We hereby approve the thesis/dissertation of

Simon A. Carroll

candidate for the **Master of Science** degree *

(signed) Michael S. Branicky

(chair of the committee)

M. Cenk Cavusoglu

GQ Zhang

(date) 05/06/2008

*We also certify that written approval has been obtained for any proprietary material contained therein.*
# Table of Contents

Table of Contents ................................................................. 1
List of Algorithms ....................................................................... 3
List of Figures ............................................................................. 4
List of Tables ............................................................................... 5
Acknowledgements ...................................................................... 6
Abstract ..................................................................................... 7
Chapter 1 .................................................................................... 8
Introduction .................................................................................. 8
  1.1 Problem Overview .............................................................. 8
  1.2 Thesis Contributions .......................................................... 9
  1.3 Thesis Outline ....................................................................... 11
  1.4 Experimental Data ............................................................. 12
Chapter 2 .................................................................................... 13
Background .................................................................................. 13
  2.1 Hybrid Systems Background ............................................... 13
    2.1.1 Hybrid Automata .......................................................... 14
    2.1.2 Examples of Hybrid Systems ......................................... 15
  2.2 Rapidly-Exploring Random Trees Background ....................... 17
    2.2.1 The RRT Algorithm ...................................................... 18
    2.2.2 Properties of the RRT .................................................. 20
    2.2.3 Modifications to the RRT Algorithm ............................. 22
    2.2.4 RRT Terminology Specific to this Thesis ....................... 32
  2.3 Explanation of Metrics ......................................................... 33
    2.3.1 Dispersion .................................................................. 34
    2.3.2 Kim Metric ................................................................... 34
    2.3.3 Reasons for Two Metrics ............................................. 35
    2.3.4 Average Dispersion .................................................... 36
    2.3.5 Normalized Kim Metric ............................................... 37
  2.4 Summary of Examples ........................................................ 37
    2.4.1 Stair Climber ............................................................... 38
    2.4.2 Train Gate ................................................................. 43
    2.4.3 Hovercraft ................................................................. 50
    2.4.4 Pendulum ................................................................. 54
    2.4.5 Helicopter ................................................................. 56
Chapter 3 .................................................................................... 61
Strategies for Dealing with Heavy-Tailed Examples ....................... 61
  3.1 Heavy-Tailed Behavior ....................................................... 62
  3.2 Background ......................................................................... 66
    3.2.1 Algorithm ................................................................. 66
    3.2.2 Cutoff Determination ................................................ 67
    3.2.3 Terms and Definitions ............................................... 71
    3.2.4 Measurement and Comparison .................................... 72
  3.3 Large versus Many Trees ................................................... 73
    3.3.1 Hovercraft ............................................................... 75
    3.3.2 Train Gate ............................................................... 82
<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td><strong>BUILD RRT</strong> <em>(start, goal)</em></td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td><strong>NEAREST NEIGHBOR</strong> <em>(x_rand)</em></td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td><strong>BUILD RRT</strong> <em>(start, goal)</em> with Success Termination</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td><strong>SAMPLE STATE ()</strong> with Biasing</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td><strong>RRT</strong> with Biasing and Obstacles</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td><strong>NEW NODE</strong> <em>(x_rand, x_near)</em></td>
<td>31</td>
</tr>
<tr>
<td>2.7</td>
<td><strong>Kim Metric</strong></td>
<td>35</td>
</tr>
<tr>
<td>2.8</td>
<td><strong>Average Dispersion</strong></td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td><strong>NEW NODE</strong> <em>(x_rand, x_near)</em> for Stair Climber</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td><strong>BUILD RRT</strong> <em>(start, goal)</em> with Success Termination and Restarts</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td><strong>Forward FOREST ()</strong> Algorithm</td>
<td>101</td>
</tr>
<tr>
<td>4.2</td>
<td><strong>Bidirectional FOREST ()</strong> Algorithm</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td><strong>Receding Horizon Search for Different Models</strong></td>
<td>123</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Car Example Hybrid Automaton</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>RRT Growth in 2 Dimensions</td>
<td>17</td>
</tr>
<tr>
<td>2.3a</td>
<td>Voronoi Regions of Pseudo-Random Points and Lattice Points</td>
<td>21</td>
</tr>
<tr>
<td>2.3b</td>
<td>Voronoi Regions of Each Node in an RRT at Three Stages of Growth</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Shorten-the-Step Strategy</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>The Problem with Dispersion</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>Stair Climber Example RRT</td>
<td>42</td>
</tr>
<tr>
<td>2.7</td>
<td>Another Stair Climber Example RRT</td>
<td>42</td>
</tr>
<tr>
<td>2.8</td>
<td>Train Automaton</td>
<td>44</td>
</tr>
<tr>
<td>2.9</td>
<td>Gate Automaton</td>
<td>45</td>
</tr>
<tr>
<td>2.10</td>
<td>Controller Automaton</td>
<td>46</td>
</tr>
<tr>
<td>2.11</td>
<td>Train Gate Example RRT</td>
<td>47</td>
</tr>
<tr>
<td>2.12</td>
<td>The Hovercraft</td>
<td>51</td>
</tr>
<tr>
<td>2.13</td>
<td>Hovercraft Example RRT</td>
<td>52</td>
</tr>
<tr>
<td>2.14</td>
<td>Pendulum Example RRT</td>
<td>56</td>
</tr>
<tr>
<td>2.15</td>
<td>Helicopter Straight Move</td>
<td>59</td>
</tr>
<tr>
<td>2.16</td>
<td>Helicopter Left Turn</td>
<td>59</td>
</tr>
<tr>
<td>2.17</td>
<td>Helicopter Right Turn</td>
<td>59</td>
</tr>
<tr>
<td>2.18</td>
<td>Helicopter Up Move</td>
<td>59</td>
</tr>
<tr>
<td>2.19</td>
<td>Helicopter Down Move</td>
<td>59</td>
</tr>
<tr>
<td>2.20</td>
<td>Helicopter with Obstacles</td>
<td>60</td>
</tr>
<tr>
<td>3.1</td>
<td>Survivor Plot of a Regular Distribution</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>Survivor Plot of a Heavy-Tailed Distribution</td>
<td>64</td>
</tr>
<tr>
<td>3.3</td>
<td>Coverage Regions</td>
<td>72</td>
</tr>
<tr>
<td>3.4</td>
<td>Survivor Plot for Hovercraft Example</td>
<td>82</td>
</tr>
<tr>
<td>3.5</td>
<td>Survivor Plot for Train Gate Example</td>
<td>88</td>
</tr>
<tr>
<td>3.6</td>
<td>Survivor Plot for Pendulum Example</td>
<td>90</td>
</tr>
<tr>
<td>3.7</td>
<td>Survivor Plot for Stair Climber Example</td>
<td>93</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of a Forest of Trees to its Graph</td>
<td>100</td>
</tr>
<tr>
<td>4.2</td>
<td>Stair Climber Problem: 1GB Strategy</td>
<td>108</td>
</tr>
<tr>
<td>4.3</td>
<td>Stair Climber Problem: 1FB Strategy</td>
<td>110</td>
</tr>
<tr>
<td>4.4</td>
<td>Stair Climber Problem: MFB Strategy</td>
<td>111</td>
</tr>
<tr>
<td>4.5</td>
<td>Stair Climber Problem: 1BiDi Strategy</td>
<td>113</td>
</tr>
<tr>
<td>4.6</td>
<td>Stair Climber Problem: MBiDi Strategy</td>
<td>114</td>
</tr>
<tr>
<td>5.1</td>
<td>An Example Path to be Invalidated Later</td>
<td>124</td>
</tr>
<tr>
<td>5.2</td>
<td>Scrap-the-Remainder Strategy, Before and After Pruning</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>Saving Parts Strategy, Before and After Pruning</td>
<td>128</td>
</tr>
<tr>
<td>5.4</td>
<td>Pruning Regions Strategy, Before and After Pruning</td>
<td>129</td>
</tr>
<tr>
<td>5.5</td>
<td>Unrealistic Hovercraft Movement</td>
<td>139</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Explanation of Distance Metric ................................................. 39
3.1 Cutoff Finding Example ................................................................. 69
3.2 Max Tree Size of 3000 ................................................................. 70
3.3 Max Tree Size of 6000 ................................................................. 70
3.4 Max Tree Size of 9000 ................................................................. 70
3.5 Max Tree Size of 12000 ............................................................... 70
3.6 Max Tree Size of 15000 ............................................................... 70
3.7 Max Tree Size of 18000 ............................................................... 70
3.8 Max Tree Size of 21000 ............................................................... 70
3.9 Success Rates as a Function of Step Sizes ..................................... 78
3.10 Dispersion vs. Number of Trees in Group .................................... 80
3.11 Average Dispersion vs. Number of Trees in Group ..................... 80
3.12 Normalized Kim Metric vs. Number of Trees in Group ................ 81
3.13 Success Rate vs. Problem Configuration for the Train Gate ........... 87
3.14 Success Rates at Each Cutoff Size for Both Strategies ................. 95
4.1 Performance Metrics for the Various Trees .................................. 116
4.2 Performance Metrics for the Various Trees, Close-up from 0-5 ....... 116
5.1 Number of Failures at Each Replanning ..................................... 134
5.2 Percentage Likelihood of Failure at Each Replanning ................. 135
Acknowledgements

I would like to thank my advisor, Dr. Michael S. Branicky, for his support, guidance, suggestions and dedication in helping me complete this thesis. I think Professor Branicky was the perfect advisor for me. Without his continued advice and assistance, this thesis would not have been possible.

I would also like to thank David Pawlowski for his role in many peer discussions, as well as helping to create the Helicopter example featured in Chapter 5. Dave was a superb colleague, and being able to work with him made this process much more rewarding and enjoyable.

Also, the National Science Foundation deserves mention, for their support of this research under grant CCR-0208919.

Finally, I would like to thank all of my friends and family who have supported me throughout this process to its completion. My fiancée Chantia deserves special mention. Without her consistent love, compassion, and encouragement, I would never have been able to accomplish this task.
Strategies for Improving Verification Techniques for Hybrid Systems

Abstract

by

Simon A. Carroll

In this thesis, we demonstrate techniques to improve upon the Rapidly-exploring Random Tree (RRT) as a tool for planning and verification of hybrid systems.

First, we perform experiments that show many planning/verification problems exhibit heavy-tailed behavior, where sampling-based algorithms sometimes require an inordinate number of nodes to solve them. We show that using restarts and multiple trees improves their solution time.

Second, we note that for many complex planning/verification problems the hybrid state space admits a natural separation into distinct modes, such that search in one does not help find a path through any other. We use a forest of trees, each tasked with solving a specific mode, to find overall solutions more quickly and with fewer nodes.

Third, we solve problems with unpredictable environment changes (because of other agents, unmodeled dynamics, or disturbances) using receding horizon search, where a new path is generated whenever the current path is invalidated.
Chapter 1

Introduction

1.1 Problem Overview

The purpose of this thesis is to improve upon sampling-based reachability algorithms, such as the Rapidly-exploring Random Tree (RRT), as tools for the planning and verification of hybrid systems. A *hybrid system* is one that combines continuous and discrete states, dynamics, inputs and outputs. Many real world systems can be modeled as hybrid systems including relays, switching electronics, and the power grid; thermostats, heating and air conditioning, and smart building control; automobile powertrains and automated highway systems; airplane autopilots and air traffic management systems; cell switching, neuronal networks, cardiac bioelectricity, and biological regulatory networks; plus embedded, real-time and networked control systems (see [3] and Section 2.1.2 for references). Since so many real world systems can be modeled as hybrid systems, extending the reach of algorithms like the RRT has the potential to improve the analysis and understanding of many problems of current interest.

A central question in the planning and verification of hybrid systems is the *reachability problem*. This problem asks: “Given an initial configuration, is it possible to reach a specific other configuration by following the dynamics of the hybrid system?” The answers to this problem have many different, important meanings, depending on the context in which the question is asked. For example, this question can be used to study stability and safety-critical properties of hybrid systems by asking whether or not the system will ever be able to reach some undesirable state. It can also be framed to confirm
liveness and performance properties by showing that the given hybrid system is capable of moving from an initial state to some desired goal set.

Currently, there are some types of problems which sampling-based planning algorithms have been inefficient at, or incapable of, solving. Specifically, there is a class of problems that can often be solved quickly, but can occasionally require an inordinate number of nodes to solve, making it difficult for current sampling-based reachability algorithms when there are uniform restrictions on time or computational resources such as memory for each query. Similarly, complex hybrid systems are difficult for current methods, because they are not adequately able to take advantage of the underlying structure of such problems. Finally, problems which have changing search spaces, whether because of other agents, unmodeled dynamics, or disturbances, are impossible to solve using traditional RRTs.

This research aims to improve the calculation time of the first two types of problems, and enable RRTs to solve the third. By adapting the RRT to contend with these issues more successfully, its reach as a planning and verification tool for hybrid systems can be extended to include more complex problems.

1.2 Thesis Contributions

This thesis demonstrates new techniques that can be applied to sampling-based reachability algorithms [5] to improve their ability to plan paths for and verify hybrid systems. Specifically, we concentrate on the Rapidly-exploring Random Tree (RRT), identify the types of reachability problems that give the traditional RRT trouble, and propose adaptations to the algorithm to improve its ability to find solutions in these cases.
The first problem for traditional RRTs that we discuss is the phenomenon of heavy-tailed behavior [10] mentioned above. We have found a variety of hybrid systems reachability problems have solution times that exhibit such behavior. We review this concept and methods to determine if an example demonstrates it. We also show that a regular RRT can often fail to find a path for a heavy-tailed example when restricted to a fixed number of expanded nodes. We then show that using several trees with fewer nodes (all rooted at the same initial point) yields a greater likelihood of success than does a single tree with many nodes. In addition, we show that restarting a tree that has been growing for too long can yield similar results. Finally, we demonstrate that these techniques are detrimental for problems which do not display a heavy tail.

Second, we address the problem of speeding the solution of complex hybrid systems reachability problems using a forest of trees to solve a problem instead of a single tree. Specifically, we find that by using several trees (each tasked with solving a part of the problem) and then uniting the partial solutions they produce, complete paths through the search space can be found more rapidly. We show that this approach, when compared with a regular RRT approach, generates fewer nodes and takes less computation time. We also show that bidirectional search can speed this up even further, but that it is important that the biasing not hinder the search.

The final problem addressed in this thesis is the problem of changing search spaces, because of other agents, unmodeled dynamics, or disturbances. This problem is addressed with receding horizon search [9]. Under this strategy, the RRT continually finds new paths through its changing search space, checking each path, and replanning if it is deemed to be no longer valid. We demonstrate the algorithm’s success in solving
problems with moving obstacles and ones in which there are unmodeled disturbances in some regions.

1.3 Thesis Outline

This thesis is organized into six chapters as follows.

Chapter 1 explains the goal of this thesis, the motivation for this goal, and provides the order of the presentation of results.

Chapter 2 introduces hybrid systems planning and verification problems, and the Rapidly-exploring Random Tree (RRT). This chapter also defines the metrics used to measure the performance of the RRT algorithms, and introduces the example hybrid systems that will be used throughout this thesis. Specifically, those examples are the Stair Climber, the Train Gate, the Hovercraft, the Pendulum, and the Helicopter.

Chapter 3 explores the phenomenon known as heavy-tailed behavior and the problems it introduces for traditional RRTs. This chapter introduces methods to determine if an example is heavy-tailed, and explores two techniques, namely using restarts and multiple trees, that can be used to help the RRT be more successful in verifying heavy-tailed examples (using a fixed number of expanded nodes).

Chapter 4 shows the benefits that can be gained by using a forest of trees in place of a single tree. This chapter explains a technique that can be used to break a hybrid reachability problem into stages, each of which can be searched with a single tree. This chapter also discusses combining these partial solutions into an overall solution to the problem. Finally, this chapter compares the performance of such forest approaches to that of singular trees, using the Stair Climber example.
Chapter 5 explores the use of **receding horizon search**, which can be used to replan paths in search spaces with changing conditions. As the conditions of the search space change and invalidate the path, a receding horizon search can plan a new path taking into account the newly encountered conditions. This chapter gives reasons why a receding horizon search might be necessary, and provides evidence of how a receding horizon search can succeed where a regular search would fail. These ideas are tested on the Hovercraft and Helicopter examples.

Chapter 6 reviews the conclusions drawn from this work, and also indicates areas which may benefit from further exploration and analysis.

### 1.4 Experimental Data

Experimental data reported in this thesis was generated on one of two computers. The first was a 1.4 GHz Pentium 4 with 256MB of RAM, running under Windows XP Service Pack 2. This computer was used to generate data for all chapters except Chapter 4. The second was a 2.4 GHz Pentium 4 with 512MB of RAM running Windows XP Service Pack 2. This computer was used to generate the data for Chapter 4. All algorithms were implemented using C++ in Visual Studio 2003. All visualizations of experiments were drawn using OpenGL.

Because all of the data, and therefore results, of these experiments were produced using randomized algorithms, the data reported here is statistical: the results from each experiment are statistics generated on a large number of trials of the same algorithm. The number of trials, as well as the way the statistics are generated, is reported along with the results.
Chapter 2

Background

In order to understand how Rapidly-exploring Random Trees (RRTs) can be used to solve planning and verification problems in hybrid systems, it is necessary to first understand what a hybrid system is and how it can be defined, what an RRT is and how it works, and how RRTs can be applied to hybrid system problems. This chapter addresses all of these issues, as well as discusses metrics that measure the performance of RRTs. It also introduces the example systems that are used throughout the thesis.

2.1 Hybrid Systems Background

A hybrid system is one combining continuous and discrete variables and dynamics. The discrete variables are known as modes or discrete states of the system. These variables take their values among a finite set of discrete values defined by the system. The continuous variables may take any value in their bounds, and move continuously according to dynamics that in general depend on the discrete state the system is in.

Hybrid systems are a convenient way to model many problems in a succinct, formal, concrete way, for examination via scientific means. They can be used to model both real and theoretical problems, and there have been several tools and methods created for examining them [1, 12]. All of this makes them a good way to examine problems and draw meaningful conclusions about them.

There are several models which can be used to describe hybrid systems. The model used in this thesis is that of hybrid automata. Hybrid automata will be discussed in
Section 2.1.1. For more information on hybrid systems and hybrid automata, the author suggests that the reader refer to [3, 21].

2.1.1 Hybrid Automata

In this thesis, all hybrid systems are modeled as hybrid automata. Hybrid automata are a formal way of specifying a hybrid system. A hybrid automaton is comprised of four components $H = (Q, X, A, E)$, each defined as follows.

- $Q$ is the finite set of discrete states of the system, known as the *discrete state space*. This is generally the set $\{1, 2, \ldots, N\}$, where $N > 0$ is an integer. The discrete state of the system modulates the continuous dynamics of the system.

- $X \subseteq \mathbb{R}^n$ is the space of continuous variables on which the system dynamics occur, also known as the *continuous state space*. The symbol $x$ is used to refer to any specific valuation of all of the variables in $X$.

- $A$ is the set of *activity functions*, which define how the continuous variables change while in each given discrete state. These are enumerated as $A = \{ a_i = f(X) \; \forall \; i \in Q \}$. Each $a_i$ is a set of ordinary differential equations or differential inclusions on the variables in $X$ governing the continuous dynamics of the system. A *differential inclusion* is defined as follows [3]:

  A differential inclusion is a dynamical equation that allows the derivative to belong to a set and is written as

  $$\dot{x}(t) \in F(x(t)),$$

  where $F(x(t))$ is a set of vectors in $\mathbb{R}^n$.

- $E$ is the set of *edges* governing the movement between the discrete states of the system. $E \subseteq Q \times Q \times G \times J$ where $G$ is the set of *guards* and $J$ is the set of *jump*
**functions** (also known as jump resets, edge resets, resets or impulses). The set $G$ satisfies $G \in 2^X$ where $2^S$ represents the power set of set $S$. The jump functions fit into the continuous state space of the system such that $J \subseteq X$. An individual edge $e$ can be referred to as $e = (q_1, q_2, g, j)$ where each of these variables is a specific element of the set of possible values for that variable.

An edge is followed when the continuous variables satisfy one of the guards. Each edge describes the process of transitioning from state $q_1$ to $q_2$ when the continuous state $x \in g$, and $j$ is the set of reassignments to the continuous variables that can take place upon following this edge.

An easy way to keep track of the difference between hybrid systems and hybrid automata is to think of every hybrid system as having an associated hybrid automaton which defines its dynamics. For the purposes of simplicity and clarity this thesis will always refer to hybrid systems, but it should be understood that the formal model of the system we use is that of a hybrid automaton.

### 2.1.2 Examples of Hybrid Systems

Hybrid systems are a convenient way to model many realistic situations. Hybrid systems have been used to model billiards [1, 24], bouncing balls [28], air traffic management [15, 29], automated vehicle highway systems [27], and temperature controllers/thermostats [1, 2, 13, 14, 15, 24].

An example of a hybrid system, given here for demonstration purposes, is a car racing forward in a straight line. This system might have three state variables: the gear the car is in, the position of the car, and the speed at which the car is moving. The gear is a discrete variable, which could be 1, 2, 3, 4, or 5. The position and speed are continuous
variables. The continuous dynamics of position and speed are governed by differential
equations or differential inclusions, which constrain the speed and acceleration,
respectively. The speed might range from 0 to 155 miles per hour, and the acceleration,
which is controlled by the gear, might range from 0 to 15 miles per hour per second. In
this case, the gear the car is in controls how the speed will change via the acceleration.
For example, when the car is in 1st gear, it can accelerate very quickly while toward the
lower end of the speed range, but it can only accelerate up to a predetermined speed,
which is much lower than the top end of the speed range. In 5th gear, the car can
accelerate well toward the top of the speed range, but can barely accelerate at all when it
is not in this range. A picture of the car example discussed above can be seen in Figure
2.1 where $v$ is the velocity of the car, and $a$ is the acceleration.

