattice [21], and a real-time version of the RRT algorithm of [24], all of which are described in Chapter 4. The parameters used for each of the algorithms are as follows. PLRTA* was run with a lookahead of 1000 nodes, and a decay steps value of 4. LSS-LRTA* was run with a lookahead of 1000 nodes. RTA* was run with a lookahead depth limit of 4. Time-Bounded lattice was run with a time-bound of 4 seconds and a weight of 3. RRT was run with a sampling limit of 500 samples. RTR* was run with an expansion limit of 5000 nodes and an avoid limit of 1000 nodes. The value of k was set to 10 and w was set to 3. The Δ parameter was set to 0.4 meters.

Figure 5-5 shows a box plot for each algorithm tested when run with 0, 3, 6, and 10 dynamic obstacles. The y axis denotes the actual cost accrued by the agent running the algorithm as reported by the simulator. Here, we see that PLRTA* is clearly the best across the board. Its partitioned heuristic and learning scheme seem to offer it a great advantage when compared to LSS-LRTA* as it was originally presented. This is expected, as LSS-LRTA* was not designed to handle the type of search space that this problem entails. RTR* seems to perform comparably to LSS-LRTA*. RTR* performs comparatively worse on the experiment with no dynamic obstacles. It seems that RTR* is able to avoid hitting moving obstacles fairly well, but it is unable to quickly get to the goal, even if there are no dynamic obstacles. The real-time version of RRT performs the worse on all the experiments, not being able to reliably get to the goal with no dynamic obstacles and having a high collision rate when dynamic obstacles are present. Time-Bounded Lattice (TBL) performs well when there are few dynamic obstacles, even performing the best of all algorithms when there were no dynamic obstacles. It is unable to reliably avoid collisions in the presence of many obstacles however. Likewise, RTA* is able to perform ok when there are no dynamic obstacles, but it is unable to avoid collisions as the number of dynamic obstacles increases, performing about as badly as RRT.

Figure 5-6 shows a line plot of the total cost accrued for each algorithm across all numbers of dynamic obstacles tested. Error bars show 95% confidence intervals around this cost. This reinforces the ranking of algorithms view in Figure 5-5. PLRTA* performs the best with RTR* and LSS-LRTA* performing comparably and second best. This plot also includes the original R* algorithm which is not real-time. As seen in the plot, the improvements made to create RTR* greatly improve performance. RTR* outperforms R* by a considerable margin.