Figure 2.1: Car Example Hybrid Automaton.
In this thesis, the primary concern is verifying whether or not systems represented in this fashion can reach specific states. This is referred to as a problem of reachability. Formally stated, the reachability problem in reference to hybrid systems is: Given a hybrid system and an initial state, can it reach a specific goal state by following the hybrid dynamics of the system. If we can answer this question it can tell us a lot about the system. Answering this question is discussed in further detail in Section 2.2.3 of this chapter, under Adapting the RRT to Verification Problems.

2.2 Rapidly-Exploring Random Trees Background

The Rapidly-exploring Random Tree, or RRT, is an innovative method for searching space first introduced in [18]. Its most obvious use arises from the problem of searching large, high-dimensional, and continuous search spaces. The RRT is a sampling-based search algorithm, and avoids the problem of a continuous search by taking a sampling of the reachable space, instead of finding the reachable set exactly. The RRT tends to grow towards the larger unexplored regions, increasing its coverage of the search space very efficiently. It is probabilistically complete [18, 22]. An example of RRT growth in two dimensions can be seen in Figure 2.2.

Figure 2.2: RRT Growth in 2 Dimensions. From left to right: the tree with 100 nodes, 500 nodes, 1000 nodes, and 2500 nodes.
2.2.1 The RRT Algorithm

The basic structure of the RRT algorithm is given below as Algorithm 2.1. This algorithm builds an RRT in a loop of three steps. The tree begins with one node at the start point of the path it is searching for. This initial node is called the root. After this initialization of the tree, the loop begins, and the algorithm first picks a random sample state, $x_{rand}$, from the search space. Once an $x_{rand}$ is selected, the tree is searched to find the nearest neighbor, which is kept track of as $x_{near}$. After $x_{near}$ is found, the tree takes a step forward from $x_{near}$ toward $x_{rand}$, by creating a new node that is a step of length $\varepsilon$ towards $x_{rand}$ from $x_{near}$. The newly created node is referred to as $x_{new}$. This loop is repeated for $K$ iterations, constructing a tree with $K$ nodes. The input variables $start$ and $goal$ indicate the root of the tree and the goal location (if there is a specific goal location), respectively.

Algorithm 2.1 BUILDRRT($start$, $goal$)

1. Tree.INITIALIZE($start$)
2. FOR $k = 1$ to $K$
3. $x_{rand} =$ Tree.SAMPLESTATE()
4. $x_{near} =$ Tree.NEARESTNEIGHBOR($x_{rand}$)
5. Tree.NEWNODE($x_{rand}$, $x_{near}$)
6. RETURN Tree

The INITIALIZE($start$) function does whatever needs to be done to initialize the tree. Usually this consists of setting the tree size to one, and setting that one node that is in the tree equal to the start state of the system. Sometimes this function also includes
setting up things such as transitions between states, and movement rules for the system,
but these things are generally taken care of in the constructor of the tree.

The SAMPLESTATE() function returns a random state from the search space
consisting of a legal configuration of all of the values of the system. The implementation
of this function is discussed in greater detail in Section 2.2.3.

In general the NEARESTNEIGHBOR(xrand) function searches sequentially through
the nodes of the tree, checking the distance between each node and xrand, and keeping
track of the node closest to xrand. Once each node has been tested, the closest is returned
as xnear. Its implementation follows below. This search does not need to be sequential,
and does not need to consider all nodes in the tree. By using some additional overhead,
the search can exclude some nodes from the search to make finding the nearest neighbor,
or a near enough neighbor, faster [23]. In this thesis, the search is always sequential,
through all of the nodes of the tree.

Algorithm 2.2 NEARESTNEIGHBOR(xrand)

1. Node xnear = Tree.NODE(0)
2. FOR (i = 0; i < Tree.SIZE(); i++)
3. IF (distance(xrand, xnear) > distance(xrand, Tree.NODE(i))
4. xnear = Tree.NODE(i)
5. RETURN xnear

The NEWNODE(xrand, xnear) function generates a new node given xnear and xrand,
and adds it to the tree. In general, this function takes a step of length ε toward xrand from
xnear according to the movement dynamics of the problem.
Each of the steps in this algorithm will have specific implementations which may differ depending on a variety of factors, such as the problem being examined, the nature of the state space, and the modifications which have been made to the RRT algorithm (to be discussed in Section 2.2.3 in this chapter). For this reason exact definitions are omitted here, and will be discussed in each section pertaining to a specific RRT setup. All RRT algorithms in this thesis, however, follow this same basic structure.

2.2.2 Properties of the RRT

Several properties of the RRT make it an efficient method for answering the reachability question for hybrid systems. The properties discussed are the RRT’s ability to exploit Voronoi regions and the probabilistic completeness of the RRT.

Voronoi Regions

Due to their probabilistic nature, RRTs are very good at filling space because they tend to explore toward the larger unexplored regions. This property is easily understandable when examined from the view point of Voronoi regions, as explained in [22, 23].

A Voronoi region is defined as the set of points which are closer to a particular vertex of a graph than to all other vertices of the same graph. Much can be learned about the properties of RRTs by looking at pictures of Voronoi regions of (nodes of) a tree. The size of the largest Voronoi region shows us the area of worst coverage of the tree. In this region, lies the point in the search space that is the farthest from any node in the tree. In fact, all large regions indicate places where the tree has not explored well, as these are the places where nodes of the trees are the sparsest. Also, the range of sizes of the Voronoi regions is indicative of how evenly the space has been explored. If all of the
regions are more or less the same size, then this indicates that the space has been evenly explored. If, however, there are some regions which are much larger than the others this would indicate that the tree explored some regions less thoroughly than others.

Figure 2.3a shows two sets of points: one generated pseudo-randomly, the other from a lattice, along with each point set’s corresponding Voronoi region. Figure 2.3b shows an RRT in a two-dimensional space, as well as each node’s corresponding Voronoi region. It shows that as an RRT grows, the largest unexplored regions tend to lie toward the edges of the trees as the tree is expanding, as it takes longer for the tree to reach these regions, because they are farther away from the root of the tree. As the tree becomes better expanded, each of these larger regions gets smaller, and eventually all of the space is well explored.

Figure 2.3a: Voronoi Regions of Pseudo-Random Points (left) and Lattice Points (right). Reproduced from [20].

Figure 2.3b: Voronoi Regions of Each Node in an RRT at Three Stages of Growth. Reproduced from [6].
Because of the probabilistic nature of the RRT (i.e., that sample points $x_{rand}$ are chosen uniformly randomly) the largest current Voronoi region of the tree has the greatest likelihood of containing the next sample point. If this region’s point is chosen as $x_{near}$ and a new node is expanded from it, its corresponding region becomes smaller. This shows the biasing of the tree towards larger unexplored regions. The larger the unexplored region is in relation to the others, the more likely a new sample point will be chosen from this region, and consequently the more biased the tree will be to exploring toward this region. This is the reason the tree fills space so rapidly.

**Probabilistically Complete**

RRTs have been shown to be *probabilistically complete*, which means that as the number of nodes in the tree approaches infinity, the probability of finding a path (if one exists) approaches one [18].

### 2.2.3 Modifications to the RRT Algorithm

Several modifications can be made to the basic RRT algorithm to achieve different objectives. Those discussed here are terminating the search under certain conditions and changing the way sampling points are chosen. By making changes to the algorithm, it can be adapted to better suit specific purposes, whether to obtain superior exploration of the search space, avoid wasted calculation time, or simply to find a goal faster. RRTs become much more powerful when these types of modifications are made, because it allows them to be tailored to specific applications.

**Success Termination**

One example of a modification to an RRT is as follows. In problems where the goal is to find a path to a certain area or region, the looping element of the algorithm can
be modified so that it stops once this area is reached, because the goal of the problem has been accomplished. This type of modification is especially useful in verification problems, because in these types of problems the goal is often to find a path to a specific region. If a path is found, then the question is answered, and no further iterations are helpful. If no path is found, then the best that can be said for the system is that it is unlikely to be able to reach that region.

To accomplish this in the algorithmic sense the only change to be made is in the looping element which, instead of completing a predetermined number of iterations, runs until a node is created in this goal region. Due to practical limitations it is necessary to check in the while condition to ensure that the tree has not exceeded its maximum allotted space (a condition known as maximum expansion), but this actually makes the modification more similar to our previous example. If we assume that $K$ in Algorithm 2.1 is the maximum tree size, then checking for maximum expansion is something the algorithm was already doing. The modified algorithm appears as Algorithm 2.3.

**Algorithm 2.3** BUILDRRT(*start, goal*) with Success Termination

1. Tree.INITIALIZE(*start*)
2. **WHILE** ((Tree.Success(*goal*) is **FALSE**) AND ($k < K$))
3. \[x_{\text{rand}} = \text{Tree.SAMPLESTATE}()\]
4. \[x_{\text{near}} = \text{Tree.NEARESTNEIGHBOR}(x_{\text{rand}})\]
5. Tree.NEWNODE($x_{\text{rand}}, x_{\text{near}}$)
6. RETURN Tree
Biasing

Another modification can be made when the problem involves finding a path toward a goal location. While iterating through the loop, on occasion, instead of setting $x_{rand}$ equal to a random sampling point, it can be set equal to the goal point. This technique is referred to as goal biasing, and pushes the tree to grow toward the goal, while still allowing it to explore the space to a certain degree, dependent on the frequency of the biasing. With the tree set to bias more frequently (a higher biasing frequency), the tree will grow faster toward the goal, but will not explore the space as well. With the tree set to bias less frequently (a lower biasing frequency), the tree will do a better job of exploring the space, and grow less quickly toward the goal. Biasing can also be used to push the growth of the tree towards a specific region, if the bias step is implemented by setting $x_{rand}$ equal to random points from the goal region instead of a goal point. This technique is useful in verification problems, because by setting the goal equal to a known failure condition, the tree will grow toward that failure condition, attempting to reach it and find a counter-example.

To change the RRT algorithm (Algorithm 2.1) to use a biasing strategy, all that needs to be done is to change the implementation of the function in step 3; see Algorithm 2.4 below. If the number of iterations is a multiple of the biasing period, $x_{rand}$ is set equal to the goal. Otherwise, a random point is generated as before. Biasing in this fashion ensures that there will always be at least as many random points as goal points chosen as $x_{rand}$ (unless of course the biasing period is set to 1, in which case there will be no random points). If for some reason it was necessary to reverse this condition, and ensure that there are more goal points than random points, one would only need to switch the if clause with the else clause. This would be an unlikely situation, however, because under
most circumstances, biasing toward the goal more than every other iteration is unnecessary to ensure fast growth toward the goal. It should also be noted that because of the infrequency of random sampling, this type of biasing would impair or eliminate many useful properties of the RRT.

The basic RRT Algorithm has been adapted for non-linear dynamic systems and hybrid systems [4, 5]. Throughout this thesis, we use these RRT extensions to solve nonlinear and hybrid example problems, which are introduced in Section 2.4. Details of our RRT implementations are provided there.

Algorithm 2.4 SAMPLESTATE() with Biasing

1. Node $x_{rand}$
2. IF ( Tree.SIZE() % BiasingFactor)
3. $x_{rand} = $ Tree.RANDOMSTATE()
4. ELSE
5. $x_{rand} = $ Tree.GOALSTATE()
6. RETURN $x_{rand}$

Adapting the RRT to Verification Problems

RRTs can be used to find answers to verification questions for hybrid systems. An RRT traces paths through the search space according to the hybrid system’s dynamics. If it is possible according to the dynamics of the system, the tree may eventually reach one of these unsafe regions. If the RRT generates a path that reaches one of these unsafe regions, it has determined that the system can reach an unsafe state, and then we know that the system needs to be changed in some way so that it no longer allows movement into this unsafe region. If the system has been tested exhaustively and
has not been shown to be able to reach an unsafe state, this does not guarantee that the system is safe. It does, however, lend some level of assurance that the system is unlikely to ever reach an unsafe state. Since the RRT is a probabilistic approach, its failure to reach a specific state or area does not guarantee that the system cannot reach that state or area, because the RRT cannot, and does not check all possibilities. It merely takes a sampling of the problem space, and as a result might miss some areas. However, because the RRT is probabilistically complete it is likely that it will find a path to any given area in the search space. There are many example of previous research which used RRTs for verification problems. A small sampling is [5, 8, 25].

**RRTs with Obstacles**

RRTs can be used not only in empty continuous space but also if there are obstacles in the space. The RRT can avoid these obstacles; the only requirement is that the tree be able to determine whether or not a node or the edge leading to it intersect an obstacle. If this check is done at each step with the creation of $x_{\text{new}}$, then the tree can ensure that it never runs into an obstacle. In performing this check, it is also necessary to ensure that the edge from $x_{\text{near}}$ to $x_{\text{new}}$ does not pass through an obstacle. The RRT works according to Algorithm 2.3, except anytime a newly created node intersects an obstacle this problem must be corrected. In this case the node is thrown out and the iteration is repeated. Other options that could be used to solve this problem are discussed below as they relate to biasing. Algorithm 2.3 can be modified to handle this by inserting a check after the creation of the new node. The purpose of this check is to ensure that the newly created node is not reached by an edge that crosses or intersects an obstacle. If it is, then
the newly created node is thrown out, and the iteration is started over, without
incrementing the counter.

If the algorithm uses biasing, and an iteration is restarted, something must be done
to ensure that the same invalid node is not created again. This is necessary because if an
illegal node was created as a result of an $x_{rand}$ that was set equal to the goal, it will be
thrown out, and the tree size will not be incremented. As a result, when the iteration is
restarted, the algorithm will again determine that it is time to create a bias node, and set
$x_{rand}$ equal to the goal, it will pick the same node as $x_{near}$, and the same illegal node will
be created. If we do not force a random node, then the algorithm will be stuck in a never
ending loop, trying to create this same illegal node.

One strategy, the one used in this thesis, is to force the algorithm to take a random
node, instead of whatever comes next in the progression. Any time a new node intersects
an obstacle, the algorithm sets a flag that tells it that the next sample point should be
random. This solves the problem by encouraging another node to be picked as $x_{near}$, and
this other node likely will not grow into an obstacle. In the case that this other node does
grow into an obstacle, the algorithm repeats the process. Since the $x_{rand}$ selected is
random, it is likely that a different node will be selected as $x_{near}$ the next time.
Eventually, the algorithm will find a place to create a node that is legal, if this is possible.
The benefit of this approach is that it does not waste any nodes growing towards
obstacles once it finds them. Its drawback is that it has the potential to get stuck at
certain steps in its development because it tosses out nodes that intersect obstacles. The
probability of this increases proportionally with the number of obstacles in the search
space.
An alternative approach to dealing with this problem is to include the newly created edge but shorten it to the point that the edge and its terminal node no longer intersect the obstacle. This approach does not repeat iterations, and does not throw out nodes once they are created. However, because it still grows toward these obstacles, it creates nodes which are not as useful, because their mobility is restricted by being so close to the obstacle. This approach would work better than the previous approach as the number of obstacles increases, and as the obstacles become more closely packed together. This is because in a situation where there are many obstacles, or the obstacles are closely packed together, the previous approach would frequently throw out nodes, and repeat iterations, whereas this approach would create nodes that were closer to the obstacles, finding paths that were closer to the obstacles, which would be necessary with the obstacles being so plentiful, or close together. An example of this strategy appears below as Figure 2.4. The red, unlabeled circle is the goal, and the green labeled circles are nodes in the tree. The yellow rectangles are obstacles. Node E, the empty green circle with the red X through it, represents a node that would have run into an obstacle. The path is shortened, and the new node E’ is created next to the obstacle.
A third approach involves ignoring any $x_{near}$ which creates an invalid node, and instead finding the closest node that does not generate an invalid $x_{new}$, and extending from it. This approach encourages growth around the obstacle by picking nodes which are not as close to the obstacle and trying to grow them toward the goal. This sometimes allows other branches that will not intersect an obstacle to grow around the obstacle toward the goal. The drawback to this algorithm is that you must keep track of a list of the closest nodes to $x_{rand}$. This can be done without much further calculation if the list is built while searching for the nearest neighbor. The nearest neighbor function must calculate the distance between $x_{rand}$ and each individual node in the tree, so to keep an ordered list of these nodes, they only need to be inserted in an ordered list as the distance for each node is found.\footnote{A similar approach was used in discrete spaces for the Rapidly-exploring Random Leafy Tree [23].} The benefit of this algorithm is that it avoids the repeating of iterations of the first algorithm, and it avoids creating nodes with restricted mobility like the second one.
The primary drawback is that this method has extra overhead in memory involved with keeping the nodes in an ordered list.

An RRT algorithm using the first obstacle avoidance approach is given in Algorithm 2.5.

Algorithm 2.5 RRT with Biasing and Obstacles

1. Tree.INITIALIZE(start)
2. WHILE (!Tree.SUCCESS(goal) AND Tree.SIZE() < K)
3. \( x_{rand} = \text{Tree.SAMPLESTATE}() \)
4. \( x_{near} = \text{Tree.NEARESTNEIGHBOR}(x_{rand}) \)
5. Tree.NEWNODE\((x_{rand}, x_{near})\)
6. IF \((x_{new}.\text{CROSSESOBSTACLE}())\)
7. Tree.REMOVE_NODE\((x_{new})\)
8. Tree.ForceRandomNode = TRUE;
9. RETURN Tree

The implementation of the SAMPLESTATE() function is the same as that discussed in Section 2.2.3. The NEARESTNEIGHBOR\((x_{rand})\) function follows the implementation of Algorithm 2.2, checking the distance from each node in the tree to \( x_{rand} \). Once the function finds the node in the tree closest to \( x_{rand} \) it returns that node as \( x_{near} \).

The NEWNODE\((x_{rand}, x_{near})\) function creates a new node and adds it to the tree. It does this by computing the line between \( x_{near} \) and \( x_{rand} \), and then computing a new node at a length \( \varepsilon \) from \( x_{near} \) along this line towards \( x_{rand} \). If the distance between the points is less than \( \varepsilon \) then the new node will be created at the same location as \( x_{rand} \). The newly created node is called \( x_{new} \), and its parent is set equal to \( x_{near} \). The NEWNODE\((x_{rand}, x_{near})\)
function appears in Algorithm 2.6 below. For demonstration purposes, the distance metric used therein is a Euclidean distance in three dimensions.

Algorithm 2.6 **NEWNODE**($x_{rand}$,$x_{near}$)

1. double $x = x_{rand}.x - x_{near}.x$
2. double $y = x_{rand}.y - x_{near}.y$
3. double $z = x_{rand}.z - x_{near}.z$
4. double $h = \sqrt{x^2 + y^2 + z^2}$
5. **IF** ($h > \epsilon$)
6. $x_{new}.x = x_{near}.x + x/h \times \epsilon$
7. $x_{new}.y = x_{near}.y + y/h \times \epsilon$
8. $x_{new}.z = x_{near}.z + z/h \times \epsilon$
9. **ELSE**  // Snap to $x_{rand}$
10. $x_{new} = x_{rand}$
11. $x_{new}.parent = x_{near}$
12. Tree.ADDNODE($x_{new}$)
13. **RETURN** $x_{new}$

The **CROSSESOBSTACLE**() function in Algorithm 2.5 returns true if the function’s calling node or the edge leading to this node from its parent crosses one of the obstacles in the search space. For simplicity’s sake in this thesis, the algorithm only tests if the node is inside each obstacle. In practice, this could be an issue, but the problems in this thesis were structured such that any edge that would pass through an obstacle would have a node inside that obstacle at one edge.
This shows that the RRT is capable of avoiding obstacles in its space, increasing its usefulness to include examples in which there are some specific regions which the system is explicitly forbidden from entering for one reason or another. An RRT can avoid these regions (if such behavior is necessary) and grow around them. Although we’ve shown several strategies for doing this, this thesis uses the first approach (removing the invalid node and restarting the iteration with a new $x_{rand}$) exclusively.

2.2.4 RRT Terminology Specific to this Thesis

In the course of this thesis, RRTs will be discussed extensively. This will require some specific terminology to deal with them. The following terms will apply to RRTs in this thesis in its entirety.

**Problem** – The problem is a pair, consisting of a hybrid system and a question. The hybrid system defines the space in which the question will be answered. The question is generally a yes or no question, such as “Can the system reach a bad state?”

**Solution** – The solution is an answer to the question. Because an RRT cannot prove the lack of a path, the only solutions possible are those that describe a path from start to goal. If a path cannot be found, then there is no solution, there is only the knowledge that it is unlikely that a path exists.

**Example** – The example is the hybrid system in which the problem takes place. The example defines specific rules for the hybrid system, and often includes a reason for the existence of this system, i.e., a real world system that it models.

**Tree** – The tree is the Rapidly-exploring Random Tree that is attempting to find a solution to the problem. It grows in the problem space according to the dynamics of the
hybrid system. The tree defines a set of locations that it has been able to reach according to the dynamics of the problem. These locations are referred to as nodes.

Any tree which is currently being grown, but has yet to finish will be called an unfinished tree. Any tree whose growth has completed, without regard for the way it finished (either by reaching a goal or growing to maximum size) will be referred to as a finished tree.

2.3 Explanation of Metrics

In generating trees, it is useful to be able to measure how well they did at achieving the goals set out for them. To do this, several metrics are used to check the performance of the RRT. Their definitions are found below. For the purpose of these definitions, the distance to the tree means the distance to the closest node in the tree.