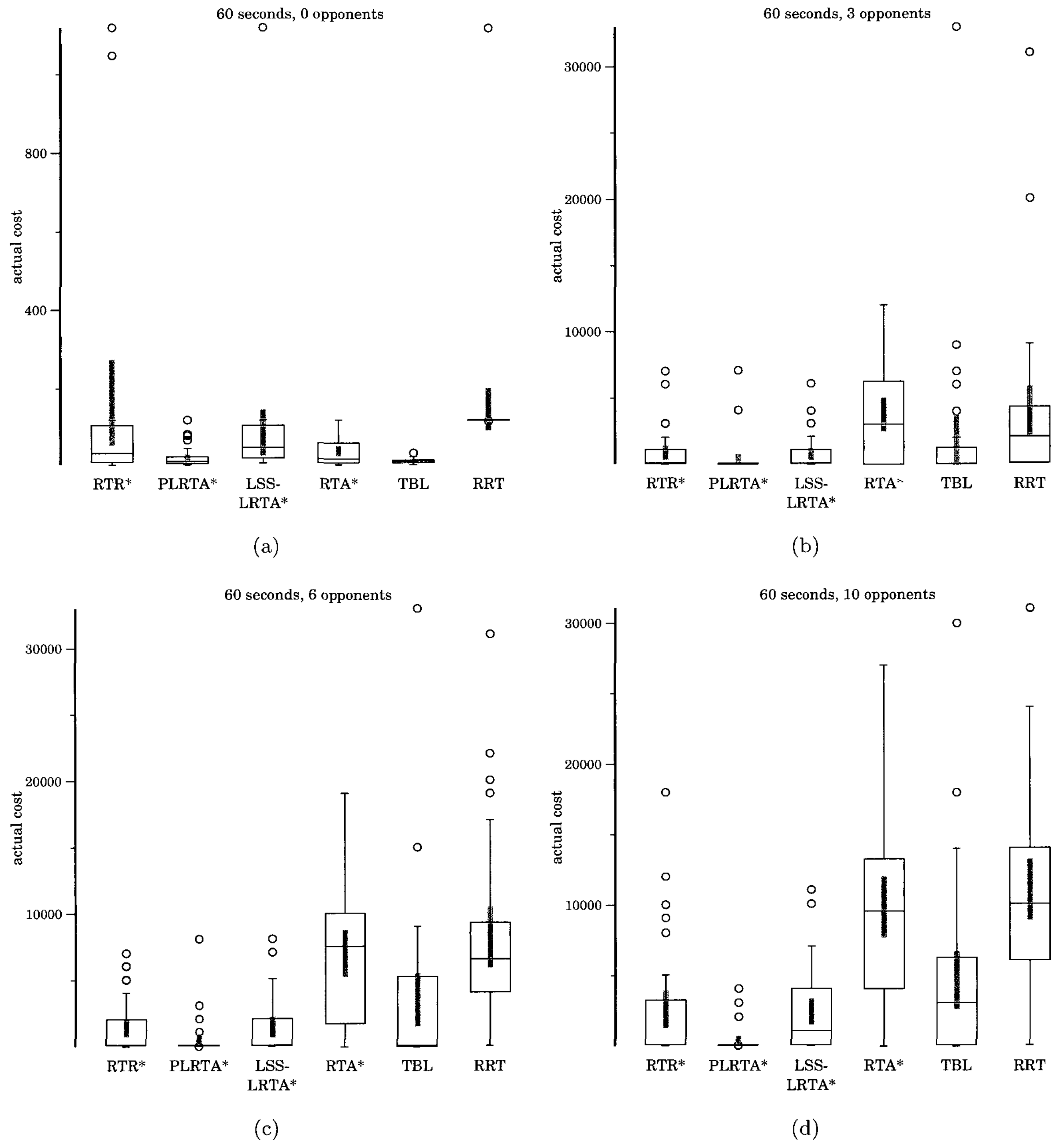


Figure 5-5: Experiments of length 60 seconds with the number of moving obstacles ranging from zero to ten. Actual cost is the total cost accrued as reported by the simulator.