The simplest of these metrics is the size of the tree, measured in nodes. This is simply the number of nodes generated in the tree. This is a useful metric because it lets us know how much calculation was done to come up with a solution. Smaller tree sizes indicate that the tree was more quickly able to find the goal, which is better.

Another simple metric is the time required to generate the tree, measured from immediately before the tree is initialized, until immediately after the final node has been created. This metric is useful because it tells us how long in real time it took to find a solution.

Slightly more complicated is the path length, which is calculated by taking the final node in the path (the one that reached the goal) and counting back to the goal via the parent pointers. Because almost all of our experiments used a fixed step size, we measured path length using the number of nodes along the path from the start node of the
tree to the goal, if the goal is reached. This is a useful metric because it gives information about how well the question was answered. Shorter paths are more efficient.

In addition to these, there are also coverage metrics, as discussed below.

2.3.1 Dispersion

The first coverage metric, the dispersion, can be found in [20]. The dispersion is defined as the radius of the largest sphere that can exist in the search space without containing any node in the tree. To find the dispersion, a grid of points is laid over the coverage region, and then each point is checked, to see how far it is from the tree. All of these distances are compared, and whichever of these distances is greatest is defined as the dispersion. The primary benefit of this metric is that it gives us some information about the coverage of the tree, and is easy to compute. Although it only gives information about the area of worst coverage of the coverage space, this is frequently enough, because sometimes it is only important what the lower bound for performance is.

2.3.2 Kim Metric

The second coverage metric, which will be referred to as the Kim metric, was first introduced in [8]. In this coverage metric, a regular grid of points, with spacing \( \delta \), is laid over the search space. For each point in the grid, the min is taken between \( \delta \) and the distance from the grid point to the tree (measured as the distance from the closest point in the tree). These distances are summed and then averaged, and the result is the coverage metric.

By taking the min between \( \delta \) and the distance to the tree, the Kim metric implicitly states within what distance it expects to find the closest node in the tree. This establishes a threshold of coverage for the tree. This threshold of coverage can be as
coarse or as fine grained as desired, to either weaken or strengthen the metric. With the grid points farther apart, the metric is calculated more rapidly, but the coverage ensured by the metric is not as good. With grid points being farther apart there are larger regions between the points which may be unexplored and are not tested. With grid points closer together, the calculation takes longer, as there are more points in the grid, but the coverage assured by the metric is more accurate, because the regions between points are smaller. The algorithm for the Kim metric appears below as Algorithm 2.7.

**Algorithm 2.7 Kim Metric**

1. `sum = 0`

2. `FOREACH (GridPoint in GridPoints)`

3. `sum += Min(δ, DISTANCE_FROM_TREE(GridPoint))`

4. `RETURN sum / NumberOfGridPoints`

### 2.3.3 Reasons for Two Metrics

Although both of these metrics measure coverage, they measure it in fundamentally different ways. The dispersion calculates the radius of the largest circle which is devoid of nodes in the tree. This tells us something about the worst area of coverage of the tree. This metric is easy to calculate, widely understood, and acceptable for measuring coverage. The Kim metric instead tells us something about the average distance between the nodes of the tree. This metric is not as widely used, but it has an additional benefit.

If in the problem space there is a large region into which the tree cannot grow, the dispersion metric is likely to always find that the largest circle will be in this region. This tells us nothing about the coverage of the area outside of this region. The Kim metric, by
making a statement about the average distance between nodes, allows us to see differences in coverage outside of this region even if we are not aware of its existence. This difference can be seen in Figure 2.5.

![Figure 2.5: The Problem with Dispersion. The picture on the right clearly covers the space better than the picture on the left; however, both would have the same value for the dispersion metric. Reproduced from [8].]

### 2.3.4 Average Dispersion

From these two metrics, two more were developed in the course of this thesis. From the dispersion, comes the *average dispersion*. This is a very similar measure to its parent, as it is calculated essentially the same way, except that instead of returning the greatest distance from a grid point to the tree, it returns the average of all of the distances from every grid point to the tree. The average dispersion is more similar to the Kim metric than is dispersion, because it, like the Kim metric, averages the distances it computes. The difference here is that there is no threshold of coverage against which the tree is competing. This metric gives us information about the coverage of the tree over the entire space, not just about the biggest hole in its coverage, and it gives it in a familiar form, measuring the average distance to the tree from (approximately) anywhere in the search space. The algorithm for average dispersion appears below as Algorithm 2.8.
Algorithm 2.8 Average Dispersion

1. sum = 0
2. FOREACH (GridPoint in GridPoints)
3. sum += DISTANCE_FROM_TREE(GridPoint)
4. RETURN sum / NumberofGridPoints

2.3.5 Normalized Kim Metric

From the Kim metric, comes the normalized Kim metric. Again, this metric is very similar to its parent in that it is calculated essentially the same way. The difference is that once calculated, the metric is divided by the distance $\delta$ between the grid points, normalizing it to be between 0 and 1. This aides in the understanding of the metric, and eliminates the reader from having to know the grid size. For example, before normalization, a Kim metric of 9.95 could have many different meanings. If the grid size was 25, then this would indicate good coverage of the space. However, if the grid size were 10, then the meaning of this metric is much different. When normalized, it is readily apparent how well the tree performed, because values mean the same thing no matter the grid size: the closer to 0, the better the coverage; the closer to 1, the worse the coverage. This moves the grid size to a more appropriate role, as additional, rather than necessary, information. Knowing the grid size lets the reader know how confident he should be in the coverage guaranteed by the number reported.

2.4 Summary of Examples

This section summarizes all of the hybrid systems examples that were explored in the course of this thesis. In it, the reader will find each example discussed in detail, as well as the implementation of the RRT Algorithm specific to the example.
2.4.1 Stair Climber

The most obvious use of the RRT comes in continuous spaces, because they make intuitive sense, and are easy to picture, but as stated in Section 2.2 they can also be used to great effect to explore hybrid spaces. The first example used in this thesis for exploration by RRTs is a stair climbing example [4, 5, 6, 21]. In this example the search space is a building with four floors. The RRT starts on the first floor, and is allowed to explore freely. Once the RRT reaches a transition region (stairway) on the first floor, it jumps to the second floor where it is also allowed to explore freely. The same behavior occurs on each subsequent floor until the tree reaches a goal on the fourth floor.

In its formal RRT algorithm, this problem exactly follows Algorithm 2.3, and the implementation of the SAMPLSTATE() function is the same as that in Algorithm 2.4. The NEARESTNEIGHBOR(x_rand) function follows the implementation of Algorithm 2.2 searching through the nodes in the tree, and returning the one that is closest to x_rand. The closest node is defined as the node on the closest floor to x_rand that is the closest to x_rand. This ensures that a new node would always be created so that the path from that node to the random node would be as short as possible.

This is a two step approach. First NEARESTNEIGHBOR(x_rand) finds the floor closest to x_rand that has nodes. For example, if x_rand were on the third floor, and there were nodes on the first and second floors, NEARESTNEIGHBOR(x_rand) would only consider nodes on the second floor. Once the floor has been found, NEARESTNEIGHBOR(x_rand) searches the nodes on that floor to find x_near. The implementation of each of the other sub-functions is as follows.

For comparison purposes, distances between points in the space were calculated by taking the square of the Euclidean distance between two points in the x-y space, and
adding to that 100000 times the difference in floors. The square of the Euclidean distance was used to speed up calculation, since all distances were non-negative real numbers, and for non-negative real numbers \( d_1 < d_2 \iff d_1^2 < d_2^2 \). The floor weighting was done as a shortcut to ensure that the distance in floors (z distance) was always more important than the distance on a particular floor (x-y distance). Adding 100000 times the difference in floors prohibited any node from being selected as \( x_{\text{near}} \) unless it was on the closest floor to \( x_{\text{rand}} \). For example, if we had 3 nodes of the form \((x, y, \text{floor})\), with \( x_{\text{rand}} = (5, 6, 3) \), \( A = (6, 5, 1) \), and \( B = (187, -179, 3) \), then \( B \) would be closer to \( x_{\text{rand}} \), because it is on the same floor, while \( A \) would be considered farther away, despite the fact that the values would indicate that it was closer in the continuous \((x, y)\) space. Table 2.1 illustrates this point.

<table>
<thead>
<tr>
<th>Values</th>
<th>Euclidean Distance = ( \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2} )</th>
<th>Comparison Distance = ( (x_1-x_2)^2 + (y_1-y_2)^2 + 100000(z_1-z_2)^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{\text{rand}} ) 5 6 3</td>
<td>0 2.45 260</td>
<td>0 400002 67349</td>
</tr>
<tr>
<td>A 6 5 1</td>
<td>2.45 0 258</td>
<td>400002 0 466617</td>
</tr>
<tr>
<td>B 187 -179 3</td>
<td>260 258 0</td>
<td>67349 466617 0</td>
</tr>
</tbody>
</table>

Table 2.1: Explanation of Distance Metric.

In this example, the RANDOMSTATE() function generates a random point in the search space of the problem by first selecting a random floor from \([1, 4]\) in the building, and then selecting a random \((x, y)\) location on the floor, where \( x \) and \( y \) are in the range of \([-200, 200]\).
The implementation of the `NEWNODE(x_{rand}, x_{near})` function for this example is similar to Algorithm 2.6 in Section 2.2, except that changes have been made to account for the difference between the variables $z$ in the previous example and `floor` in this one. There are two specific changes. The first is in the method of computing $x_{new}$, and the second is for transitioning between floors.

The first is that in this example the floor is not used to compute the line between $x_{near}$ and $x_{new}$, where $z$ was used to compute this line in the previous example. The floor is not used to compute this line because in this example movement is restricted to each individual floor. The line is computed in $x$-$y$ space, and the `floor` is only used when nodes make transitions between floors. The edges that change floors occur in vertical lines, so there is little computation to be done: just the value for `floor` is changed.

The second change is that there must be a method for transitioning between separate floors of the problem space. In this implementation, the algorithm checks each new node to see if it lies inside the upward transition region for that floor (which is a square between $[180, 190]$ for $x$ and $y$ on even numbered floors and $[-180, -190]$ for $x$ and $y$ on odd numbered floors). If it does not, then nothing happens. If it does, then in addition to the original new node, a second new node is created with the first new node as the parent. The second node is created at the same $x$-$y$ coordinates as the first node but on the next highest floor. The algorithm appears below as Algorithm 2.9.
Algorithm 2.9 **NEW NODE** \( (x_{\text{rand}}, x_{\text{near}}) \) for Stair Climber

1. \( \text{double } x = x_{\text{rand}} x - x_{\text{near}} x \)
2. \( \text{double } y = x_{\text{rand}} y - x_{\text{near}} y \)
3. \( \text{double } h = \text{SQRT}(x^2 + y^2) \)
4. \( \text{IF } (h > \epsilon) \)
5. \( x_{\text{new}} x = x_{\text{near}} x + x/h \times \epsilon \)
6. \( x_{\text{new}} y = x_{\text{near}} y + y/h \times \epsilon \)
7. \( x_{\text{new}} \text{floor} = x_{\text{near}} \text{floor} \)
8. \( \text{ELSE } // \text{ Snap to } x_{\text{rand}} \)
9. \( x_{\text{new}} = x_{\text{rand}} \)
10. \( \text{IF } (x_{\text{new}} \text{.INTRANSITIONREGION}()) \)
11. \( x_{\text{new}} \text{parent} = x_{\text{near}} \)
12. \( \text{Tree.ADDNODE}(x_{\text{new}}) \)
13. \( x_{\text{new2}} = x_{\text{new}} \)
14. \( x_{\text{new2}} \text{parent} = x_{\text{new}} \)
15. \( x_{\text{new2}} \text{floor} ++ \)
16. \( \text{Tree.ADDNODE}(x_{\text{new2}}) \)
17. \( \text{RETURN } x_{\text{new}} \)

Visual examples of this problem can be seen in Figures 2.6 and 2.7, which show two sample runs of the Stair Climber problem. Figure 2.6 is an orthogonal projection of each of the four floors, while Figure 2.7 is a picture of the three-dimensional representation of the entire tree on all four floors. The vertical lines in Figure 2.7 show transitions among floors from the transition regions. The vertical line leaving the fourth
floor indicates that a node has reached the goal. This example, in general, can be explored with both forward trees which have been explained in this chapter, and backward trees, which will be discussed in Chapter 4. In all pictures of this example, including those in other chapters, green trees indicate forward growth (trees trying to grow up the building from the base floor) while red trees indicate backward growth (trees that are growing down the building from the top floor). Figures 2.6 and 2.7 show only forward trees.

Figure 2.6: Stair Climber Example RRT. Floors 1-4 (left to right), 15000 nodes.

Figure 2.7: Another Stair Climber Example RRT. Floors 1-4 (bottom to top), 15000 nodes.
This example shows that the RRT can be adapted to explore a hybrid system with specific switching conditions. This is useful information because this shows that RRTs can explore a hybrid system as successfully as a continuous one if given enough opportunity to expand. This shows that using RRTs for hybrid systems verification is an interesting and worthwhile idea to explore, because an RRT can adequately explore a hybrid system.

2.4.2 Train Gate

The Train Gate problem is a benchmark hybrid systems verification problem taken from [12]. This problem was also studied with RRTs in [25]. It deals with a train approaching a crossing with a road. This problem has three entities, the train, the gate, and the controller, each of which have their own separate automaton. The train is the reason for the problem, i.e., because there is a train crossing a road, and the potential exists for the train to crash into cars. Something must be done so that cars and trains do not crash into each other. The gate is what is done to correct the problem, i.e., the gate is there to prevent cars and trains from crashing into each other, by restricting the mobility of the cars that want to cross the track. The controller is what allows the train and the gate to work together allowing safe passage of both trains and cars through this crossing. Specifically, the controller tracks the train, and communicates to the gate what needs to be done to prevent the cars and trains from crashing.

The first of these three components is the train. The train is at position \( x \), measured as the distance between the front of the train and the crossing, which is at \( x \) value 0. The train’s rate of approach can vary in the range specified by the train’s discrete state. When the train is far from the gate, its speed varies in the range \([-50, -40]\).
As the train gets closer and is approaching the gate, its speed varies in the range 
[-50, -30]. Once it passes the gate, its speed varies in the range [30, 50]. The hybrid
automaton for the train appears below as Figure 2.8.

The second component is a gate across the road at the crossing, blocking cars
from crossing the train track while the train is passing through. The arm of the gate is at
an angle of \( g \) degrees with the surface of the road. When \( g = 0 \), the gate is fully closed,
and when \( g = 90 \), the gate is fully open. The gate, when moving, moves at a constant
speed of 9 degrees/second, either positively or negatively. The hybrid automaton for the
gate appears below as Figure 2.9.
The third component is a controller which communicates to the gate the train’s position. Upon detecting that the train is within 1000 [distance units] of the gate, the train is said to be approaching the gate, and the controller waits a constant amount of time, $\alpha$, before sending a “lower” signal to the gate. The controller has a time variable $t$. This variable keeps track of how long the controller has been waiting since detecting the train’s approach to send the lower signal to the gate. The goal is to find for which $\alpha$ values the system will work correctly. The hybrid automaton for the controller appears below as Figure 2.10.

![Figure 2.9: Gate Automaton. Reproduced from [12].](image)
The gate starts in the open position at a 90 degree angle with the surface of the road allowing cars to pass through. Upon receiving the “lower” signal from the controller, the gate begins to close at a constant rate of 9 degrees/second, until it is completely closed. The gate remains closed while the train passes through the gate, at which point the controller again waits a constant amount of time, $\alpha$, before sending a “raise” signal to the gate. Upon receiving the raise signal, the gate begins to open at a constant rate of 9 degrees/second, until it is completely open. A visual example of this behavior can be seen in Figure 2.11.

The red rectangle on the left of the picture shows the bad region, where the gate is not fully closed while the train is passing through. The other two vertical lines in the picture, which are blue, show the closest and farthest distance the train can be from the gate when the gate starts to close. The horizontal and angled line shows the continuous states of both the train, and the gate. The horizontal line at the top right of the picture
shows the train approaching the gate, and the gate waiting for the lower signal. As the line descends at an angle toward the bottom left, the train is approaching the gate while the gate is closing. When the line flattens out, the gate has closed completely. This flat line passes under the bad region, it does not intersect it. In the cases where the tree grows into the bad region, where the train has arrived at the gate ($x = 0$) and the gate has not fully closed ($g > 0$), the simulation is stopped, and no yellow line is generated. The system is said to work when the gate always closes completely before the train reaches it, and remains closed until the train has completely passed through the gate.

The colors of the line representing the continuous state variables of the train and the gate indicate the discrete state of the train. Violet is the “far” state, cyan is the “near” state, and yellow is the “past” state. There are many cyan branches because in this region the gate is closing. The train can move at any speed within its range of [-50, -30], which, in combination with the moving gate, creates a movement envelope that the tree can grow into. This does not happen in other parts of the tree because only the train is moving, so the tree progresses in a straight line.

Figure 2.11: Train Gate Example RRT. X direction = train $x$ value (-105 to 1050 shown); Y direction = gate $g$ value (0 to 90); violet = train state 0; cyan = train state 1; yellow = train state 2; red rectangle = bad region”; blue lines = close signal region; grey lines = coverage metric calculation points; white line = line from farthest coverage metric point to its closest node.
Testing

The algorithm for this example is exactly the same as Algorithm 2.3. The `SAMPLESTATE()` function is implemented using Algorithm 2.4. The `RANDOMSTATE()` function generates sample points $x_{\text{rand}}$, by independently generating a random state for each of the train, gate, and controller from their respective hybrid automata. Once each of these random states is generated, a random legal state is generated for the continuous variable of each automaton according to the restrictions of the state. It is important to note that these random states for each automaton need only be legal for the individual automaton, not the system at large. For example, a sample point might have the train in the “past” state, with $x = 2$, the gate in the “lowering” state with $g = 75$ and the controller in the “idle” state with $t = 0$. It is legal for the train to be passing through the gate, it is legal for the gate to be lowering, and it is legal for the controller to be idle, but it is not legal in the system at large for each of these components to be in these particular states at the same time. Once this random point in the problem space is constructed the `NEARESTNEIGHBOR(x_{\text{rand}})` function, following the implementation of Algorithm 2.2, finds the closest node in the tree to it by calculating the distance between each node in the tree, and $x_{\text{rand}}$, and storing the result as $x_{\text{near}}$.

The distance metric for determining the nearest node in the tree to the sample point is based on a Euclidian distance on the continuous variables of the system ($x$ from the train, $g$ from the gate, and $t$ from the controller) except that the distance in each dimension is multiplied by the average rate of change of that dimension to account for the importance of each continuous variable. For example, to determine the train component for the distance, the difference between the $x$ values of nodes $x_{\text{rand}}$ and $x_{\text{near}}$ is multiplied by 40. This results in the following distance metric, where a symbol preceded by $\Delta$
indicates the absolute difference in that value between the two points \( x_{\text{rand}} \) and \( x_{\text{near}} \). The square root is eliminated from this metric to speed calculation:

\[
D = 40 \times \Delta x + 9 \times \Delta g + \Delta t.
\]

If the gates of the two points were traveling in opposite directions (i.e., one is opening and the other is closing), then 9 distance units are added to the distance metric. This is done to account for the fact that a gate traveling in the opposite direction has to stop, and change direction which is assumed to take approximately 1 time unit.

In the creation of new nodes, it is important to understand the concept of the \textit{movement envelope} of a node [21, 25]. Because this is a rectangular hybrid automaton, from each node, there is a whole region into which the tree can move forward from this node, defined by the maximum and minimum rates of change of the continuous variables. In this example, the only variable which has a range instead of a single value for its rate of change is the position of the train. Thus, when the maximum or minimum of an envelope in this example is addressed, it refers to the maximum or minimum of the velocity of the train.

When each new node is created, each of the continuous variables is checked to see if any have crossed a transition region. If any of them has, its value is set equal to the transition value, and the state change is noted in the corresponding automaton.

Three approaches were tested in constructing the bias nodes in the \textsc{SampleState()} function. The first sampling approach, called \textit{bad-point biasing}, uses a particularly bad point as the bias point. This approach was tested because in this example, it is easy to imagine the worst possible situation, with the train being at the gate, and the gate still being completely open.
The second approach, called *bad-region biasing*, instead of occasionally picking a particular bad point, picks a point randomly from somewhere inside the bad region, defined as anywhere where the gate is not completely closed, and the train is passing through the gate. These points were generated using the same method as in the `RANDOMSTATE()` function, but by restricting this function’s search area to the bad region. This approach was tested because the problem gives a definition for the “bad region” that the system should avoid, that being the region in which the gate is not completely closed while the train is passing through the gate.

The third approach, called *unbiased sampling*, did not use a biasing approach. This would be reflected in the `SAMPLESTATE()` function always returning a `RANDOMSTATE()`. This approach was a control approach, to show that biasing approaches improve the speed of the algorithm.