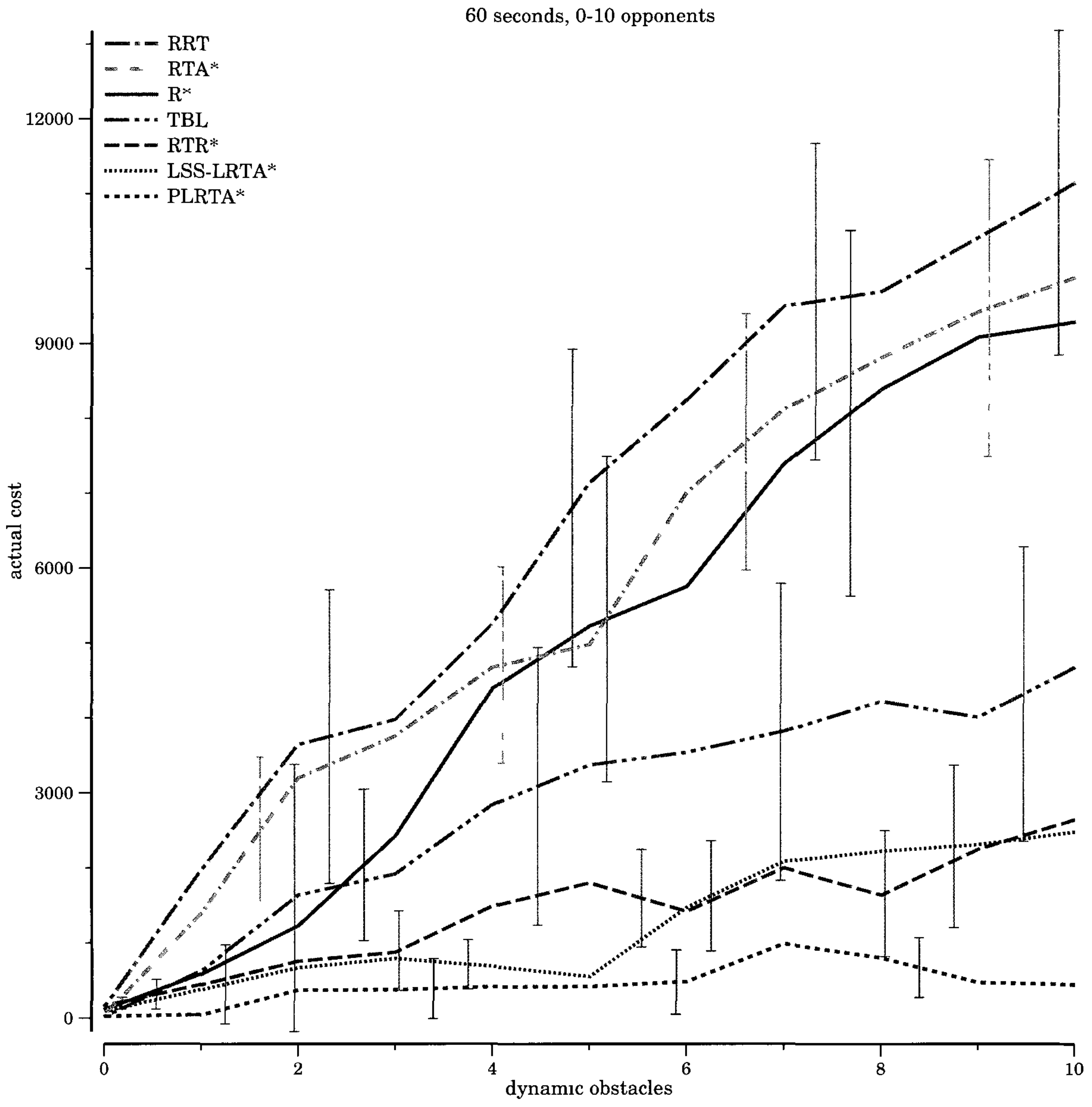


Figure 5-6: Experiments of length 60 seconds. Shows actual cost accrued versus number of dynamic obstacles for each algorithm with 95% confidence intervals.

CHAPTER 6

CONCLUSIONS

The purpose of this thesis was to investigate techniques for solving the problem of real-time, kinodynamically feasible motion planning for mobile robots among moving obstacles. A new real-time algorithm called Real-time R* was developed, based on previous work involving heuristic search and random sampling. Its development was motivated by the need for a real-time search method that could handle the high dimensionality of the state space that exists for this problem. This thesis shows how to transform the original R* algorithm to a real-time algorithm (RTR*). This involved making several modifications that allowed RTR* to satisfy the definition of a real-time algorithm. Modifications that improved performance specifically for this domain were also discussed.

Real-time R* was evaluated alongside several other algorithms in a simulated environment. These algorithms included state-of-the-art real-time search algorithms, state-of-the-art motion planning algorithms dealing with moving obstacles, and state-of-the-art sampling algorithms, all commonly used to solve hard problems in robotics and AI. In the evaluations, RTR* was seen to perform comparably to LSS-LRTA*, the previous state-of-the-art real-time search algorithm. It also vastly outperformed the original R* algorithm. The only algorithm that clearly outperformed RTR* was the recently developed PLRTA*, an algorithm based on LSS-LRTA* and developed specifically for the domain at hand.

The success of PLRTA* for this type of problem shows the advantage of partitioning the costs associated with the problem into static costs and dynamic costs, allowing the search algorithm to treat each differently. One type of problem where RTR* could have an advantage over PLRTA* is where the state space is very large, perhaps due to a small action duration, and the branching factor is also very large, due to many actions being available to the robot. This is because PLRTA* is still constrained to perform an A* shaped lookahead and may not be able to cope with the large numbers of nodes that exists in a state space like this.

6.1 Future Directions

While RTR* didn't perform the best of all algorithms evaluated, there are several techniques that could possibly improve RTR* or real-time search algorithms for this problem in general. This section will describe these ideas in detail as well as reasons why they may be useful for solving the problem of real-time motion planning among moving obstacles.