### 2.4.3 Hovercraft

This example is taken from [8]. It involves a hovercraft moving around in an area that is divided into two distinct sections. The first, in which the craft starts, is windy, and the wind in this area will have an effect on the motion of the craft. The second, where the goal lies, is inside of a building; inside this area, the wind cannot affect the motion of the craft. The hovercraft has six continuous state variables, namely $x_1$, its position on the horizontal axis, $x_2$, its position on the vertical axis, and $\theta$, the angle the craft is facing from the positive horizontal axis, plus their respective velocities $v_{x_1}$, $v_{x_2}$, and $\omega$. The latter have dynamics given by their accelerations, defined by Newton’s laws of motion:

$$ma_{x_1} = (f_1 + f_2) \cos(\theta) + f_{x_{air}}(x, v_{air}(x))$$

$$ma_{x_2} = (f_1 + f_2) \sin(\theta) + f_{x_{air}}(x, v_{air}(x))$$
where $f_{x_{air}}(x, \nu_{air}(x))$ is the force on the craft due to wind in the $x_1$ direction,

$f_{x_{air}}(x, \nu_{air}(x))$ is the force on the craft due to wind in the $x_2$ direction, and $\tau_{air}(x, \nu_{air}(x))$ is the torque on the craft due to wind in the $\theta$ direction. The vector $\nu_{air}(x)$ is the velocity vector for the wind whose components $x$ and $y$ equal $[\alpha x_1, \beta x_2]$ when $\sqrt{(x_1)^2 + (x_2)^2} \leq 100$ (when the craft is outside the building) and $[0, 0]$ when $\sqrt{(x_1)^2 + (x_2)^2} > 100$ (when the craft is inside the building). The forces $f_1$ and $f_2$ are forward actuating forces for the hovercraft, each of which is applied at a distance $l$ in opposite directions from the center of mass of the craft, measured perpendicular to the craft’s forward facing direction. A figure of the hovercraft, and the points where forces $f_1$ and $f_2$ are applied in relation to the hovercraft’s center of mass, appears below as Figure 2.12.

![Figure 2.12: The Hovercraft. Reproduced from [8].](image)

These forces are controlled by the driver of the craft, and can vary freely in the range [-10, 10]. $m$ is the mass of the craft and $J$ is the craft’s rotational inertia in the $\theta$ direction. It is assumed that the craft stays at a constant distance above the ground, eliminating the $z$ direction for simplicity. The area the craft explores is defined as
[0, 200] in both the $x_1$ and $x_2$ directions (drawn as horizontal and vertical axes in figures). The hovercraft starts at $(x_1, x_2) = (0, 0)$, and is searching for a goal region defined as $x_1 \in [190, 200], x_2 \in [0, 10]$. A visualization can be seen in Figure 2.13, which can be interpreted as follows. The tree, showing the movement of the hovercraft, is drawn in white. The red square at the bottom right of the picture is the goal region, which is centered around the goal point, the red dot in the center of the box, at $(195, 5)$. The blue quarter-circle shows the region in which the wind has an effect on the craft’s motion. Inside the circle, the wind affects the craft’s motion, and outside of the circle it does not. The green ‘L’ at the bottom left corner of the screen indicates the origin. The vertical bar points along the positive $x_2$-axis, and the horizontal bar points along the positive $x_1$-axis. Each picture is surrounded with a blue outline, which has nothing to do with the problem.

Figure 2.13: Hovercraft Example RRT.
**Testing**

The RRT is grown according to Algorithm 2.3, with the biasing version of the `SAMPLESTATE()` function found in Algorithm 2.4. `GOALSTATE()` returns a node with values \((x_1, x_2)\) equal to \((195, 5)\), the centroid of the goal region. All other values are set to zero. The `RANDOMSTATE()` function returns an \(x_{rand}\) in 6-dimensional space. \(x_{rand}\) has random values for \(x_1\) and \(x_2\) in the range \([0, 200]\), and a random value for \(\theta\) in the range \([0, 2\pi]\). The random values returned for the velocities \(v_1, v_2,\) and \(\omega\) were selected from the ranges \([-73, 73]\) for \(v_1\) and \(v_2\), and \([0, 2\pi]\) for \(\omega\). In this example, there were no explicit upper bounds on the velocities of the craft, but random sampling points were never chosen to have velocities greater than 73 in any direction because that was the maximum velocity observed in testing.

The `NEARESTNEIGHBOR(x_{rand})` function follows the implementation of Algorithm 2.2 searching the tree to find the closest node in the tree to \(x_{rand}\), and keeping track of its index as \(x_{near}\). The distance metric used for this search is the sum of the squares of the distances between nodes in the \(x_1\) and \(x_2\) directions. The other four coordinates are not matched, because the goal of the problem is simply to arrive in the goal region. There is no condition for the direction the hovercraft should be facing or the speed it should be going when it arrives in the goal region, so these conditions are not used in calculating the distance metric.

The `NEWWNODE(x_{rand}, x_{near})` function generates new nodes as follows. To move the craft forward from \(x_{near}\) toward \(x_{rand}\), ten random configurations of forward actuating forces are tested. For each example, the first configuration is tried and its forces used to

---

\(^2\) Although velocities would normally be chosen to be both positive and negative, we chose them to only be positive in the range \([0, 2\pi]\) in this example. This does not prohibit rotation in both directions, as the force inputs to the craft \(f_1\) and \(f_2\) can cause rotation, and are allowed to vary freely in the range \([-10, 10]\).
generate an $x_{\text{new}}$. The distance in two dimensions from $x_{\text{new}}$ to $x_{\text{rand}}$ using the distance metric described above is saved to be compared against later configurations. After this, nine other random configurations generated for that specific iteration are tested to see if any of these configurations move the craft closer to $x_{\text{rand}}$. If any configuration moves the craft closer to $x_{\text{rand}}$, then that configuration and the resulting distance from $x_{\text{rand}}$ are saved and tested against the remaining random configurations. At the end, the configuration that moves the craft closest to $x_{\text{rand}}$ is chosen, and used to generate the child, $x_{\text{new}}$. This is done by progressing the node forward in time by updating its positions, velocities, and accelerations. The algorithm generates this step using an Euler method, taking five short steps each of length $\varepsilon/5$ that add up to a length of $\varepsilon$. This approach is used in favor of taking one long step because it does a better job of approximating the changing dynamics as the hovercraft moves. Once the closest $x_{\text{new}}$ is found, it is added to the tree. This is repeated until either the tree reaches the goal, or expands to the maximum tree size.

### 2.4.4 Pendulum

This example was found in [4, 5, 6]. It involves a pendulum, which has a motor at its hinge capable of applying torques on the pendulum. This problem has continuous variables $\theta$, the angular position; $\dot{\theta}$, the angular velocity; and the discrete input variable for the torque. The torque can take any of the values in the set of torques $\tau$, which in this case was $\{-1, 0, 1\}$. The pendulum starts hanging at rest straight down from its hinge, at 0 radians of rotation. The goal of the problem is to get the pendulum to stop at $\pi$ radians of rotation, directly above its point of rotation. This is accomplished by using the motor to torque the pendulum until it gets to the top of its circle of rotation. In all but the simplest cases, the pendulum will have to be swung back and forth to build momentum,
because the motor will not be strong enough to turn it directly to the top. The pendulum has a length $l$, and a mass $m$, which is concentrated completely at the end of the pendulum.

The RRT algorithm for this problem follows that of Algorithm 2.3. The \texttt{NEARESTNEIGHBOR(x\textsubscript{rand})} function follows that of Algorithm 2.2. The \texttt{SAMPLESTATE()} function follows the implementation of Algorithm 2.4. The \texttt{RANDOMSTATE()} function generates random nodes $x\textsubscript{rand}$ are created by generating random values for the continuous variables of the pendulum, from the ranges $[-\pi, \pi]$ for the angle, and $[-10, 10]$ for the angular velocity. The \texttt{NEWNODE(x\textsubscript{rand}, x\textsubscript{near})} function generates new nodes by testing each of the possible torques, and selecting the one which will get $x\textsubscript{near}$ closest to $x\textsubscript{rand}$. The torques are tested by using each torque to progress the continuous variables $\theta$ and $\dot{\theta}$. The angle $\theta$ is updated as a function of the angular velocity $\dot{\theta}$ as $\theta + = \dot{\theta} \times \varepsilon$. The angular velocity $\dot{\theta}$ is updated as a function of the angular acceleration $\ddot{\theta}$ as $\dot{\theta} + = \ddot{\theta} \times \varepsilon$. The angular acceleration is updated according to the influences of gravity and the torque $\tau$ from the motor. The function for the angular acceleration is

$$\ddot{\theta} = (-3 \times g)/(2 \times l) \times \sin(\theta) - (3 \times \tau)/(m \times l^2),$$

where $l$ and $m$ are the length and mass measurements of the pendulum, $g$ is the gravitational constant 9.8 and $\varepsilon$ is the step size.

The distance function returns the square of the Euclidean distance between two nodes in $\theta \times \dot{\theta}$ space, the same as the search space of the problem.

A visual representation of the problem can be seen in Figure 2.14. The tree is shown in white, while the lines of the green ‘L’ are coordinate axes, and the small red box on the right is the goal region. The $x$-axis plots the angular position of the mass at
the bottom of the pendulum, measured counterclockwise from the position of the pendulum at rest, and the y-axis plots its angular velocity in the counterclockwise direction. It can be seen in the pictures that the mass is swinging back and forth by the spiraling path the tree takes to get from the initial point at the origin to the goal inside the red box at the far right.

![Figure 2.14: Pendulum Example RRT.](image)

### 2.4.5 Helicopter

The Helicopter example was created in [26]. This is a good visual example of RRTs at work. It involves a helicopter, flying in a corridor of space where

\[ x \in [-400,400], y \in [-200,200], \text{ and } z \in [-150,150] \]

(bordered on each side by buildings) and trying to avoid obstacles in the form of projectiles that are flying at it. The helicopter starts at an initial point (-400, 0, 0) in the space, and attempts to fly through this space to a goal at (400, 0, 0). As the helicopter is flying, projectiles are fired at it from the goal,
directly toward the helicopter’s current location (see Figure 2.20). One projectile is fired at each time step of the algorithm, until the helicopter reaches some threshold of proximity to the goal (250 distance units from the goal plane).

**Flight Dynamics**

For this demonstration, the flight dynamics of the helicopter needed to appear realistic, but did not in fact have to be realistic. To accomplish this goal, a simplified set of flight dynamics was implemented. The helicopter was allowed to move in the three dimensions, but was not allowed to move freely. The helicopter would, in each stage of its development, choose one of five possible movements for the next step. Specifically, the helicopter could elect to fly straight, turn right or left, or make a move upward or downward. A visual sample of each of these movements appears in Figures 2.15-2.19.

At each stage of development of the tree, the helicopter chose the one of these five options that got it closest to $x_{rand}$. Additionally, each motion has associated yaw, pitch, and roll values which are interpolated through each motion. The purpose of these motions was to make the visual simulation of the helicopter’s flight look as real as possible without spending too much time on it.

The interpolation of the values depends on the current step, and on the next step. For example, if a right turn was the current action, the value of the roll would change as the helicopter went through the step. If the following step is a straight segment, the roll will change linearly from 0 at the beginning of the turn to 45 in the middle of the turn, and back to 0 at the end of the turn, so as to be ready for the straight segment. If, however, the following step was another right turn, then the roll would change linearly from zero at the beginning of the turn, to its maximum value of 45 degrees at the end.
Since the next step is also a right turn, the roll value finishes at the maximum in order to be ready for the next turn. Essentially, the turn is extended to two segments. These values have little effect on the RRT portion of the experiment.

**Projectiles**

The obstacles in this example are spherical projectiles, with a radius of 10 distance units. The helicopter is 17 distance units long and 14 distance units wide. Projectiles are fired at a rate of one per second, and move at a speed of 2 distance units per second (twice the speed of the helicopter). They are fired from the goal toward the helicopter at its current location, and move in a straight line. The reason that projectiles are fired from the goal is to increase the likelihood that their path intersects the path of the helicopter. Since the helicopter is going toward the goal from its current location, and the projectiles are going toward the helicopter’s current location from the goal, the probability is high that their paths will intersect unless the helicopter is able to route itself around the projectiles. Because of this constraint, the projectiles often necessitated that the helicopter change its flight path to negotiate them.
Figure 2.15: Helicopter Straight Move.

Figure 2.16: Helicopter Left Turn.

Figure 2.17: Helicopter Right Turn.

Figure 2.18: Helicopter Up Move.

Figure 2.19: Helicopter Down Move.
To begin, the helicopter plans an initial path and starts following this path. At each step of the algorithm, the helicopter determines whether it can continue flying along the path it has created, or if it must plan a new one to avoid some projectile. It does this using the same algorithm it used to plan the initial path, but with significant knowledge of each projectile, including where it is, what direction it is heading in, and how fast it is moving. Using this knowledge, the helicopter is able to replan amongst obstacles in its configuration space-time [7] to avoid all of the projectiles, despite the fact that it moves more slowly than the projectiles do. This sort of planning is an example of receding horizon search, which is discussed in greater detail in Chapter 5.
Chapter 3

Strategies for Dealing with Heavy-Tailed Examples

In the course of our solving several problems using RRTs, there appeared a
dichotomy of problem types in terms of the distribution of sizes of solution trees.

The first type of problems have sets of solutions that have few small solution
sizes, and the number of solutions increases as the size gets bigger, until a point is
reached at which there are many solutions concentrated in a small range of relatively
large solution sizes. At sizes above this range, no solutions occur.

The second type of example has solutions that occur at a more consistent
frequency throughout. There is a range in which most of the solutions occur, as there was
in the previous type of problem. This type of problem is different however, because after
this range, solutions continue appearing, unlike in the previous example. For these types
of examples, the frequency of solutions never falls to zero. This phenomenon is called
heavy-tailed behavior [10], and it is discussed in Section 3.1.

The difference between these problem types caused the RRT algorithm to perform
better or worse depending on the type of problem. Specifically, the RRT algorithm was
much more successful at solving the first type of problem than it was at solving the
second type of problem. New strategies were necessary to improve the odds of solving
the second type of problem. These strategies are described in this chapter, including the
algorithm used to generate solutions using this strategy. There are also definitions of the
terms used to discuss these strategies, and the measurements used to compare them.
3.1 Heavy-Tailed Behavior

A problem exhibiting heavy-tailed behavior has a heavy-tailed distribution of solutions. A heavy-tailed distribution is characterized by an infinite moment (i.e., an infinite mean or variance) [10]. Because of this extreme variance, researchers use median, as opposed to mean, to measure search difficulty, because the median tends to be much more stable. For a discussion of why the median is a more stable metric than the mean, see [10].

The best way to determine that an example might be heavy-tailed is by using a semi-log survivor plot, where one plots the probability that a solution was not found vs. solution size. To make this concrete, the notion of cumulative probability must first be defined. Given a set of solutions where \( x \) represents the size of an individual solution, let \( f(x) \) represent the empirical probability of reaching a solution in exactly \( x \) steps. Given this, let \( F(x) \) equal the cumulative probability of finding a solution in at most \( x \) steps. In the survivor plot mentioned above, each point \((x, y)\) is a pair where \( x \) is an individual solution size, and \( y \) is \( 1 - F(x) \), the complement-to-one of the cumulative probability (i.e., the probability that a solution was not found before size \( x \)). This plot shows how the probability of not finding a solution changes as the solution sizes get bigger. As the line moves down, the probability of finding a solution increases. Figures 3.1 and 3.2 below depict semi-log survivor plots for a regular distribution and a heavy-tailed distribution, respectively.

In Figure 3.1 the line decreases rapidly toward \(-\infty\). In that figure, there seems to be a size by which the large majority of all solutions will have been found. This number is approached exponentially, with the implicit limit being the “maximum” solution size.
In such a distribution it is almost guaranteed that all solutions will be found before reaching this solution size.

![Survivor Plot of a Regular Distribution](image)

**Figure 3.1: Survivor Plot of a Regular Distribution.** This is a Gaussian distribution with mean 5 and variance 10.

In Figure 3.2, the general trend of the data is a straight line. This is indicative of a heavy-tailed distribution. It shows that as the solution sizes get larger and larger, the probability of not finding a solution does not precipitously decrease. As the solution sizes continue to grow, there is never a point at which all solutions are likely to have been found. Hence, there is no implicit limit to the size of solutions. This causes RRT searches which limit the search by the number of nodes to fail at consistently finding solutions, because there are always cases in which a solution will exceed the size allocated to the tree no matter how much size is allocated.
Survivor plots can be produced for any group of solutions to a problem. Once the solutions are gathered and plotted, the graph will indicate whether or not the example exhibits heavy-tailed behavior. Knowing this, the RRT generation algorithm can be amended to take advantage of this knowledge, improving the odds of finding a solution and the average amount of time to find that solution. Indeed by examining the plot, the search can be structured for the most efficient use of the available search nodes, using the strategies of multiple trees and restarts, which are discussed in the following sections.

The tendency of heavy-tailed problems to occasionally create solutions that are orders of magnitude larger than the mean causes traditional RRT algorithms to fail at finding solutions quite frequently. A traditional RRT algorithm allows for only one tree to find a solution, and the tree only has one chance to find the correct solution. Even
further, the number of nodes is usually limited, meaning that the tree will not have
enough nodes to find the outlying solutions when the sequence of random numbers
creates one. This chapter proposes an amendment to the traditional RRT algorithm to
collect better deal with heavy-tailed problems. The new methods are called \textit{multiple trees} and
\textit{restarts}. \\
\textbf{The Multiple Trees and Restarts Methods}

The multiple trees method requires a change in how a potential solution is
defined: it considers each potential solution in terms of its potential nodes, the number of
which is finite. This amendment to the algorithm suggests splitting these nodes into
multiple trees rooted at the starting point of the algorithm, instead of lumping all of the
nodes into one tree. This method has two benefits for heavy-tailed problems. The first of
these is that the multiple trees often branch out in different directions allowing the tree to
explore different parts of the space. In doing this, the potential solution has a better
chance to avoid bad initial choices, which will be demonstrated later to be damaging to
an algorithm’s ability to be successful. The second benefit is that with the nodes split
between multiple trees, calculations will be faster than if all of the nodes are in one tree.

It should be noted that this method is almost exactly the same as allowing trees to
restart themselves. Other research on heavy-tailed problems, such as that presented in
\cite{10}, suggests allowing a solution to restart itself when it has gone too far without finding
a solution. It should be noted that this strategy is almost identical to the multiple trees
strategy, because the multiple trees strategy is similar to a single tree that receives a
number of restarts. For example, if each solution is allowed to restart itself 3 times, this
is similar to four trees rooted at the same point. The only necessary factor is that the
restart condition for the tree with restarts is the same as the failure condition for each tree in the multiple-tree strategy. This way if the same tree were generated using both methods, the single-tree strategy would restart the tree at the same point that the multiple-tree strategy would begin a new tree. At this point, the only potential difference between the two strategies is that the restarts strategy will stop as soon as it finds a solution. In our implementation, the multiple trees strategy works this way as well, because it generates one tree at a time, and only generates the next tree if the first one fails. These strategies are different conceptually, but in implementation they are identical. Section 3.4 further illustrates that these strategies produce similar results.

3.2 Background

This section outlines a few topics which will be useful in discussing the ideas of this chapter. Section 3.2.1 discusses the algorithm used for building RRTs in this chapter. Section 3.2.2 discusses a strategy for determining the tree size cutoff for a specific problem. Section 3.2.3 introduces and defines a few terms, and explains how they will be used in this chapter. Section 3.2.4 addresses the various ways the trees in this chapter are measured and how they are compared.

3.2.1 Algorithm

The algorithm used in this chapter for the multiple tree search is similar to the method described by Algorithm 2.3. The only difference is that in this case if the algorithm fails to find the goal, it scraps the old tree, and starts a new tree, continuing this behavior until it has reached its maximum number of trees. If the algorithm has not found the goal by the time it has reached this limit, then it is deemed unsuccessful, and
returns the final failed tree. Algorithm 2.3 can be modified to implement this change as follows.

**Algorithm 3.1 BUILDRRT\((start, goal)\) with Success Termination and Restarts**

1. **Tree.INITIALIZE\(start\)**
2. **FOR** \(i = 0; i < \text{MaxNumTrees}; i++\)
3. **WHILE** ((Tree.SUCCESS\(goal\) is FALSE) AND \(k < K\))
4. \(x_{\text{rand}} = \text{Tree.SAMPLESTATE()}\)
5. \(x_{\text{near}} = \text{Tree.NEARESTNEIGHBOR}(x_{\text{rand}})\)
6. **Tree.NEWNODE**\(x_{\text{rand}}, x_{\text{near}}\)
7. \(k++\)
8. **IF** (Tree.SUCCESS\(goal\))
9. **RETURN** Tree
10. **RETURN** Tree

**3.2.2 Cutoff Determination**

When using either of these strategies it is important to determine when a tree is getting too large. Without knowing when it is time to restart a tree, the benefits of either of these strategies are lost. It is crucial to determine the correct size at which to cutoff a potential solution.

The question that needs to be resolved here is “When is adding nodes to the tree no longer worth the added cost of computation?” The answer is relative to the problem to be solved, and the resources available. If there is no limit to how long the trees can run and how much memory they can occupy, then there should be no limit to how many nodes can be used to solve the problem. In the more likely situation in which time or
computational resources are limited, then there will be limits, which can be set depending on how important it is to see every solution to completion.

An easy way to figure out where a good cutoff might be is to run a large set of trees with a very large maximum solution size. Take these results, break them up with several possible cutoffs, and see at what point trees stop finding solutions as frequently as is desired. One convenience about this method is that several possible cutoffs can be tried with the same set of data. Another is that it does not require any complex thought or computation.

In doing this, it is important that all trees finish with a solution. If there are trees that do not finish before the maximum solution size, the maximum solution size must be increased. Only trees that finish can be classified, because trees which do not finish cannot be fit into any particular grouping. It is impossible to tell if they would have required another 3 extra nodes or an extra 10000 nodes to finish.

Once a group of trees which finish has been obtained, the results can be broken up into groups of approximately the same size. By grouping solutions in this way, it is easy to see when solution frequency begins to taper off. As solution size increases and solutions become less frequent, it becomes much easier to decide where examples should be terminated as failed, and start over. This data can also indicate whether one is facing a heavy-tailed example or not. If there appears to be a solution size by which all solutions have terminated, then the problem may have a regular distribution. If, however, solution sizes keep appearing in every grouping, then that would indicate a heavy-tailed distribution. This is essentially the same information that comes off of a survivor plot.
Table 3.1 shows an example of this strategy, used to determine where the cutoffs should be for the Stair Climber example of Section 2.4.1. In these groupings of solution sizes, a logical cutoff might be between 12 and 15 thousand, because all of those groups before here have 10 or more solutions, and all groupings after this point have fewer than 5. The fact that there are 10 solutions which happen between 12000 and 15000 is interesting, because they could be anywhere in this range. Further investigation would likely be necessary to find out where the cutoff should be. If all of the examples finish before 12100 nodes, then that should probably be the cutoff. If, however, they happen very close to 15000, then maybe those examples are not important or require too much additional calculation, and the cutoff should be 12000. Either way, it is easy to see how a table like this one can be used to find a cutoff. It clearly shows where the majority of the trees are finishing. In this case, the vast majority fall between 6000 and 12000 nodes, with the remainder happening gradually as sizes approach 21000. No matter what the constraints are on the resources used to solve the problem, a table like this makes it easy to find an appropriate cutoff.

<table>
<thead>
<tr>
<th>0 - 3000</th>
<th>3000 - 6000</th>
<th>6000 - 9000</th>
<th>9000 - 12000</th>
<th>12000 - 15000</th>
<th>15000 - 18000</th>
<th>18000 - 21000</th>
<th>&gt; 21000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>38</td>
<td>45</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: Cutoff Finding Example.

Those that fell in the final bin reached the maximum solution size without finding a solution (in this case there were none). These proportions stayed approximately the same when running experiments with a set of trees grown to each of these individual cutoff sizes. Using the max of each range listed above as the maximum tree size,
representing a potential cutoff, similar proportions of trees finish in each group as above. Also, a similar proportion reaches the cutoff size. Tables 3.2-3.8 show this explicitly.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>&gt; 3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 3.2:** Max Tree Size of 3000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>&gt; 6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 3.3:** Max Tree Size of 6000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>6000-9000</th>
<th>&gt; 9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>38</td>
<td>62</td>
</tr>
</tbody>
</table>

**Table 3.4:** Max Tree Size of 9000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>6000-9000</th>
<th>9000-12000</th>
<th>&gt; 12000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>40</td>
<td>33</td>
<td>27</td>
</tr>
</tbody>
</table>

**Table 3.5:** Max Tree Size of 12000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>6000-9000</th>
<th>9000-12000</th>
<th>12000-15000</th>
<th>&gt; 15000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>27</td>
<td>41</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

**Table 3.6:** Max Tree Size of 15000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>6000-9000</th>
<th>9000-12000</th>
<th>12000-15000</th>
<th>15000-18000</th>
<th>&gt; 18000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>33</td>
<td>45</td>
<td>15</td>
<td>15</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 3.7:** Max Tree Size of 18000.

<table>
<thead>
<tr>
<th>0-3000</th>
<th>3000-6000</th>
<th>6000-9000</th>
<th>9000-12000</th>
<th>12000-15000</th>
<th>15000-18000</th>
<th>18000-21000</th>
<th>&gt; 21000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>29</td>
<td>47</td>
<td>19</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 3.8:** Max Tree Size of 21000.
This information confirms what was learned in the first set of data, run to the very large solution size of 50000. This indicates that running a large set of examples to a very large solution size will yield all the data that is necessary to find an appropriate cutoff.

Using data like this, groupings of solutions can be added to solutions that would have been found by increasing the cutoff size. By checking the percentage of all solutions found at this cutoff, one can find a cutoff which will capture the necessary percentage of all solutions.

3.2.3 Terms and Definitions

The following terms and definitions will be useful in comparing the performance of different search strategies:

The growth distance is the distance of each of the individual steps in the tree added together. This is computed by multiplying the number of nodes by the step size. This shows the distance the tree would reach from the origin if all of the steps taken were in the same direction and there was no branching. This growth distance is a way to compare trees with differing step lengths and node counts. For example, a tree that has a step length of 1 distance unit and 10000 nodes would have the same growth distance as a tree that had a step length of .25 and 40000 nodes.

The reachable state space includes all of the state space that can be reached from the initial point without violating the dynamics of the problem. The reachable state space may, however, contain some regions that the tree must fill according to the problem dynamics. In these regions, the tree will have complete coverage. An example of this type of region is a straight line.
The **coverage region** is a subset of the reachable state space which excludes regions that are ensured to be covered completely by the problem dynamics. Coverage metrics are computed only on the coverage region. By excluding regions where the problem dynamics force perfect coverage, coverage metrics computed on the coverage region give better comparison data for different configurations of the same problem.

Figure 3.3 below illustrates the idea of a coverage region. Assume that the trapezoidal green area plus the horizontal red lines extending from the trapezoidal green area are the places that the tree is able to reach. Given this, the trapezoidal green area is the coverage region, while the horizontal red lines are excluded from the coverage region.

Figure 3.3: Coverage Regions.

### 3.2.4 Measurement and Comparison

To say that one tree is larger than another is to say that if both trees used all of their nodes, the first tree would have more nodes than the other. This idea is used only to differentiate between different trees in the same set of tests. For example, using a growth distance of 10000, a large tree would be one that had 30000 nodes and a step size of .33, while a small tree might have 10000 nodes and a step size of 1. Despite the fact that both trees have the same growth distance, the first tree is considered larger than the second,
because it has more nodes. In each set of tests, there were never more than three types of
trees used, so the adjectives large, medium, and small are sufficient for comparison.

Five metrics were calculated on each set of trees. These five metrics were: the
final size of the tree, the time it took to generate the tree, and three of the four coverage
metrics discussed in Section 1.4. Those three coverage metrics are dispersion, average
dispersion, and the normalized version of the Kim metric. The reason for omitting the
original version of the Kim metric is that it does not provide any information that is not
already provided by the combination of the normalized Kim metric and the distance
between the points in the dispersion measurement grid. The coverage metrics were
calculated using a regular grid of points with one point at each pair \((x,y)\), where both \(x\)
and \(y\) are multiples of 10, and in the range \([0, 200]\).

It should be noted that the coverage metrics only have useful meaning in cases
where the algorithm extends to maximum size. Since these cases all have the same
number of nodes, it is meaningful to see which one did a better job of covering the space.
In cases where the algorithm does not reach maximum size coverage metrics lose their
value, because the algorithm stops searching as soon as it reaches the goal. This leads to
trees which have different numbers of nodes. It follows logically that a tree that has more
nodes should do a better job of covering the space, so when trees have different numbers
of nodes it is not as meaningful to compare their coverage.

### 3.3 Large versus Many Trees

The strategy we suggested for dealing with heavy-tailed examples was to split the
nodes into multiple trees, and see if this does a better job of finding solutions than would
a single tree. Alternatively, this is similar to asking the question, “Given a predetermined
number of nodes, how best can they be used to solve a problem?” This question deals with the utilization of limited resources.

This section compares the default RRT strategy of using all of the nodes in one tree with the new strategy of splitting nodes into multiple trees in the same space, rooted at the same node. The first strategy offers the benefit of being able to explore the most completely, because it has more nodes in the same tree. The second strategy offers a different advantage. Because it splits its nodes among different trees, it often branches out in different directions, so although it is not able to explore any particular path as thoroughly as a single tree, it is often able to explore paths that go in different directions. While none of these trees will be able to explore as well as the tree in the first strategy, it will be shown later that the set as a whole can sometimes explore better than can the single tree from the first strategy. In this scenario, the limited resources are the nodes that make up the RRTs.

In some of the initial testing, the ability of the algorithm to reach the goal region was largely dependent on what the tree did in the early stages of its development. In some cases the tree, by making poor initial choices, would almost completely eliminate from its exploration sections that it should have been able to explore. If the goal region was one of these sections, then the utility of future nodes was compromised. In such cases, the tree would continue in failing to reach a reachable goal even after increasing the limit drastically. Since the most important concern in a verification problem is reaching the goal region, all of these nodes are essentially wasted, because they will not be able to help the tree reach the goal. Not only that, but these wasted nodes may give the tree superior coverage of the search space, indicating that it is more likely that a path
to the goal region does not exist. This does more than waste time and computation. It strengthens the bad information already provided, because a path to the goal does exist. In this case, one remedy is to start over with a new tree, and hope that it does not make similar bad choices early on in its development.

The fact that these bad initial choices were so damaging to the results caused speculation that in using a specific number of nodes, it might be better to generate several trees each with a portion of the nodes as opposed to one large tree with all of the nodes. By having fewer nodes and more trees to search with, the algorithm can restart itself if it finds that it has grown too far without finding the goal. This gives the algorithm several chances to avoid bad initial choices, and since verification only requires one path to the goal region, only one tree in the group needs to be successful. This should provide some protection against these bad initial choices, and better allow the algorithm to find a path to the goal region.

In testing this strategy, three problems were examined: the Hovercraft, the Train Gate, and the Pendulum. The results are given in the following subsections.

3.3.1 Hovercraft

We first tested the multiple trees strategy with the Hovercraft example of Section 2.4.3. In the initial testing for this example, trees with the smallest step size had a success rate of only 40%, but the average tree size for a successful tree was less than 5000 nodes, despite the fact that 10000 nodes were available. This means that in the average successful case, over half of the nodes were not used. This may have been due to bad initial choices, and this data may have meant that trees could be ruled out as failures after only 5000 nodes, and the remaining 5000 nodes could be used to generate a new
tree, and see if that tree would be successful where the initial tree failed. If the average success rate of 40% holds, then it is likely that two trees, with approximately a 40% chance of success each, would have a higher total success rate than 40%.

The expected rate of success for a pair of trees can be computed if the success rate for a single tree of the same size is known. The success rate for the pair of trees is equal to 1 minus the probability that both trees fail. Given the 40% success rate from above, the probability that two consecutive trees would fail is \((1 - .4)^2 = .36\). Thus, the expected success rate for two trees is 64%.

**Terms and Definitions**

These terms will aid in discussing the success of the Hovercraft example.

A *success example*, or *success tree*, is a tree that reaches the goal before expanding to the maximum tree size.

A *failure example*, or *failure tree*, is a tree that does not reach the goal and thus expands to its maximum tree size.

**Testing Strategy**

Three sets of data were calculated for this section, one set for each step size. The step sizes used were .25 distance units, .33 distance units, and .5 distance units.

In each set of data, there were multiple configurations tested, each consisting of a set of trees totaling the same growth distance as all other configurations. For each configuration, the growth distance was 10000. With step size .25, this allowed three different sets of trees: one tree of 40000 nodes, two trees with 20000 nodes each, and four trees with 10000 nodes each. With step size .33, three different sets of trees were tested: one tree of 30000 nodes, two trees of 15000 nodes each, and three trees of 10000 nodes
each. With step size .50, only two different sets of trees were tested: one tree of 20000 nodes and two of 10000 nodes each. The differing step sizes restricted the configurations that could be used with each step size. For this reason, Tables 3.9–3.12 each have three bars for the sections of one and two trees, and only one for the sections of 3 and 4 trees. Each step size had a single-tree configuration and a two-tree configuration. Only step size .33 had a three-tree configuration, and only step size .25 had a four-tree configuration.

**Results**

In comparing one large tree against several smaller trees, dividing the nodes among multiple trees improved performance, but the performance gain declined quickly as the number of trees increased. Trees of all step lengths experienced significant improvement from dividing the nodes among two trees as opposed to lumping them all into one tree. Further division, however, caused the performance of the sets of trees to diminish. The shortest step size $\varepsilon = .25$ required 40000 nodes to reach a growth distance of 10000. When all of the nodes were in a single tree, this tree was able to find the goal 81% of the time. When nodes were split into two trees of 20000 nodes apiece, the success rate jumped, to 92%. This is close to the expected success rate of 96.3%, the difference likely coming from solutions that would have been found between 20000 and 40000 expanded nodes. However, when the nodes were further divided into 4 trees of 10000 nodes apiece, the performance dropped to 80.6%, even lower than it was with all of the nodes in a single tree. Similar trends were found with a step size of .33, and .50.

This data indicates that in finding paths to goal regions, it is not always enough to just throw a lot of nodes at the problem. It is important that they be organized into trees
in such a way that they will be most likely to grow a path to the goal. If a tree has developed to its maximum size and not found the goal region, the problem might not be insufficient nodes. The tree may have grown in a peculiar way, and it might be best to start over. This is also beneficial because computationally it is easier to plan an entirely new tree than to add the same number of nodes to a tree that is already well developed. This is true because the \textsc{NearestNeighbor}(x_{\text{rand}}) function must examine each node already in the tree, which makes the growth rate of the tree $O(n^2)$. As discussed before, it also has the benefit of new trees possibly avoiding the bad choices the previous tree may have made. If the same number of nodes were added to the first tree, these nodes too, would be subject to any bad choices the first tree made in its development. Table 3.9 shows the success rates of the different step sizes. Each bar indicates the cumulative results of 100 runs with that configuration.

![Success Rate vs. # Trees](image)

Table 3.9: Success Rates as a Function of Step Sizes.

---

3 If there are $n$ nodes in the tree already, to add one new node, the algorithm must check each of those $n$ nodes to see which is the closest to the new node. To add $n$ new nodes, each of the nodes already in the tree must be examined once for each new node, giving a growth rate of $O(n^2)$.
The reason that bad initial planning can be so damaging to the success rate of a tree, is that these choices can impair the coverage of the tree. Two smaller trees frequently branch out in different directions, and thus cover the space better. This characteristic is evident when comparing the coverage statistics of the trees and groups of trees. With the step size of .25, the single large tree had dispersion, average dispersion, and normalized Kim metrics of 70.2, 22.3, and .59 respectively. When these metrics were calculated on the union of the two smaller trees, improvement was noticeable, with values 52.0, 12.4, and .44. In all metrics, the union of two smaller trees covered the space better than did the one larger tree. The coverage metrics also demonstrate the reason for the failure of further division of nodes. The union of the four smallest trees returned values of 72.14, 19.0, and .59. The dispersion, which indicates the biggest hole in the coverage, is larger for these small trees than for either of the other configurations. However, the average dispersion, which measures coverage over all of the space, is slightly better than that of the single large tree. This is because the four small trees cannot reach very far into the search space, leaving a large hole on the farthest areas from the origin, where the goal region is. This causes the large dispersion. This is also the reason that these smaller trees fail. Because of each tree’s limited growth distance, it is difficult for them to reach all the way to the goal. However, since the set does a better job of covering the area it can reach, the average dispersion is markedly lower than for other configurations. These trends were also seen with the other step sizes of .33 and .50. These numbers are illustrated in Tables 3.10–3.12. Again, each bar represents the average results of 100 runs for that configuration.
Table 3.10: Dispersion vs. Number of Trees in Group.

Table 3.11: Average Dispersion vs. Number of Trees in Group.
Table 3.12: Normalized Kim Metric vs. Number of Trees in Group.

All of this data shows that the Hovercraft problem benefits from the division of nodes among separate trees. An RRT trying to solve this problem usually finds its goal early in its development. If it has not, then it probably has made mistakes early in its development which have compromised the rest of the tree. If so, then it is not useful to develop that tree any further to solve the problem. Rather it is better to assume that it will not succeed, since the probability is high that it will not. Thus, the tree should be thrown away, and a new tree should be grown.

In examining the results of the Hovercraft example, it stands out as a problem with a heavy-tailed distribution. In all of the testing for the Hovercraft, the mean solution size was on average 1014 nodes larger than the median solution size. This was due to a few instances which were much larger than the majority, and which fell well outside of the standard deviation of the examples. These very large instances were so infrequent, however, that they did not affect the median, allowing it to stay relatively constant. This is a primary characteristic of a heavy-tailed example. The survivor plot for the
Hovercraft appears in Figure 3.4. In this plot, the line has a few humps, but the overall shape is a straight line, indicating a heavy-tailed distribution.

![Figure 3.4: Survivor Plot for Hovercraft Example.](image)

**3.3.2 Train Gate**

We also tested multiple versus single trees for the Train Gate example of Section 2.4.2. In the initial testing of the Train Gate example, the data showed that this problem might also benefit from division of nodes. For example, suppose one had a computational resource of 5000 nodes. Next, in testing with a step size of .1, an $\alpha$ value of 9.8, and a biasing frequency of 10 (the combination that required the most nodes to reach the goal) the tree required just over 2000 nodes to reach the goal on average, leaving 3000 nodes unused under the 5000 node maximum. If the average tree that finds a path to the bad region is using less than half the nodes, the tree can be split, using half
of the nodes in each of two trees in case one of the trees makes bad initial choices. We perform these and other experiments below.

**Terms and Definitions**

The results for this example are addressed in terms of the algorithm’s ability to correctly predict an invalid value, because this is the primary purpose of the algorithm.

For the purposes of this example, a few terms are defined below, along with a clarification of the difference between their usages here, as opposed to how they have been used in previous sections.

The reason for using different definitions here is that this example is fundamentally different in that there is no goal region. In this problem, there is a “bad region” which the tree must avoid, whereas the other problem had a “goal region” that it must reach. In this example, there are some parameter choices for which the problem specification does not allow the tree to reach the bad region. There are others in which the problem specification allows the tree to reach the bad region, although it might not necessarily do so in a particular run. In the previous example, all configurations are *a priori* able to find the goal. More to the point, in this example, some examples should fail and some examples should *a priori* succeed.

A **success case** is a case where the algorithm cannot find a counter-example based on the problem specification. The problem is specified in such a way that the tree is unable to reach the bad region. Because success is expected, this is called a success case.

A **failure case** is a case in which the problem specification allows the algorithm to find a counter-example. The problem is specified in such a way that the tree is able to reach the bad region. Because a failure is expected, this is called a failure case.
A **successful tree** is a tree which conforms to the expected outcome of the problem specification, either finding a counter-example if it should be able to, or expanding to maximum size without finding a counter-example if it should not be able to find a counter-example. This differs from the previous example where a successful tree was a tree that reached the goal without expanding to maximum size.

A **failed tree** is a tree which does not conform to the expected outcome of the problem specification, failing to find a counter-example if it should have been able to find one (technically this could also mean a tree which finds a counter-example when it should not have been able to find one, although this was not possible since all trees conformed to the problem specifications). This differs from the previous example where a failed tree was any tree which expanded to maximum size without reaching the goal.

**Testing Strategy**

Our testing strategy used several groupings of nodes, each using a step size of .25 distance units. The first sets of tests used 10000 nodes, in two different groupings. One set was a single tree with all 10000 nodes; the other was comprised of two trees of 5000 nodes each. Each test was biased once every 10\textsuperscript{th} iteration.

A second grouping of tests used 5000 nodes in similar groupings. The first set used a single tree with 5000 nodes, while the second set used two trees of 2500 nodes each.

Since RRTs cannot prove that a configuration is valid, only configurations with invalid \( a \) values were tested, to draw conclusions about the algorithm’s ability to predict these invalid configurations. This means that all of the testing was done with failure cases. Also, the only results which can be compared are those in which trees sometimes
failed to find the bad region. If the trees were always successful in predicting invalid values, there is no way to improve their success rate.

**Results**

Splitting nodes between multiple trees for this example had mixed results, but generally was not beneficial. In some cases splitting up nodes between two trees caused a slight increase in the success rate, but in other cases it caused a slight decrease in the success rate. In no case was the success rate affected to any great degree.

In the first group of tests, there was little conclusive evidence either way. There were only two failure cases in which there were any failed trees. With $\alpha$ equal to 9.8, the large trees produced only 6 successful trees compared with 44 failed trees, a 12% success rate. The pairs of smaller trees produced only 5 successful trees on this $\alpha$ value, and 45 failed trees, only a 10% success rate. This shows that the larger trees did slightly better, but the difference is small: only one successful tree.

With $\alpha$ equal to 10.0, the large trees produced 39 successful trees and 11 failed trees (78% success rate), while the pairs of trees produced 41 successful trees and 9 failed trees (82% success rate). Here, the pairs were slightly more successful, but again, the difference is small. The fact that in both cases the difference between the two success rates was negligible and that a different grouping performed better in each case tells us that tree splitting may not provide any benefit for this example.

There is one thing that must be examined as a potential reason that there is not a large difference in the success rate for these cases. It could be that there are too many nodes to obtain any useful information. A glut of nodes might be allowing too much exploration capability for the problem to be reasonably analyzed. In order to test for
improvement by using a different strategy, there must be ample opportunity for the
algorithm to fail. If one strategy almost always reaches the goal after 7000 nodes, and the
other almost always reaches after 4000, and if testing extends to 10000 nodes, there will
be no way to tell the difference between the two by the success rate. This is the
motivation for the second group of tests.

The second group of testing showed a small but clear difference in the success
rate of the two algorithms. Again, there were only two failure cases in which there were
any failed trees, and they were at the same $\alpha$ values as in the previous group of tests. For
$\alpha$ equal to 9.8, the single trees were only able to predict that this value was invalid in 4 of
the 100 tests. Although this is a very low success rate it is still better than the pairs of
two smaller trees, which were only able to predict this invalid value once out of the 100
tests. For $\alpha$ equal to 10.0, the single large tree was able to correctly determine that this
value was invalid 56 times out of 100, while the pairs of smaller trees were only able to
predict correctly 50 times out of 100. This difference is slightly bigger in this set of tests
than it was in the first set, but is still less than 6%, despite the fact that there were
numerous opportunities for failure.

Table 3.13 summarizes the success rates of each of these configurations
compared. Each bar represents the cumulative data for all of the runs of that
configuration (50 for tree size = 10000 and 100 for tree size = 5000). In three of the four
configurations, the strategy with a single tree outperformed the set of two trees. What is
more notable, however, is that in each case the difference is small, with the largest
instance being only 6%.
Table 3.13: Success Rate vs. Problem Configuration for the Train Gate.