6.1.1 Depth-based Random Successor Generation

For this domain, techniques discussed in section 5.2.5 were used to make the subproblems encountered in RTR* easier to solve. The goal test was relaxed to allow nodes within a certain radius of the goal state to still be considered goals. While this certainly makes the subproblems in RTR* easier to solve, these problems can still be very hard, even with no moving obstacles in the way. This is because of the motion model that constrains the robot. The assumption that is made when generating successor nodes within a certain radius of their parent node is that because the distance between the start and goal node of this subproblem is shorter, the problem will be easier to solve. This is not actually the case depending on the type of vehicle being used and its motion constraints. Another way to generate the random successors of a state is to not sample from all the states a certain distance away, but to sample from all the states a certain number of actions away. The idea is to do a breadth-first search out from the state to some depth d, using the motion primitives available to the robot. Then, select a state residing at depth d at random and use this as the successor to the node. This could allow for easier subproblems, since instead of making the assumption that shorter distance equals easier to solve, the motion model of the robot is used to sample from states that are known to only be d actions away from the robot's current state. Depending on the static and dynamic obstacles of the world, the actual distance may be greater than d, so the goal radius approach of section 5.2.5 should still be used to allow for more flexibility in the plans generated.

Since it would be much too costly to do a breadth-first search out from each node that we need to generate successors for, this can be precomputed and all the nodes at the depth limit can be stored in an array. To sample the nodes, a random array index is picked. To generate the states to put in the array, the breadth-first search should be done from a robot at location $(0, 0)$ with a heading of $0\,^{\circ}$. A separate array must be made for each possible speed that the robot can have. Then, when tasked with picking a random successor for some state at location (x, y) with heading θ and speed v, the array corresponding to the speed v is used. A state is picked at random and is then translated by the position (x, y) and heading θ of the robot to reflect this state.

6.1.2 Breadth-first Style Look-ahead

Most real-time heuristic search variants complete the lookahead phase of their search by performing a limited A* expansion from the current state. When a complete path to the goal is not found, an action to take is selected based on the information provided by the nodes generated during the lookahead phase. If the lookahead phase is seen as taking samples of the nearby search space to base decisions off, then it will be biased by the heuristic used during A*. One way to get rid of this bias is to not do an A* search, but instead to do a breadth-first search to a specified depth-bound. This has the advantage of eliminating the heuristic bias during the lookahead phase. The disadvantage to this approach is that possibly many more bad states may be visited because the heuristic can't be used for pruning.

6.1.3 Local Search Variations

RTR* as it was presented in this thesis uses weighted A* for both the top-level and low-level search algorithm. Through observing the performance of RTR* on several different experiments, it seemed that one of its main weaknesses was not being able to plan low-level paths to connect the sparse states reliably. This could possibly be fixed by using different algorithms to compute the low-level paths. Using RRT based algorithms could help speed up this portion of the algorithm at the expense of solution quality, since they are sampling based. However, since the top-level search of RTR* is still heuristic and cost based, there will still be attention given to finding low cost solutions.

Some possible algorithms to use for the low-level search algorithm are RRT, MA-RRT, and RRT*. RRT would most likely allow for finding solutions the quickest, but also of the worst cost. It would then be up to the top-level of the RTR* algorithm to filter out bad paths. The MA-RRT would probably provide the best middle ground performance, finding solutions quickly, but also paying attention to costs. RRT* has been shown to compute much better solutions than RRT but also takes longer to converge on these solutions. An anytime version of RRT* [13] could perhaps be used to manage this trade-off.

6.1.4 Learning

Another quality of real-time search that is desirable is that given enough time, it will be able to escape a local minima caused by heuristic error and limited lookahead. While the sparse level state space used by R* allows it to escape local minima more efficiently than a normal A* style search, given a sufficient sized local minima or a sufficiently small lookahead, R* is still susceptible to getting stuck in a local minima. Real-time search methods solve this by raising the h value of states to reflect previous actions that have already been taken. With R*, doing this approach at the low level is not practical, as the open and closed lists used during the low level search are thrown away at the completion of the search. RTR* could benefit from learning if done at the sparse node level, however it is not obvious how to accomplish this.

6.1.5 Real-time Search with Inadmissible Heuristics

Inadmissible heuristics are heuristics that may overestimate the cheapest cost path to the goal from a state. While using inadmissible heuristics breaks many of the guarantees provided by heuristic search algorithms, they can none the less be very useful to search. An inadmissible heuristic that could prove to be very beneficial for planning in this domain is one that captures the costs of collision with dynamic obstacles, since current heuristics used only capture the cost of not being on the goal.

Dynamic Obstacle Heuristic

A heuristic that could greatly improve search performance is one that could capture the costs of the dynamic obstacles as they move throughout the world. Since the costs associated with dynamic obstacles are parameterized by time, the heuristic would have to take this into account. It is not clear how to create an admissible heuristic that is capable of doing this while also being relatively easy to compute. If the admissibility requirement is dropped, it becomes much easier to design such a heuristic.