The results of the sets of tests showed that this problem did not benefit from division of nodes among separate trees. In all cases, the difference in the success rate between the two approaches was small, and usually the approach with the one large tree did better. This shows that dividing nodes among trees is not always a good idea to improve success rates (see Section 3.1 for an explanation).

We should note also that the Train Gate problem is not heavy tailed. The survivor plot for the Train Gate example with step size of .25 and a maximum tree size of 5000 appears in Figure 3.5. As you can see in this plot the trend is downward, approaching 4500 almost asymptotically. This indicates that the Train Gate example has a regular distribution, and not a heavy-tailed distribution. This, coupled with the fact that the median and mean are approximately the same size (in the examples of 500 nodes, the mean was 2126 and the median was 1977), provides convincing evidence that the Train Gate problem has a regular distribution of solution sizes. We conjecture that this is the reason that splitting trees did not improve the success rate of this example.
3.3.3 Pendulum

The Pendulum problem of Section 2.4.4 was tested to show that it is possible to determine that an example is heavy-tailed without comparing single and multiple trees. The testing strategy for this problem used a maximum tree size of 50000 and a step size of $\varepsilon = .1$. With this configuration, there were only two failures and the average tree size was 13800. However, the median of 9600 was much smaller, and as solution sizes got bigger, the solutions found do not trail off.

Most of the solutions (57/100) have less than 12000 nodes. However, after 15000 nodes, larger and larger solutions keep appearing at a consistent rate (approximately 3 more solutions in each successive group of 3000 additional nodes). This indicates that the Pendulum problem is heavy-tailed.

Figure 3.5: Survivor Plot for Train Gate Example.
A look at a survivor plot for this problem further indicates that its solutions follow a heavy-tailed distribution. A new set of data was generated to create this plot. This new set had a maximum tree size of 100000 nodes, which allowed all examples to finish. In this set of solutions, the mean solution size was 13700 and the median was 10300, consistent with the previous set of testing. Again, the majority (59/100) of solutions were found before they reached 12000 nodes and after 15000 nodes there were just a few additional solutions at increments of 3000 nodes. This continued until 42000 nodes, when all but one solution had been found. The remaining solution took more than 35000 additional nodes, finishing at over 77000 nodes. Such a drastically larger solution is a further indicator of a heavy-tailed example. The survivor plot confirms this.

Figure 3.6 shows the survivor plot for the Pendulum problem. This plot confirms the conclusions that were drawn from examining the data generated for this problem. Looking at this plot, the trend toward the end is a straight line. This plot, coupled with the data above, convincingly shows that the Pendulum problem has a heavy-tailed distribution of solutions.
3.4 Multiple Trees versus Restarts

To illustrate that restarts and multiple trees produce similar results, the Stair Climber example from Section 2.4.1 was tested twice. The first testing strategy (the *multiple trees* strategy) was the same as the previous strategies, i.e., using a set of trees as a potential solution to solve the problem. The second strategy (the *restarts* strategy) was slightly different. Instead of having a limited number of trees, this example would allow a tree to restart as many times as necessary to achieve success in the size allotted. The first strategy has been talked about at length in previous sections of this chapter, and requires no additional explanation. The second is explained in more detail in the next paragraph.
The restarts strategy allows for an unlimited number of trees to be grown. The restriction is instead on the number of successes that will be generated. Testing in this way allows for a variable success rate, although the number of successes is fixed. This is similar to the previous tests, where the total number of possible solutions was fixed, and the success rate was variable because the number of successes was variable. In both cases, the success rate indicates how well the algorithm can solve the problem presented. Results of testing with the Stair Climber example of Section 2.4.1 are expected to be similar for both strategies since each is allowed to use multiple trees to find the next solution, and is given sufficient opportunity to fail, and start again with a new tree. This different strategy is examined to illustrate that there is no difference in results between this strategy and the previous one. Because this section is focused on restarts as opposed to multiple trees, the restarts strategy is discussed first.

### 3.4.1 Restarts

The restarts strategy testing was done using a biasing factor of 10 (defined in Section 2.2.3), and a step size of 5. The initial set of test data was generated with a maximum tree size of 50000 nodes, in order to determine an appropriate cut off for the following sets of data. Using the method described in Section 3.2.2, it was determined that the appropriate cut off was of 12000 nodes, and several sets of data were compiled. In each of the remaining sets of data, examples were allowed to run to maximum tree size, and if unsuccessful, the failure was recorded, and the example restarted. This was continued until 100 successes were recorded.

The 4 groups of tests had cutoffs (i.e., maximum tree sizes) of 9000 nodes, 12000 nodes, 15000 nodes, and 20000 nodes. In 2 tests to 100 solutions with 9000 nodes
possible each, the problem failed 192 and 293 times, respectively. When the cutoff size was increased just 3000 nodes to 12000, there were only 36 and 49 failures. An increase of just another 3000 nodes to 15000 reduced the number of failures to 18 and 13, respectively. At the final maximum solution size (cutoff = 20000 nodes), there were only 2 trials which did not succeed. These numbers show that the cutoff determination strategy is effective: the biggest decrease in failures came in moving to the proposed solution size.

Although the failure results for this example showed more failed trees than would have been expected, the general trend still shows that this is a problem with a regular (i.e., not heavy-tailed) distribution. These numbers decrease quickly, which is characteristic of a regular distribution. Indeed, as the maximum solution size increases, the number of failures decreases dramatically, by more than a factor of 2. A survivor plot of the Stair Climber example generated from the original cutoff determination data appears as Figure 3.7. It shows that in this example the curve takes a drastic turn down. This shows that the Stair Climber example has a regular distribution of solution sizes.
3.4.2 Multiple Trees

Again, this experiment was done using a biasing factor of 10, and a step size of 5. The strategy for this set of testing was kept as similar to the previous set as possible to facilitate ease in comparison. Again, there were four configurations run with maximum tree sizes of 9000 nodes, 12000 nodes, 15000 nodes, and 20000 nodes respectively (as was the case for restarts in Section 3.4.1). For each configuration, 50 examples were run. Each single example had four trees, all rooted at the same node.

In the set of examples with 9000 nodes possible each, the algorithm was unable to find a solution 8 times out of 50, a success rate of 84%. This is a much better success rate than the restarts strategy, but this is the success rate for examples, each of which can include up to four trees. In this strategy, one success might include 3 failed trees. In the
previous strategy, three failed trees followed by a successful tree would be a 25% success rate. To compare the two of these fairly, one needs to compare the success rate for all trees.

The success rate for individual trees in the multiple trees strategy at 9000 nodes per tree was 32.6% (42/129), a little worse than once out of every three trees. This is similar to the better of the two restarts strategy trees (the one using 9000 nodes), which succeeded at a rate of 34.2%, a little better than once out of every three trees. This is evidence that the two strategies produce similar results, although one must be aware that the results come in a different form depending on the strategy.

For the trees at other sizes, comparison results were similar overall, but fluctuated less with the multiple trees strategy. At 12000 nodes maximum per tree, the success rate for examples in the multiple-tree strategy was 100%, and the success rate for individual trees was 73.5%. The success rate for this configuration using the restarts strategy was 73.5% in the better case and 67.1% in the worse case, for an average of 70.2%. This is about the same as the multiple-tree strategy.

At 15000 nodes maximum per tree, the success rate for examples in the multiple-tree strategy was again 100%, and the success rate for individual trees was 84.75%. This is a little worse than at the previous size, but it did not manifest in the success rate of the strategy. The same configuration using restarts had a success rate of 84.75% in the worse case, and 88.5% in the better case, for an average of 86.6%. The results for this configuration are very similar for both strategies, as they were for the first configuration with 9000 nodes per tree.
At 20000 nodes maximum per tree, the success rate for examples using the multiple-tree strategy was 100%, and the success rate for individual trees was 96.2%. The same configuration using restarts had a success rate of 98.0%, about the same as multiple-tree strategy.

Success rates are summarized in Table 3.14 below. The results for the restarts strategy are the averaged success rate from both sets of experimental data.

<table>
<thead>
<tr>
<th></th>
<th>9000</th>
<th>12000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restarts</td>
<td>29.2%</td>
<td>70.2%</td>
<td>86.6%</td>
<td>98.0%</td>
</tr>
<tr>
<td>Multiple trees</td>
<td>32.6%</td>
<td>73.5%</td>
<td>84.6%</td>
<td>96.2%</td>
</tr>
</tbody>
</table>

Table 3.14: Success Rates at Each Cutoff Size for Both Strategies.

The difference in these two strategies is more in the expectation of what is to happen than in any actual results. Using a restarts strategy is a pessimistic way to approach the problem. The algorithm tries for success, and if it does not find success, it starts over again, and keeps repeating this behavior until it finds success. The multiple trees strategy understands that it will fail sometimes, and considers it good enough to find one success in several tries. The multiple trees strategy expects a certain number of failures for each success, and allocates enough trees per group to experience all of these failures, and one success. This, however, requires knowing how often to expect a success, a topic which has not been covered to this point. Fortunately, there is an equation in [11] which can be used to determine how many trees will be needed to achieve a success:

\[ P[S > s] = (1 - p)^{\frac{s}{c}} \cdot P[A > s \mod c]. \]
In this equation, $S$ represents the total number of nodes needed to find a solution, $s$ represents the number of nodes you are allowing to find a solution, $A$ represents the number of nodes the algorithm will need to expand to find a solution, $c$ represents the cutoff to restart a tree, and $p$ is $P[A < c]$, i.e., the probability that a solution will be found before the tree reaches the cutoff $c$.

The first term $((1-p)^{s/c})$ shows the probability that all of the trees that reach the cutoff will fail. The second term $P[A > s \mod c]$ shows the probability that the algorithm will be unable to find a solution in the nodes left over after all of the cutoffs. Multiplying these terms together gives the probability that $s$ nodes will not be sufficient to find a solution. Given a cutoff size $c$, we can use this equation to find an $s$ that yields an acceptable success rate. See [11] for a more detailed explanation.
Chapter 4

Forests

One of the problems with using RRTs to search hybrid space is that they require a lot of calculation, which can take a long time depending on the speed of the computer on which they are run. One possible way to deal with this is to break up the path from the starting point to the end point using multiple trees. The fact that each tree is concentrating on solving a different part of the problem can sometimes be exploited to solve the problem much faster than would be possible with only one tree. In this type of a search, the path is split up into sections, and each section is solved independently of the others. The partial solutions are then combined to create a full solution for the problem. This is most conveniently done when there is prior knowledge that the path will have to pass through specific points in a predetermined, ordered way before it reaches the goal. Then the search can be divided into several parts occurring between sequential pairs of these points, instead of from the initial point to the goal. In a case such as this, it is also useful to bias toward those specific points, instead of toward the overall goal. By splitting up the search, each path that must be generated is significantly shorter than the entire path from start to goal; and once a segment is generated, the algorithm stops searching in that area, and moves on to another, thus saving time by omitting further calculations for segments which already have a path.

As it turns out, this speeds up the computation quite a bit, not only by restricting the search to specific areas, but also by keeping tree sizes smaller, a fact which was shown in Chapter 3 to speed calculation. These savings can be seen not only in the
number of nodes required to find the path, but also and more importantly in the
calculation time required to find the path.

The introduction of other trees into the search space of the original tree opens the
door to some new schemes for biasing. Goal biasing has already been discussed in
previous chapters, but with several new interesting points in the search space, there are
new possibilities for biasing. If we have an ordered set of points that the solution must
pass through, it makes a lot of sense to bias towards these points sequentially. Instead of
biasing toward the ultimate goal point, the search can begin by biasing to the first
necessary point in the ordered set. After reaching this point, the biasing point can be
moved to the second necessary point until the tree reaches that point. This continues until
the tree reaches the ultimate goal point. This sequential biasing strategy was also tried in
[21].

Even if there is no set of necessary points, one can be created to facilitate
searching. If, for example, a tree were used to find a land route from New York City to
Los Angeles one might use Cleveland, Chicago, Denver and Las Vegas as intermediate
points. While it is certainly possible find a route from New York to LA without passing
through these intermediate points, having this list might help the tree find a more optimal
path quicker. Again, even though these points are not necessary, a similar strategy can be
employed, using the next closest point as the biasing point, instead of using the goal.

This strategy of using multiple bias points lends itself well to a *multiple-tree*
search. With a multiple-tree search, a single tree can be rooted at each of the bias points.
Each tree can then be biased toward the next point in the set. These trees can then be
generated sequentially or in parallel. Once all of the trees have found their paths from local origin to local goal, the paths can be linked together to create the overall solution.

A forest of trees can be likened to a graph, where each vertex is a tree and an edge is drawn between two vertices once the two trees become connected. In this case, a solution to the planning problem is a connected component of this graph that contains both the start tree and the goal tree. The root of each tree in the forest represents a vertex in the graph, and the branches of the trees represent possible edges. The branches of two separate trees can connect if there exist nodes in each tree which have the same values for all of their state components. This can happen when the trees are growing toward one another if a point in the sample tree is chosen as a biasing point, and it is within $\varepsilon$ of the growth tree. Once the branches of two trees connect with one another, the two trees are connected, equivalent to creating an edge in the graph connecting the two vertices represented by the roots of the trees. At this point these “tree vertices” are said to be connected.

When considering the problem in this fashion, the goal is to create a connected component of this graph which contains both the start point and the end point. If we can connect the graph in this way, then there is a path from the initial point to the goal point. This is equivalent to trees having found a path that will, by moving from tree to tree, be able to reach the goal from the initial point. Figure 4.1 shows this comparison.
Figure 4.1: Comparison of a Forest of Trees to its Graph.

The magenta lines show the path from node to node, and each node is highlighted by a dot. As in the picture of the graph, the initial point is red, the goal is green, and the intermediary nodes are yellow. As you can see, the trees have found a path from the initial point to the goal point, and have connected with two intermediary nodes, forming a connected component of the graph that includes both the initial and goal points.

While this technique is most useful in cases where there is a set of required intermediate points along any path from the start to the goal, this technique can also be used in cases where there is no such set. As illustrated above, a few chosen points can be assigned to a set of intermediate points, even though they may not necessarily be required in the path from start to finish. In this situation, it is useful to grow trees from all of the points in parallel instead of exploring each section in a separate step. This would allow a tree to potentially grow around an assumed intermediate point if it finds such a path.

4.1 Stair Climber

The Stair Climber example from Section 2.4.1 is ideal for a managed forest search because it is an example which has clearly defined sections into which the search can be divided. This example is split into four floors, so it can be naturally and easily
split into these four sections to be searched. Because this problem can also be solved with a single tree, it makes a great comparison example for this technique.

4.1.1 Testing

In this set of tests, the biasing frequency was 10, that is, out of every 10 iterations of the algorithm, $x_{rand}$ would be set equal to the bias point once, and would be generated randomly 9 times.

In the case where a single forward tree is used, the algorithm operates identically to Algorithm 2.3. When there is more than one forward tree used, each tree is grown to find its goal independently of the other trees, and the trees are grown sequentially in the order in which they would be traversed. The tree starting at the origin of the problem is grown first. Next is the tree whose origin was the goal for the first tree. This order is continued until the final tree has reached the goal of the problem. This algorithm is given below, for a group of forward trees. BUILDRRT($start[i], goal[i]$) works the same here as in Algorithm 2.3. The variables $start$ and $goal$ are replaced by $start[i]$ and $goal[i]$, which are the start and goal for tree $i$.

**Algorithm 4.1 Forward FOREST() Algorithm**

1. Tree trees[N]
2. FOR (i = 0; i < N; i++)
3.   trees[i].BUILDRRT($start[i], goal[i]$)
4. RETURN trees

For bidirectional searches, the algorithm is similar to the multiple forward tree algorithm, except that any two trees inhabiting the same search space are grown in parallel. This is true whether the search space is a single floor, or the entire building.
Because of this change, the FOREST() algorithm must be modified so that the
BUILDRRT(start, goal) function for each tree can run in parallel with the other. During
each iteration of the FOREST() algorithm each tree takes one step of its BUILDRRT(start,
goal) algorithm. After a step by either tree, the FOREST() algorithm checks to see if the
newly created node causes the trees to intersect. If the new node does not cause the trees
to intersect, then the algorithm continues creating nodes. If the new node does cause the
trees to intersect, then that section of the algorithm completes. If there are more floors to
be searched, then the algorithm moves to the next floor. If there are no more floors to be
searched, then the algorithm completes.

There is also a slight change in the biasing for bidirectional searches. In a
bidirectional search, biasing the forward tree toward the goal is equivalent to biasing the
tree toward the root of the backward tree. Since the backward tree grows out from the
goal, any branch in the backward tree can reach the goal. Given this, there are two
options for the biasing. If the forward tree is biased toward the goal, this would
encourage the trees to connect at the goal. If the forward tree is biased toward the
backward tree, this would encourage the trees to connect wherever it is possible. The
former strategy encourages shorter paths at the expense of prolonged computation, while
the latter encourages finding a path more quickly at the expense of a less optimal path.
Biasing the forward tree toward the backward tree is more consistent with the goal of
decreasing computation time, and thus it is the strategy used here. The algorithm for a
bidirectional search appears below as Algorithm 4.2. It is written with one forward tree
and one backward tree, but the change to multiple forward and backward trees is
straightforward. To change the algorithm to accommodate a pair of trees for each floor,
one need only place a \texttt{FOR} loop around the \texttt{WHILE} loop so that each pair of trees can be grown. The \texttt{STEP()} function is one iteration of the \texttt{BUILD_RRT(start, goal)} function from Algorithm 2.3.

\textbf{Algorithm 4.2 Bidirectional FOREST() Algorithm}

1. Tree forward, backward
2. Path path
3. \texttt{bool cross = FALSE}
4. \texttt{WHILE (!cross)}
5. $x_{new} = \texttt{forward.STEP()}$
6. \texttt{IF (backward.CROSSES(x_{new})）}
7. \hspace{1em} \texttt{cross = TRUE}
8. \texttt{ELSE}
9. \hspace{1em} $x_{new} = \texttt{backward.STEP()}$
10. \texttt{IF (forward.CROSSES(x_{new}))}
11. \hspace{1em} \texttt{cross = TRUE}
12. path = \texttt{RETURN PATH(forward, backward)}
13. \texttt{RETURN path}

\textbf{4.1.2 Measurement}

For the one-tree-per-floor algorithm, in which each tree is responsible for searching only one floor, all of the nodes in each of the trees are unioned together into one tree for the purposes of calculating coverage metrics. This is done only after each tree has been completed and all metrics except the coverage metrics have been calculated.
Adding all of the nodes from separate trees destroys the parent pointers, but this information is not necessary to calculate coverage metrics.

The coverage metrics were calculated using a regular grid of points on each floor, with one point at each pair \((x, y)\) where both \(x\) and \(y\) are multiples of 10, and in the range \([0, 200]\). The distance between each point in the grid, and the closest node in the tree to that point was calculated as a true Euclidean distance in the \(x-y\) space only ignoring the floor, or \(z\) component. This simplification causes no loss of information as long as there is at least one node on each floor (because if the node and the grid point are on the same floor, the \(z\) component is zero). In testing, each tree had at least one node on each floor.

### 4.1.3 Results & Analysis

As expected, splitting the search up between these sections helped immensely in the number of nodes generated, in the quality of paths generated, and especially in the time it took to generate the paths. The results of using a multiple-tree strategy showed significant improvement over a single-tree strategy. Using a single-tree strategy, the algorithm took an average of 8481 nodes and 30 seconds to find a path to the goal on the top floor from the center of the bottom floor. Using a forest with one tree on each floor, these results were reduced to 1555 nodes on average and under two seconds to calculate. These staggering performance increases show how much better a forest strategy is for this example.

The reason that splitting up the search helps so much is that the distinct sections of the search area are so separated. The only way to get from one floor to the next is to take the staircase. There is only one staircase on each floor (a square between \([180, 190]\) for \(x\) and \(y\) on even numbered floors and \([-180, -190]\) for \(x\) and \(y\) on odd numbered
floors). The staircases only allow the tree to move up to the next floor. They do not allow it to move down to the previous floor. There is also a progression of floors that must be followed. Once a path is found for a specific floor, there is no reason to continue searching that floor, and any additional nodes that might have been used to find additional paths on that floor would be better utilized finding a first path on a new floor. There is almost no value to finding additional paths; after a path is found on one floor that floor should no longer be searched. This is why the managed forest method works so well for the Stair Climber example. Once a path has been found on a particular floor, no more nodes are expanded on that floor. This way, new nodes are always concentrated where they are most useful, in places where there is not already a path. In the single tree example, some nodes are expanded in places where a solution has already been found, leading to additional nodes being expanded which are not useful.

It is because the example is composed of essentially distinct sections that solving each section individually makes a lot of sense when compared to solving the entire problem at once. By dividing the problem, four new problems are created which are each simpler than the original problem; and since the connections between the individual solutions are trivial, the solution to the original problem is easier to find.

Several solution strategies were compared on their ability to find solutions to this problem. In addition to division by direction of growth, the strategies can be divided into two other categories: single-tree strategies and multiple-tree strategies. The primary distinction between single-tree strategies and multiple-tree strategies is that trees in single-tree strategies must move between floors, while multiple-tree strategies have at least one tree dedicated to each floor, making movement between floors unnecessary for
individual trees. This explains why the bidirectional search with two trees is considered a single-tree strategy.

While using forward trees only, three configurations were tested. The first was a single tree, biased toward the goal on the fourth floor (1GB). This strategy appears in Figure 4.2. The second was a single tree, biased toward the transition region on the highest floor the tree had reached to that point (1FB). This strategy appears in Figure 4.3. The third was one tree on each floor, biased toward the transition region on that floor (MFB). This strategy appears in Figure 4.4.

In using a bidirectional search method, there were two strategies tested. The first (1BiDi) used two trees, one forward from the start and one backward from the goal. This strategy appears in Figure 4.5. The second (MBiDi) used two trees per floor, one forward tree starting at the entry point of the floor (the down staircase) and one backward tree starting at the exit point of that floor (the up staircase). This strategy appears in Figure 4.6. A table comparing the performance of all of these strategies appears at the end of this chapter as Table 4.1.

The first of the forward strategies performed the worst among those strategies. Because the tree did not have any real information about where to go until it got to the top floor, exploration before this floor was essentially random. In some cases, the staircase on that floor was in the same \((x, y)\) location as it was on the top floor, so the biasing toward the goal on the top floor helps. But in other cases, the staircase was on the opposite side of the floor, so the biasing was detrimental. In addition to this, all branches of the tree would be explored simultaneously regardless of the floor they are on, so the
bottom floors should be more fully developed than the top floors since a part of the tree has been on those floors longer.

It should be noted that biasing toward the goal on the top floor is not a bad idea. If there are many staircases on each floor, biasing toward the goal on the top floor encourages the tree to find the staircase closest to the goal on the top floor leading to shorter path lengths. In this set of experiments, there is only one staircase per floor.

Because the distance is calculated by finding the nearest node on the nearest floor and the tree starts on the bottom floor, at any given point the highest current floor (HCF) the tree has reached is the mostly likely floor to get a new node. This is because in addition to any $x_{rand}$ generated that falls on the HCF, any $x_{rand}$ generated on a floor above this one will also cause $x_{new}$ to be created on the HCF because it is the closest to all floors above it. The more floors above the HFC, the more exaggerated this effect will be.

When the HCF is the first floor, 100% of the new nodes are generated on the first floor, because it gets all of the nodes from any $x_{rand}$ generated on the 1$^{st}$, 2$^{nd}$, 3$^{rd}$, and 4$^{th}$ floors. When the HCF is the second floor, only 75% of the new nodes are on the second floor. The second floor gets all the nodes from any $x_{rand}$ generated on the 2$^{nd}$, 3$^{rd}$, and 4$^{th}$ floors, but any $x_{rand}$ on the first floor will cause the first floor section of the tree to be expanded. This continues if the HCF is the third floor, as it would only get 50% of the new nodes, those for the 3$^{rd}$ and 4$^{th}$ floors, while the second and first floors would get the other 50% (split evenly 25% each). Once the HCF is the highest possible floor, this effect disappears because there are no floors above the HCF, and each of the four floors gets an even portion of the new nodes.
Traits of all of these factors can be seen in Figure 4.2, a picture of the four floors of a single tree biased toward the goal exploring the Stair Climber problem. The first thing to notice is that once the tree reached the fourth floor, it had no problem reaching the goal. This is due to the top floor biasing. Once the tree reaches the top floor, it quickly finds the bias point it is searching for. The second thing to notice is the relative exploration on each of the other floors. The most well explored floor is the bottom floor, because it had the most time to grow, and as a result over the course of the algorithm, had the largest percentage of nodes expanded into it.

You would expect that the next most well explored floor would be the second, but that is not the case. This is because the transition from the second floor to the third floor is at the same $x$-$y$ location as the goal, on the top floor. Because of this, the tree was able to find this transition relatively quickly. This is evidenced by the almost direct path on the second floor from the entry point of the floor at the bottom left corner of the frame, to the transition region, at the top right corner of the frame. Being able to find the transition quickly, the tree grew from the second floor to the third floor relatively quickly, so the second floor was the HCF for only a short while, explaining why it is not as well explored as the $3^{rd}$ floor.

![Figure 4.2: Stair Climber Problem: 1GB Strategy. Floors 1-4 (from left to right).](image_url)
The second forward strategy, using a single tree biased toward the transition region of the HCF performed significantly better. This strategy gave the tree information about where to go on each floor, greatly improving the tree’s ability to find the transition region, and move up to a new floor. Since the algorithm is still growing the entire tree, nodes will still be expanded on floors below the HCF, with the same likelihood as before, but since the biasing is always on the HCF, each floor is solved more quickly, as was the second floor in the previous example.

Figure 4.3 shows a sample tree using this strategy. As you can see, each of the floors is less developed than those of the tree using the previous strategy. Also notice that on each floor, the path from the initial point to the goal point is almost straight, as it was on floor 2 of the previous example. This is the direct effect of the per floor bias. Since the tree knows where the transition region is, it can be found largely due to the biasing, which leads to a straighter path than would random exploration. Because this algorithm can find each staircase so much faster, there are many fewer nodes in this tree than in the previous one. This tree for example had only 2283 nodes, whereas the tree in the previous picture had 8613. This explains why the previous tree covered the search space so much better than this one does. Still, since it still grows from the bottom floor to the top floor, the bottom floors of this example are better explored than the top floors, because all floors the tree has reached are explored simultaneously, and each has at least a 25% chance of getting a new node at any iteration of the example. The bottom floors have had more opportunity to receive new nodes than the upper floors.
When using the third forward strategy, the one-tree-per-floor strategy, performance got even better. The path lengths were essentially the same, but because each floor was solved separately, and once a floor was solved, it was not explored any further, fewer nodes were expanded than with the 1FB strategy. Since each tree was contained in a single floor and its growth stopped when it reached the goal, the algorithm was able to avoid wasting nodes in previously solved sections. All new nodes would be expanded on a floor that did not have a solution; this led to a noticeable increase in performance. In the MFB strategy, trees averaged 1509 nodes, while in the 1FB strategy, trees averaged 2276 to find a solution. As a result of the smaller node count, the examples were faster to compute, the MFB trees taking an average of 1.15 seconds less to complete than the 1FB trees. Although it seems obvious that the MFB trees should take less time to compute by virtue of having fewer nodes, it still bears mentioning, because the strategy is different.

Since the MFB strategy uses four trees instead of only one, it is important to take into account the effort that it takes to link together the paths from each of the trees. This is an additional step required over the 1FB and 1GB strategies, strategies which each use only a single tree, so there is no linking cost. In this case, it is very simple to link the trees together, because the movement strategy is simple, and the goal of each tree is at the
same \((x, y)\) location as the initial point of the following tree. In general, linking two trees could be difficult, if the movement strategies were difficult or different, if there were any carryover effects (such as momentum), or if the two points to be linked are not at the same location. In this case none of these things (or anything else that would make linking difficult) is true, so linking these trees is trivial, and its contribution to the calculation time is negligible. A sample MFB tree can be seen in Figure 4.4.

![Figure 4.4: Stair Climber Problem: MFB Strategy. Floors 1-4 (from left to right).](image)

The results for the bidirectional strategies were more extreme. Bidirectional strategies were both the best and worst out of all of the strategies tested.

The 1BiDi strategy using a forward and a backward tree, biased toward each other, performed worst out of all of those that were tried. This strategy’s significant differences from the previous two single-tree strategies were the reason for worse performance. The most obvious difference is that there are two trees growing instead of one. One would expect that this would help the forest find the goal faster, because it can search forward and backward at the same time, allowing the forest to solve two parts of the search space at once. However, this did not help in practice because of the second major difference, the biasing.

In the previous two examples trees were biased toward a goal, whether it was the final goal on the top floor, or the intermediate goal of the staircase on the highest floor.
explored to that point. This biasing helped push the tree toward the goal and find it faster. In this two tree approach, the trees are biased towards each other by picking random nodes from the opposite tree as $x_{\text{rand}}$. By doing this, we encourage the trees to run into each other, the success condition. In this example this actually hurt the forest’s ability to find staircases, the primary obstacle of finding a solution path. Because these staircases are the only way to get from floor to floor, and all of them must be traversed to find a solution path, making the ability to find the staircases crucial to the algorithm’s success. This strategy was bad at finding the staircases because each tree was biased toward the opposite tree. In this case, this is only useful when the trees are on the same floor, because only then do they have a chance to intersect, and terminate the search.

When the trees are on different floors, this biasing is very harmful, as was the biasing toward the top floor in the 1GB strategy on the 3rd floor. Here, this harmful biasing is happening on every floor.

Since biasing towards the opposite tree is accomplished by picking a random node from the opposite tree as $x_{\text{rand}}$, the only time the trees are biased toward a transition region is during the first few iterations. At this time, the opposite tree is confined to a relatively small $x$-$y$ space around the transition region it originates from, making biasing meaningful. Once the tree has explored the search space to any meaningful degree, the biasing toward the opposite tree is meaningless, because the opposite tree has explored its $x$-$y$ space fully. Therefore, in the $x$-$y$ space, picking nodes from the opposite tree is similar to picking random nodes, essentially eliminating the effect of biasing. This explains why there is no performance gain from this type of biasing. In the floor dimension, biasing is still somewhat useful, but its contribution is negligible, because it is
similar to the HCF effect discussed earlier. The HCF effect is more powerful, because
the biasing is so infrequent. Because this biasing is so ineffective, it makes this method
similar to an unbiased search.

Figure 4.5 shows a picture of the 1BiDi strategy. Notice how both the forward
and backward tree are well developed on each floor that they reach (1 and 2 for the
forward tree, and 4 and 3 for the backward tree). Also notice that once the two trees have
reached the same floor (which in this case happens when the forward tree reaches the
third floor), there is little growth required for them to cross and the algorithm to
terminate.

![Figure 4.5: Stair Climber Problem: 1BiDi Strategy. Floors 1-4 (from left to right).](image)

In contrast to the poor results of the 1BiDi strategy, the MBiDi strategy
outperformed all other strategies in terms of both node count and generation time. Using
a pair of trees on every floor instead of one tree on each floor showed improvement in the
number of nodes generated. The MBiDi strategy used 1266 nodes on average, while the
MFB strategy used 1510. As one might expect, this example also ran faster, taking an
average of almost a second less than the MFB strategy. However, the improvement from
using a bidirectional strategy was not as great as the difference from using separate trees
for each floor instead of a single tree. This indicates that the primary performance
increase was due to the restricting of the search to specific areas and not the ability to
search both forward and backward. In every coverage metric, the forward search yielded slightly better results than the bidirectional approach. This is not at all surprising because the forward trees had more nodes than the bidirectional trees.

The reason that the bidirectional search found the goal slightly faster than the forward search could be that in the bidirectional search trees were biased toward the opposite trees, instead of toward a specific point. The biasing in bidirectional searches is less direct than that in the forward only searches, because it is biasing toward a larger area (the entire opposite tree instead of just a goal point). Because of this, each tree had more points at which it could be successful. With a bidirectional search, anytime the trees cross, they are successful, and this can happen anywhere on the floor. With a forward only search, there is only one success point, so naturally, it takes longer to find it than for the two trees (which are growing toward each other) to cross. However, this could lead to inferior paths, because paths which grow away from the optimal path are just as likely to reach the opposite tree as those which grow most directly toward the goal, if such paths exist in the opposite tree. This is evidenced by the fact that the MBiDi method consistently returned longer path lengths than did the MFB strategy. A picture of the MBiDi strategy can be seen in Figure 4.6. In each case, the backward tree grows out from the blue rectangle.

![Figure 4.6: Stair Climber Problem: MBiDi Strategy. Floors 1-4 (from left to right).](image)
A possible reason why the MBiDi search is only somewhat more successful than the MFB search could be that in this problem, a branch of the forward tree can never be dead (i.e., be unable to reach the goal). The goal is reachable from everywhere in the search space under all conditions. A backward search is normally useful because it helps to avoid dead paths as the tree gets closer to the goal. Here there are no such paths to avoid, so it is not surprising that the backward search does not help much here.

Results for all of the above examples can be seen in Table 4.1. Particularly noteworthy is the large difference in time values. When looking at the results of the tests, they can be divided into two groups: the fast tests, and the slow tests. The slow tests included the 1GB strategy where the algorithm took an average of 50 seconds to find the goal, and the 1BiDi strategy where it took 106 seconds. Although the second of these strategies took twice as long as the first, they are of similar order of magnitude. Compare these, however, to the fast tests, the 1FB strategy and both multiple-tree strategies (MFB and MBiDi). The one-tree-per-floor bias strategy took an average of only 2.84 seconds to find a path to the goal. The two multi-tree strategies were even faster, taking an average of 1.69 and .75 seconds, respectively. Since these times differ by orders of magnitude from each other, the smaller times do not show up well on the chart. It appears that the two multi-tree strategies are not even on the chart. These strategies did not run instantaneously, they just ran much faster than the slow single-tree strategies.

Table 4.2 shows a close-up of the range from 0 to 5 of the metrics for the various trees. In Table 4.2, the measure for time is in seconds, to better show the performance of the one-tree-per-floor strategies.
Table 4.1: Performance Metrics for the Various Trees.

Table 4.2: Performance Metrics for the Various Trees, Close-up from 0-5.
Chapter 5

Receding Horizon

[In this chapter, the word “rover” will be used to generically refer to the object following the path.]

In some situations, the search conditions of the problem will be different than the model used to find the path. This could happen for many reasons. For example, in representing the real world system, some inaccurate approximations might appear in the computer model. If these inaccuracies appear in a quantity that affects the growth of the tree, then the path grown by the model will differ from the one actually traced out in the real world.

Another reason could be that the conditions of the space change, and are not known when the model is developed. An example of this type of condition would be the location or number of obstacles. Here, the path will not differ (its steps will lead to the same locations), but its validity might, if an obstacle crosses the path. In situations such as these, a receding horizon search can be helpful [9].

In a receding horizon search, a path is planned in the same way as in previous searches (following Algorithm 2.3). Then this path is followed, and checked at each step for correctness. Any time the path is invalidated by the problem conditions, the path is replanned. The replanning must allow the rover to advance far enough along the invalidated path to give the algorithm time to plan a new path. For this reason, steps are generally long enough that the path can be replanned in a single step. Replanning is done using the same planning model that was used to plan the original path. The same model is used because it is not usually possible to discover the real model. Even if it is possible,
discovering it would likely require more resources than replanning in the original planning model. Replanning in the planning model allows the rover to continue on, instead of being stuck and unsuccessful.

Because the two models are different, a path cannot be generated in the planning model. What is generated in the planning model is a control sequence of inputs for the rover to follow. The rover follows these inputs in the real system, and thus traces out a path.

For example, if the rover were a person walking from one building to another, the sequence might read “Exit building; turn left; take 200 steps; turn right, take 40 steps, turn right, take 10 steps, turn left; enter building.” These directions would probably work if the person who generated it takes steps of the same length as the person who is to follow them, and if the streets are clean, dry, and free of traffic. However, if the person blindly followed these directions in a real world situation, she is likely to bump into someone, end up at the wrong location, or be hit by a car. What is more likely to happen is that the person following the directions will use them as a starting point, and make adjustments along the way. For that reason, their actual path will be different from the path dictated by the control sequence.

Similarly, the rover first plans an initial path from start to goal, and then it begins following this path. Along the way conditions may change, and if the path is invalidated, it must be replanned in real time. This is an important point. The rover does not stop to replan the path; it continues to travel the path it has already generated while it attempts to find a path to avoid the obstacle or correct the invalidation. This tactic is useful, especially in situations where conditions are changing, because if the path is invalidated,
it can be replanned without compromising the movement of the rover (i.e., it does not have to wait). This also handles the situation where problem dynamics do not allow the rover to stop midway.

The primary benefit of generating a path in this fashion is that there is no need to anticipate everything that will happen along the path beforehand. An initial path can be generated, and the algorithm can use this path until it is invalidated, at which point a new path can be planned to correct the invalidation. That way, the only waiting required is during the generation of the initial path. Depending on the changing conditions and how they arise, this could be a large savings when compared to taking everything into account at the beginning. If all of the problem conditions are not known beforehand, then this is the only way to proceed. The only requirement is that it be able to generate a path around the obstacle in less time than it takes to travel to the obstacle.

5.1 Background

In testing the receding horizon search, two problem models are used, to simulate the difference between planning a path and following the planned path. The planning system is the system used to generate a control sequence which is expected to find the goal, and the real system is the system in which the actual path is generated. The planned control sequence will sometimes be referred to as the planning path. The actual path is generated in the real system using the control sequence provided by the planning path. To simulate the difference between these systems in practice, the two systems must differ in some meaningful aspect. The planning system is built using the best information known about the real system beforehand. The real system simulates a perfect model of the real world.
Once a path to the goal is found, it is extracted from the tree so that the only nodes remaining in the tree are those in the planning path. After the path is extracted, the algorithm follows the control sequence indicated by this path under the real system, in an effort to reach the goal. This is done one step at a time, with each step being the same distance (measured in time) as it is in the initially generated path. After each step, the path is checked to see if any of the conditions of the problem have changed. If they have not, then the algorithm continues following the planning path. If, however, the problem conditions have changed, then the algorithm checks to see if the changed conditions have ruined its planning path.

In cases where a step lands the algorithm somewhere other than the planned location, the path is always ruined. If a change in an obstacle occurs, the path is only ruined if the newly changed obstacle intersects the current path. If the current path is invalidated, a new path to the goal is generated from the current location, using the planning system. Once a new path to the goal is generated, the rover follows the new path, as it did with the initial path. This behavior continues until the rover reaches the goal, at which point the algorithm is said to be successful.

### 5.1.1 Types of Invalidations

There are two types of invalidations that will be considered in this chapter:

*obstacle invalidations* and *path invalidations*.

**Obstacle Invalidation**

An obstacle invalidation occurs when the path is invalidated by a new or changed obstacle in the search space. This type of invalidation causes specific steps of the path to be invalidated, because they intersect an obstacle. When pruning due to an obstacle
invalidation, each step of the path that is affected by the changed obstacle must be pruned. Affected steps are those with which the obstacle intersects.

**Path Invalidation**

A path invalidation occurs due to an inconsistency between the planning model and the real model. This happens when the rover takes a step and its location after taking that step does not match the location predicted by the planning path. When pruning due to a path invalidation, a new path must be created from the new location. The remaining nodes in the path may be pruned, or may remain, depending on the pruning algorithm implemented. Nodes in the region where the invalidation occurred are likely to be pruned, because the dynamics in that region have already caused a path invalidation. Here, a *region* is defined as a discrete state in the hybrid automata of the problem, which defines a specific set of movement dynamics for the rover. Pruning algorithms will be discussed in Section 5.2.

**5.1.2 Algorithm**

In general, the Receding Horizon search algorithm `BUILD_RRT(start, goal)` function is similar to Algorithm 2.3. There are two differences. First, in this version, the function accepts a variable. The variable accepted by the function indicates the start point of the search. This variable was implicit in Algorithm 2.3, but because the starting point of the search was always the same it did not need to be explicitly stated.

If the node is already in the tree, then the search happens from this node forward. Nodes before the start point remain in the tree, but are ignored by the `NEAREST_NEIGHBOR(x_rand)` function. They are merely kept around for the continuity of the path (i.e., so that the tree shows the path the rover has traveled when it reaches the goal).
Nodes after the start point of the search must be pruned from the tree, according to one of the strategies detailed in Section 5.2 below.

If the node is not already in the tree, then it becomes the initial node, and the search takes place from here forward. This is the case of the original Algorithm 2.3. The origin of the search is a node that is added to the tree as the initial node, and thus it fits this model. This also can happen due to a path invalidation, because the rover has landed in a location not in the original path. This allows the same algorithm to serve for both the initial planning tree and the subsequent replanning.

The second difference is that the tree must be pruned before the \texttt{BUILD\textsc{RRT}}(\texttt{start, goal}) function. This was not necessary in previous chapters because the search space never changed. Once a path from start to goal was found, it was sufficient to be a solution. In this chapter, however, the search space may change, and the first path generated may not solve the problem. Because of this difference, the tree must be pruned. How the pruning is done depends on the replanning strategy employed. As mentioned earlier, replanning strategies will be discussed in Section 5.2 below.

The receding horizon search is given in Algorithm 5.1 below.
Algorithm 5.1: Receding Horizon Search for Different Models

1. Tree.BUILD(RRT(start, goal))
2. int currentIndex = 0;
3. Node currentNode;
4. WHILE (!Tree.REACHEDGOAL(goal))
5. currentNode = Rover.TAKESTEP();
6. currentIndex++;
7. IF Tree.PATHISINVALID()
8. Tree.INSERT(currentNode, currentIndex)
9. Tree.PRUNE()
10. Tree.BUILD(RRT(currentNode, goal))
11. RETURN Tree

5.2 Specific Pruning Strategies

Many strategies can be used for the pruning in this type of search. A few examples are: scrapping the remainder of the tree and starting over; saving parts of the original path; avoiding the new problem area. There are various advantages and disadvantages to each of these strategies, but they all share one thing in common. They find problems in the path created by the planning system, and help the rover avoid them.

The difference between these strategies is how each handles an invalidation of the planned path. Each of these strategies prunes nodes from the previously found path differently. In the first and simplest strategy, no nodes are preserved. In this strategy, all of the remaining nodes are removed from the tree on an invalidation. In the second strategy, those parts of the path which were not invalidated by the changed conditions are
preserved. In this strategy, only those nodes which are invalidated are pruned. In the third strategy, parts are preserved based on what region they are in. In this strategy, any region in which an invalidation occurs causes all parts of the path in that region to be pruned. In all other respects these strategies are the same. Specifically, the path is always planned according to Algorithm 5.1. The path is checked at each step to see if any of the changing conditions have caused an invalidation. Success is determined by checking whether the rover is in the goal region.

The path in Figure 5.1 will help illustrate how each of these strategies deals with an invalidation. For each strategy, the path will be invalidated and pruned according to that strategy. This will give a good visual demonstration of what is happening with each of these strategies.