The objective of this heuristic is as follows: given a robot state s at time t, return a heuristic that captures the dynamic obstacle cost of getting to a goal state g at some time $t' \geq t$. The main idea of this heuristic is to do a Dijkstra search over a small set of abstract states parameterized by location and time. First, to make the heuristic easy to compute, the world must be divided into large grid cells. Next, a time step parameter t_{step} is defined. This will be the amount of time that passes when transitioning from one grid cell to another and its value should reflect the size of the grid cells and the speeds that the robot is capable of. Lastly, a maximum number of time steps, k is specified. This will define the maximum time $t_{max} = k \cdot t_{step}$. This is the maximum future time that the Dijkstra search will consider.

The Dijkstra search will take place over a three dimensional matrix as shown in Figure 6-1, where the dimensions are x position, y position, and time t. The vertices of the graph

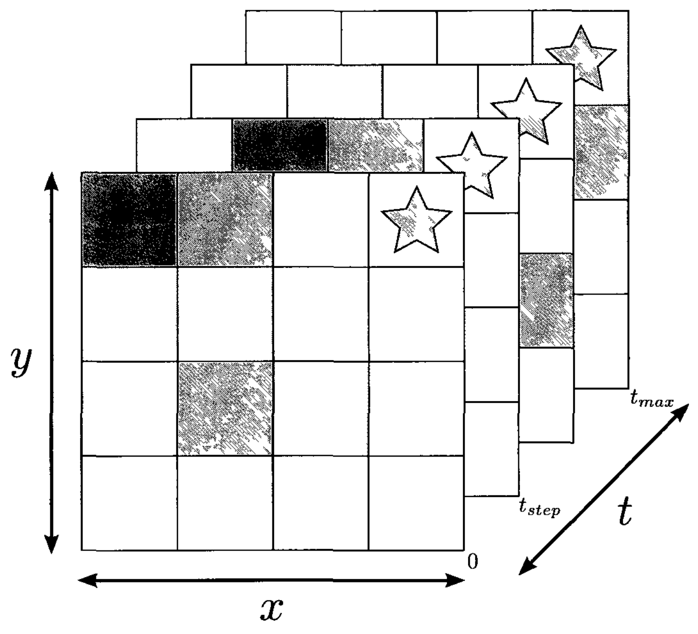


Figure 6-1: Visualization of an inadmissible heuristic that would capture the costs of the dynamic obstacles over time.

used by the Dijkstra search consist of each cell position at each time t where $t = \imath \cdot t_{step}$ for $0 \leq \imath \leq k$. The edges connect each vertex with vertices of the adjacent cells at one time step in the future. The edges are directed, as it should not be possible to traverse backwards in time. The costs associated with each vertex, shown in the figure as shades of red, can be computed using the dynamic obstacle costs of the cell. The cost of traversing an edge from v to v' is defined as the cost associated with vertex v'. Let G be the set of cells that include the goal state, across all time steps up to t_{max}. The weight that the Dijkstra associates with each of these vertices should be set to the grid cost of the vertex. The weight of all other vertices should be set to infinity. After running Dijkstra's algorithm, the weight of each vertex will correspond to the cheapest cost path of reaching one of the goal states. During search, the heuristic value of a state s at time t is looked up by determining which cell that state falls in. The time value used should be the one defined during the Dijkstra search that is closest to t.

Effects on Search

Using inadmissible heuristics with search can greatly boost search performance by providing a more realistic estimate of the cost to go . While many search algorithms rely on admissible heuristics for certain optimality guarantees, some of the same guarantees can still be made using an inadmissible heuristic if an admissible heuristic is also available to the search algorithm [37]. These techniques are not applied to real-time search however, as the search algorithms must find complete paths to the goal. In order to use inadmissible heuristics with real-time search, a technique would have to be developed to avoid the situation where the inadmissible heuristic would always prevent the real-time search algorithm from reaching the goal. This could perhaps be accomplished by using the inadmissible heuristic in tandem with an admissible heuristic and assigning a weight to each. Early on in the search, the weights are such that only the inadmissible heuristic is used. As the agent revisits states however, the weights could be updated to increasingly rely on the admissible heuristic. More work must be done in this area to determine the effects of using inadmissible heuristics with real-time search algorithms.

APPENDICES

APPENDIX A

R* PSEUDOCODE

Pseudocode for the main R* search loop:

```
procedure rstar($s_{start}, s_{goal}$)
1. $OPEN \leftarrow \emptyset$, $CLOSED \leftarrow \emptyset$
2. $g(s_{start}) \leftarrow 0$
3. insert $s_{start}$ into $OPEN$
4. while $OPEN \neq \emptyset$ AND $priority(s_{goal}) \geq \text{argmin}_{s' \in OPEN}\,(priority(s'))$ do
5.     remove $s$ with the smallest priority from $OPEN$
6.     if $s \neq s_{start}$ AND no local path to $s$ has been found then
7.         reevaluate($s$)
8.     else
9.         expand($s$)
10. return incumbent solution if found, impossible otherwise
```

Pseudocode for the reevaluate function of R*:

```
procedure reevaluate($s$)
11. $path(parent(s), s), pathcost(parent(s), s) = computeLocalPath(parent(s), s)$
12. if $path = \texttt{null}$ OR $g(parent(s)) + cost > w \cdot h(s_{start}, s)$ then
13.     $avoid(s) \leftarrow$ true
14.     $parent(s) \leftarrow \text{argmin}_{s' | s \in SUCCS(s')}\,(g(s') + pathcost(s', s))$
15. $g(s) \leftarrow g(parent(s)) + pathcost(parent(s), s)$
16. insert/update $s$ in $OPEN$
```

Pseudocode for the expand function of R*:

```
procedure expand($s$)
17. if $is\_goal(s)$ AND $s$ is better than current incumbent solution then
18.    set $s$ as the current incumbent solution
19. insert $s$ into $CLOSED$
20. $SUCCS(s) \leftarrow$ the set of $k$ randomly chosen states a distance $\Delta$ from $s$
21. if $distance(s, s_{goal}) \leq \Delta$ then
22.    $SUCCS(s) \leftarrow SUCCS(s) \cup s_{goal}$
23. $SUCCS(s) \leftarrow SUCCS(s) - SUCCS(s) \cap CLOSED$
24. foreach $s' \in SUCCS(s)$ do
25.       $pathcost(s, s') \leftarrow h(s, s')$
26.    if $s'$ hasn't been generated before OR $g(s) + h(s, s') < g(s')$ then
27.       $parent(s') \leftarrow s$
28.       $g(s') \leftarrow g(s) + pathcost(s, s')$
29.       insert/update $s'$ on $OPEN$
```

APPENDIX B

COMMUNICATION PROTOCOL

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

B.1 Initialization:

B.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.

B.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.

B.2 Operation:

B.2.1 Simulator to Agent

- **state time goal num-dyn-obstacles dyn-obstacles**: This message tells the agent what 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.

B.2.2 Agent to Simulator

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

APPENDIX C

COMFIGURATION FILE SPECIFICATION

An example configuration file that is used by the simulator is shown here. The first section corresponds to global parameters for the simulator and the world model. The second section corresponds to parameters for a specific robot agent in the simulation. Each line of the schema includes first the field name, then the field type, then the default value of the field.

```
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 D

DIVISION OF LABOR

Much of the work that went into the infrastructure that made the experiments used in this thesis possible was split between Jarad Cannon and myself. This includes creating the simulator, the experiments and plotting system, the problem domain and heuristics, and implementing other relevant algorithms to compare against. The breakdown of who was responsible for different parts of the project is as follows: the main portions of the project that Jarad was responsible for are:

1. Writing the communication protocol for the simulator.
2. Writing the manager module for the simulator.
3. Writing the opponent modeling used by the simulator.
4. Writing generation and parsing code for the configuration files used by the simulator.
5. Keeping track of statistics during the simulation.
6. Writing the planner module of the simulator and interface for it to call the different search algorithms used.
7. Implementing RTA*.
8. Implementing LSS-LRTA*.
9. Implementing PLRTA*.
10. Implementing Time-Bounded Lattice.

The main portions of the project that I was responsible for are:

1. Creating the motion primitive representation.
2. Creating a program to automatically generate physically accurate motion primitives according to a set of vehicle and motion parameters.
3. Creating a program to allow users to trace paths for dynamic obstacles to follow during simulation.
4. Creating and optimizing the search domain.
5. Writing the heuristics used for search.
6. Writing the framework for running experiments, including parsing output from the simulator, storing the data, and plotting the results.
7. Writing the rendering module of the simulator.
8. Writing the graphics code for the simulator.
9. Writing the collision checking code for the simulator.
10. Writing the agent and world models for the simulator.
11. Implementing proper thread synchronization in the simulator.
12. Implementing the RRT algorithm.
13. Implementing the R* algorithm.
14. Implementing the RTR* algorithm.

BIBLIOGRAPHY

[1] David M. Bond, Niels A. Widger, Wheeler Ruml, and Xiaoxun Sun. Real-time search in dynamic worlds. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 16–22, 2010.

[2] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2383–2388, 2002.

[3] Jarad Cannon. Robot motion planning using real-time heuristic search. Master's thesis, University of New Hampshire, December 2011.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[5] D. Ferguson, T.M. Howard, and M. Likhachev. Motion planning in urban environments: Part ii. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1070–1076, September 2008.

[6] Dave Ferguson and Anthony Stentz. Field d*: An interpolation-based path planner and replanner. In Sebastian Thrun, Rodney Brooks, and Hugh Durrant-Whyte, editors, *Robotics Research*, volume 28 of *Springer Tracts in Advanced Robotics*, pages 239–253. Springer Berlin / Heidelberg, 2007.

[7] S.S. Ge and Y.J. Cui. New potential functions for mobile robot path planning. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 16(5):615–620, October 2000.

[8] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[9] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2):107–113, April 1993.

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

[11] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[12] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *Robotics: Science and Systems (RSS)*, 2010.

[13] S. Karaman, M.R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the rrt*. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1478–1483, May 2011.

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

[15] Jongwoo Kim and J.P. Ostrowski. Motion planning a aerial robot using rapidly-exploring random trees with dynamic constraints. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 2200–2205, September 2003.

[16] Nathan Koenig and Andrew Howard. Gazebo - 3d multiple robot simulator with dynamics. `http://playerstage.sourceforge.net/index.php?src=gazebo`, 2003.

[17] Sven Koenig and Xiaoxun Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18:313–341, June 2009.

[18] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1398–1404, April 1991.

[19] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[20] James J. Kuffner and Steven M. Lavalle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE/RSJ International Conference on Robotics and Automation (ICRA)*, pages 995–1001. 2000.

[21] Aleksandr Kushleyev and Maxim Likhachev. Time-bounded lattice for efficient planning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4303–4309, Piscataway, NJ, USA, 2009. IEEE Press.

[22] Y. Kuwata, G.A. Fiore, J. Teo, E. Frazzoli, and J.P. How. Motion planning for urban driving using rrt. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1681 –1686, September 2008.

[23] S.M. LaValle and Jr. Kuffner, J.J. Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 473–479, 1999.

[24] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning.

Technical Report TR 98-11, Computer Science Department at Iowa State University, 1998.

[25] Jinhan Lee, C. Pippin, and T. Balch. Cost based planning with rrt in outdoor environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 684–689, September 2008.

[26] Maxim Likhachev and David Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *International Journal of Robotic Research*, 28:933–945, 2009.

[27] Maxim Likhachev, David Ferguson, Geoffrey Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[28] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference (NIPS-03)*. MIT Press, 2004.

[29] Maxim Likhachev and Anthony Stentz. R* search. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 1, pages 344–350. AAAI Press, 2008.

[30] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Publishing Co., 1971.

[31] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, 2011.

[32] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3-4):193–204, 1970.

[33] John H. Reif. Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 421–427, October 1979.

[34] E. Rimon and D.E. Koditschek. Exact robot navigation using artificial potential functions. *IEEE Transactions on Robotics and Automation*, 8(5):501–518, October 1992.

[35] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1652–1659, 1995.

[36] Jordan Thayer. Search visualizations. `http://www.cs.unh.edu/~jtd7/stills/index.html`, 2011.

[37] Jordan Thayer and Wheeler Ruml. Finding acceptable solutions faster using inadmissible information. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 2010.

[38] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, September 2005.

[39] Paul Vernaza, Maxim Likhachev, Subhrajit Bhattacharya, Sachin Chitta, Aleksandr Kushleyev, and Daniel D. Lee. Search-based planning for a legged robot over rough terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2380–2387, 2009.