![Figure 5.1: An Example Path to be Invalidated Later.](image-url)
5.2.1 Scrap-the-Remainder Strategy

The simplest strategy for dealing with invalidations of the path is to prune all future nodes in the tree, removing them from the path, and start replanning from the current location. The benefit of this strategy lies in its simplicity. This strategy applies all of its computational muscle to path planning, and none to trying to reduce its workload. This strategy is also the only one that works in all situations. The other strategies require additional knowledge about the problem to do their creative replanning. This one needs only the planning system, and a method for knowing when its paths are no longer valid.

The disadvantage of this method is that it uses brute force to find its replanned solutions. If only a small part of the path is invalidated, or if several large parts remain valid, then this strategy will duplicate much of the work it has already done by replanning the valid segments of the path.

In Figure 5.2, the area enclosed by the thick red line represents a tree that has fallen in the search space, invalidating all nodes in that region. This strategy finds the first node in the path that has been invalidated, and removes it, and all nodes that come after it. Although this costs the tree part of a path that it may have been able to connect to later, it makes the pruning step faster than the other strategies, which gives it more time to refind that section of the path.
5.2.2 Saving Parts Strategy

This strategy saves parts of the tree and does not prune the remainder of the path when an invalidation is found. Instead, it prunes only what is necessary to avoid the invalidation, and replans only what is necessary to create a new path to the goal. This is generally helpful with obstacle invalidations, because they invalidate specific nodes.

To benefit from doing this, the path must potentially be able to connect the part before the invalidation with the part after the invalidation. If the only coordinates that must be matched are positions, and if movement is not restricted by any strict dynamics, then it is straightforward to connect to a previous path. If, however, velocities and accelerations, or other more complicated coordinates must be matched, then connecting these paths becomes more difficult because there are more coordinates that must be matched.

The advantage of this strategy is that it potentially leverages the work that has already been done. A strategy like this can save parts of the path which have not been invalidated. Because of this, the new tree does not have to generate the entire path, only the part that has been invalidated. This may allow the new tree to find a path more quickly than if it had to replan the entire path.
A disadvantage of this strategy is that it may save parts of the tree which will be invalidated later. This can cause the newly generated path to rely on pieces of the path that were generated without taking into account the new information that invalidated the old path. This raises the potential for wasted computation. For example, in Figure 5.3, a moving obstacle first invalidates one node in the path (Invalidation 1) and later invalidates another node in the original path (Invalidation 2). In this case, computation may be wasted if both nodes are not thrown away immediately, which may be possible if the motion of the obstacle is predictable. If the path generated because of Invalidation 1 links with Invalidation 2, the path will have to be replanned again when Invalidation 2 is discovered. This is especially detrimental if a suboptimal route is generated in order to save computation by linking with the soon-to-be-invalidated segment.

There are ways to mitigate this, but those may also incur additional computation. For example, when a segment of a path is invalidated by a moving obstacle, the invalidation engine can check to see if the moving obstacle will later invalidate any other segments of a path. Then these segments can be thrown out as well, and the new path can avoid segments which will be invalidated later. To be able to do this, the algorithm must be able to predict the obstacle’s movement, which may not be trivial. The remaining segments might still be invalidated later by other obstacles that have not been introduced yet, but any new path also has this problem.

In Figure 5.3, the blurry red line indicates the path of an obstacle, and the red open circles highlight places where the obstacle crosses the path in both space and time. In this strategy, only those nodes which the obstacle would have hit are pruned from the path, and all others are left in the hopes that the path can connect to them later.
5.2.3 Saving Regions Strategy

A variation on this approach is to prune parts of the path in regions which cause invalidation. This strategy can be applied in response to path invalidations. This strategy is beneficial for path invalidations because they generally occur due to incorrect modeling of movement dynamics defined for a region.

If the regions are clearly separated, and known to the search beforehand, then the pruning strategy can be adjusted when an invalidation occurs. Instead of just pruning the segment that was invalidated, and keeping the rest of the tree, the region where the invalidation occurred can be pruned since it is known that the path is invalid (i.e., in this region, following the control sequence the path indicates will not lead the rover to the locations the path describes).

When pruning an entire region, a few differences will be seen in the generation of a successful path. First, since more of the path is being thrown away, each replanning may take longer, since there is potentially more distance to cover. Also, while the rover is in the region causing invalidations, replanning will be necessary after every step, since in this region the planning system is not the same as the real system. Since the parts outside of the bad region are being saved, this strategy will encourage the path to get out
of the bad region as soon as possible. This saves computation because once it is outside of this region, replanning may not be necessary after every step.

In Figure 5.4, the red rectangle represents a region where path invalidations occur. Upon taking the first step into this region, the rover discovers that its path is no good here. With this knowledge, the rover prunes all nodes inside this region. This is different than the previous strategy, because in that strategy, only nodes that were invalidated were pruned. Here, all nodes that are inside the bad region are assumed to be invalid, although the validity of each node is not tested.

![Figure 5.4: Pruning Regions Strategy, Before and After Pruning.](image)

### 5.3 Avoiding Obstacles: Helicopter with Projectiles

In some problems, all of the obstacles are not known in the initial planning phase. If these problems are solved with an RRT similar to the kind generated by Algorithm 2.3, obstacles appearing or arising after the planning phase could invalidate the path while the rover is following it. This makes the traditional RRT inadequate for this type of problem. However, conditions change during execution frequently in the real world, meaning this type of problem is still interesting and its solutions still useful. The Helicopter example from Section 2.4.5 is a problem of this type.
The Helicopter example deals with a helicopter flying through space and trying to avoid projectiles which are being fired at it. In this example, projectiles are shot at a helicopter as it flies toward its goal. The projectiles are moving, and more of them appear as the helicopter continues to fly. This is exactly the kind of problem a receding horizon search is designed to solve. In this problem, the helicopter finds its path, and then, as it is flying, it must continually adjust its plan to avoid the new projectiles being shot at it.

5.3.3 Testing Strategy

In order for this receding horizon strategy to work, the helicopter must be able to accomplish two things while taking each step in the path. First, it must be able to determine whether or not the current path is still valid. This is necessary so that the helicopter knows whether or not to continue on its current path. If the current path is still valid, then nothing further is required this step. If it is not, then the helicopter must also be able to plan a new path before finishing the current step.

In this example, each flying segment takes the helicopter one (simulated) second to fly. This meant that the path validation and replanning needed to take less than a second at each step. This is important, because as long as the algorithm can plan a new path to the goal in less time than it takes to take a step in the old path, then the algorithm can successfully find a path in real time without having prior information of the changing conditions, or when and how they change.

Despite the ability to replan quickly, a new path is only planned when necessary. At each step, the helicopter takes inventory of all of the projectiles in the space, and all of their future positions in four dimensions (spatial dimensions x, y, z, and the time
dimension \( t \), i.e., the \textit{configuration space-time} [7, 17]. If it finds a location in the future where its path and a projectile intersect, then it decides to replan the path. If no such intersection is found, then the helicopter simply takes the next step in its current path. It should be noted that because the helicopter keeps track of the time dimension it is possible for a projectile to cross the path in space without replanning being necessary.

When replanning is necessary, however, all nodes after the first invalidation in the current tree are thrown away, and a new tree is planned using the same algorithm that was used to generate the previous tree. This time the algorithm starts planning from the next step in the tree, and plans a path around and through all of the projectiles in the space. The replanning takes place while the helicopter follows the current step in the current tree, and must be finished by the time it has to take the next step. This is a version of the scrap-the-remainder strategy, from Section 5.2.1.

\textbf{5.3.4 Results}

The helicopter was always able to avoid the projectiles being fired at it, despite the fact that the helicopter moved much slower than the projectiles. This is a testament to the receding horizon search’s ability to negotiate projectiles which were unknown at planning time. Even though all of the projectiles appeared after the helicopter started flying, none of them was able to hit it. This was not due to inaccurate projectiles. The fact that replannings did occur shows that some of the projectiles were on target when they were fired.

In 145 trips from the origin at (-400, 0, 0) to the goal at (400, 0, 0), the helicopter was always able to safely navigate the field of projectiles without being struck by one. In the 145 trips, there was an average of 124.5 projectiles per trip, requiring an average of
just under 42.5 replannings per trip to avoid them. The average final path length (the
number of steps the helicopter actually flew) was 51.8 steps of 1 second. The biasing
period for the search was 4.

The number of replannings necessary on each trip indicates that many of the
projectiles fired were accurate enough to hit the helicopter were it not for the receding
horizon search. The helicopter, however, was able to route itself around the projectiles
because of the search. In no case was a projectile ever able to hit the helicopter. This
shows that the search allowed the helicopter to safely navigate this dangerous airspace.

Out of 6164 total plannings, there were 57 cases in which the tree reached the
maximum solution size of 2500 nodes before finding a path to the goal. Of these 57
maximum solutions sizes, only 5 of them took longer than a single step (one second).
Among the group that reached maximum size in less than a second, only three were
unable to find a path to the goal in the time that remained in the step.

There were 8 cases where the algorithm failed to find a path in a single step (5
where the first planning took more than one step plus 3 where the failed first planning
plus the second planning took more than one step). In these 8 instances, it was able to
find a path within the first tenth of a second of the next step, and thus would have
required two steps.

It should also be noted that the algorithm never failed to find a new path to the
goal on consecutive attempts, indicating that the problem was never too hard to be solved
by the RRT. This shows that the algorithm was almost always able to replan a path when
its current path is invalidated by a projectile in only a single timestep. In cases where it
was not able to do so, one additional timestep was required. Since the projectiles were
detected immediately, and none was fired from less than two steps away from the helicopter, this is acceptable in this case. If it was not acceptable for the algorithm to take two steps to replan if necessary, one could reduce the maximum tree size. It will be shown later that both the median and average tree size of a successful path was much smaller than the maximum. With a smaller maximum tree size, most of the solutions would still be found. A smaller maximum solution size would also allow replanning to happen more quickly, as the later nodes are more difficult to plan than the early nodes. By removing nodes from the maximum tree size, the run time of the Nearest Neighbor search (the computationally expensive part of the algorithm) reduces proportionally to $n$ (on each lookup) and $n^2$ (cumulatively), where $n$ is the number of nodes for a maximum size tree.

Another method to achieve faster replannings might be to apply the knowledge gained in Chapter 3. In this case, by using 2500 nodes to plan two consecutive trees of 1250 nodes each, replanning would happen more quickly, even if both trees reached maximum size. In addition, the helicopter would have two chances to find a new path, instead of only one.

The average tree size in the replanning was 116.6 nodes, with a median tree size of 62 nodes. For this example, replannings were frequent, but happened quickly, which is good for a receding horizon search. The benefit of the search is its ability to replan, so the fact that replannings were frequent allows the search to leverage its benefit. On each complete trip from the origin to the goal, the algorithm had to replan at least 33 times, and on average, it had to replan 42 times. The fact that replannings were quick helped this example usually meet the requirement of replanning in a single step. The average
replanning time was .0202 seconds, which is well under the 1 second allotted for replanning. This is one of the reasons that the helicopter was so successful using this strategy. The problem set out for it happened to work quite well with the type of search implemented.

Replanning was more likely to fail as the helicopter got closer to the goal. This was because as the helicopter got closer to the goal, it had less space between itself and the newly appearing projectiles that caused it to replan, making navigating around them more difficult. The number of unsuccessful trees that happened after each number of previous replannings appears in Table 5.1.

<table>
<thead>
<tr>
<th>Number of Failures at Each Replanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Failures</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Number of Failures at Each Replanning.

There is an especially high concentration of failures around the 30th replanning. This is because all of the trees had to replan at least 30 times, but many did not need to replan 40 times. This means that at the 30th replanning, many trees were already close to the goal. The number of failures drops off after the 40th replanning, as trees reach the goal. This is because some trees have finished by this point, and so there are fewer opportunities for failure. Table 5.2 better illustrates the correlation between approaching
the goal and having trees that fail by showing the % of trees that failed at each replanning. For each bar in Table 5.2 only those trees that needed to replan that many times before finishing are included.

![Percentage of Failures at Each Replanning](chart)

Table 5.2: Percentage Likelihood of Failure at Each Replanning.

In Table 5.2 we can see that the more times a tree has to replan, the higher the likelihood that each replanning will fail. At earlier replannings, there are very few failures among many trees. At later replannings, there are more failures. This happens because as the helicopter approaches the goal, the projectiles are closer to the helicopter when they appear, giving the helicopter less space to plan around them.

Another interesting point that can be gleaned from Table 5.2 is that even in the situations most prone to failure, the probability of failure is still low. The percentage of trees that failed at a given replanning was only higher than 10% once. This happened at the 55th replanning, and only 9 trees needed to replan 55 or more times. In all other cases, the probability of a failure was less than 4%. This shows that the RRT was quite capable of solving this problem.
5.4 Path Correction: Hovercraft with Disturbances

Receding horizon search is also useful for correcting paths that have gone awry due to path invalidations. A path can be said to have failed if following the control sequence dictated by it causes the rover to end up in a different location than expected. This indicates that the conditions assumed by the generation algorithm were inaccurate. If this happens, a receding horizon search can be implemented to find a path to the goal point.

The Hovercraft example of Section 2.4.3 can easily be adapted for a receding horizon search because it has two regions, and the rules are different in each. The Hovercraft example has a region in which there is wind, and a region in which there is not. If the wind were inaccurately modeled, it would be a good reason to use a receding horizon search. In this adaptation of the Hovercraft example, the real system was different from the planning system in the way the wind works. Here, the planning system is identical to the system described in Section 2.4.3, but the real system is different in that the coefficient $B$ for the force $f_{x_{air}}$ is .02 (it was .01 in the planning model).

5.4.1 Testing Strategy

The receding horizon search used here is the one discussed above in Algorithm 5.1, using a scrap-the-remainder pruning strategy. The step size here is 1 time unit, the biasing factor is 10, and the maximum tree size is 10000.

After each step, the location coordinates $x_1$, $x_2$, $\theta$, $v_1$, $v_2$ and $\omega$ are compared against those in the initial path. If the coordinates generated exactly match those that the path predicted, the hovercraft moves on to the next control input. If, however, the coordinates differ, then the rest of the path is scrapped, and a new path is generated. This
behavior continues until the hovercraft reaches the goal, at which point the algorithm is said to be successful.

5.4.2 Results

In testing of the Hovercraft example with a receding horizon search, the trees were able to find the goal with great regularity. The reason for this is that when this algorithm detects a problem in the path it has chosen, or is unable to find the goal, it generates a new tree. By doing this, it is exploiting the information learned in Chapter 3. For this example, restarting a tree is a good way to find a path to the goal. In fact, because growth to the maximum size does not immediately mean failure, there were only 24 cases in which the tree was not able to find the goal out of 350, a success rate of just over 93%.

Receding horizon search was shown to work well to correct an erroneous path, as well as to produce a slightly shorter path. The path that resulted from the receding horizon search was actually about a step and a half shorter (26.80 steps) than the originally generated path (28.27 steps). This shows that the path length does not change much with replanning. This makes sense because even as the algorithm is taking steps and ending up in different locations than intended, it is still solving essentially the same problem, and, because the difference between the two models is small, it is still making progress toward the goal. This is not a great enough distance to indicate that the receding horizon method always finds a better path, and in some of the tests, the initial path was shorter than the final path. Also, because of this receding horizon search, the craft is still able to reach its goal. This otherwise would not have been possible under the real model, because there is no way to plan in the real system.
This example, however, did not pass the primary requirement necessary for a working receding horizon search. Here the replanning took too long. Each time a path had to be replanned, it took an average of 4.8 seconds to replan the path, at an average size of 2616 nodes. While this might not seem long in real time, each of these segments is only 1 second long in flight time, so by the time a new path is found, the craft could potentially be much further along a wrong path, and might not be able to correct itself from its new current location.

One way to solve this problem would be to make each of the steps longer, to give the algorithm more time to replan. Allowing the hovercraft to take longer steps could allow it to get much further off the intended location then before, but if the algorithm can plan a new path in a single step, then it should be able to correct.

Another way to ensure that the replanning does not take too long could be to restrict the maximum size of the trees. This would force the algorithm to find solutions faster, or restart the tree. This could have a negative effect on the success rate of finding paths, which would require more replanning. That might cause the same problem if many replannings are required for each step.

Making each of the steps twice as long did not show favorable results as a solution to the problem of lengthy replan times. The high speed of the hovercraft caused it to have trouble stopping at the goal, flying past the goal frequently. These longer steps also caused the hovercraft to make turns at sharp angles. This is allowed by the model, but might not be possible in the real world. An example of such movement appears in Figure 5.5, highlighted with a yellow circle.
It is likely that this could be solved by taking more diligence in constructing the model, and more realistically reproducing the physics of the craft. However, this would require more calculation to generate paths, causing the algorithm to take longer to find a path and exacerbating the problem the more diligent and realistic model was trying to solve. Even if it were possible to implement a more realistic model solving the above mentioned problems without taking longer to calculate, the results for the algorithm with these longer steps showed that it still took longer to find a path than it did to take a step.

![Figure 5.5: Unrealistic Hovercraft Movement](image)

Using a smaller tree also did not show favorable results. Although these trees were calculated faster than both of the previous trees, it was still not able to terminate in the time it took to take a step. With a tree size of just 2500, compared with 10000 for the previous two examples, the algorithm still took 3.2 seconds to plan new paths on average.
Although the maximum tree size was a quarter of the previous size, most of the solutions were only two fifths smaller than the previous size. This explains why, despite the maximum tree size being much smaller, the planning times were not much smaller. The average tree size for these examples was 1602 nodes, much closer to the previous maximum of 2616 nodes.

All of this indicates that this problem cannot replan fast enough to find a path to the goal in a single timestep. The dynamics of this problem are complicated in that there are six coordinates to keep track of, and a complicated set of equations that govern their relationships. This was likely the reason that replanning took so much longer here than it did for the helicopter. The Helicopter example had simple movement dynamics. It always moved at the same speed, and could only choose one of five possible movements, restricted to three spatial dimensions. The Hovercraft on the other hand, had infinite possible inputs, a range of speeds to move at (of which we tested 10 at random), and six spatial dimensions of movement. The complications from having to plan paths with these dynamics made the Hovercraft take too long to replan.

This problem can indeed be solved with a receding horizon search, but it requires some modification to the search algorithm. The hovercraft would have to plan as much of the path as possible in the time it had to replan. Then it would find the point in the tree it has grown that is closest to the goal. The hovercraft would treat this point as the goal temporarily, and prune from its tree everything but the path to this point. The hovercraft would then start following the path to this point. While it is following this path, it would continue to plan the rest of the path to the goal [9].
Chapter 6

Conclusion

6.1 Summary

In this thesis, we showed through several examples that the Rapidly-exploring Random Tree (RRT) can successfully be used to plan for and verify problems modeled as hybrid systems. We also identified some problems that the RRT may run into, and proposed solutions to mitigate their effects on the RRT’s ability to plan paths.

In Chapter 3, we discussed the phenomenon of heavy-tailed behavior. We reviewed methods to determine if an example demonstrates heavy-tailed behavior. We showed that a regular RRT can sometimes fail to find a path for a heavy-tailed example. We also showed that several small trees all rooted at the same initial point have a greater likelihood of success than a single large tree. In addition, we showed that a similar effect can be achieved by restarting a tree that has been growing for too long. Finally, we demonstrated that this technique is detrimental for problems which do not display a heavy tail.

In Chapter 4, we introduced the idea of using a forest of trees to solve a problem instead of a single tree. Here, we suggested that by using several trees, each tasked with solving a part of the problem, and then uniting the partial solutions they produce, a path through the search space can be found more rapidly. We showed that this approach, when compared with a regular RRT approach, generates fewer nodes, and takes less computation time. We also showed that using a bidirectional search can speed this up even further, but that it is important that the biasing not hinder the search.
In Chapter 5, we explored receding horizon search. We showed that this search can sometimes solve problems that a regular RRT would be unable to solve. Specifically, receding horizon search can account for a changing search space, while a regular RRT must have complete knowledge of the search space before planning its path. This was shown to have utility for avoiding obstacles in a search space. It was also shown that a receding horizon search works better with simple movement dynamics, due to timing restrictions on the algorithm.

6.2 Future Work

In the course of the work that produced this thesis, several ideas arose that would benefit from further investigation. They are detailed below.

6.2.1 Step Size Determination

In the course of this thesis, it was usually shown that larger step sizes performed better. This however is not an absolute truth. The reason that larger step sizes performed better in this thesis is because no testing was done to try to find the ideal step size. For future work, someone could increase step sizes incrementally to find the point at which larger sizes begin to hinder the performance of the tree.

6.2.2 Biasing Frequency

Biasing factors (e.g., to the goal) were also not examined in much depth. In general, once a biasing factor was picked, it was the only one used, to prevent that factor from affecting the data. Biasing factors and strategies could be examined in more depth to determine the best ways to bias effectively. The goal could be to improve the odds of
finding a solution, to increase coverage, to reduce the number of nodes, or all three of these.

6.2.3 Goal Point Convergence

Frequently in this thesis, the qualification for reaching the goal was that the tree reaches within some bounding region around the goal point. To establish a more perfect connection to the goal point, the idea of breaking the search into sections discussed in Chapter 4 could be adapted. Instead of breaking the tree into roughly equal sections as in Chapter 4, the original search method could be used to find a path to the goal region around the goal point, as before. We could then start an entirely new search limited to the goal region using the same RRT method. The only changes would be that the new initial point would be the point inside the goal region that the original tree had reached, and we would use a much smaller step size to ensure that we stay within the goal region. If necessary, this process could be repeated several times until a sufficiently close match to the goal is obtained.

6.2.4 Negative Bias

If it is determined that a legal region has undesirable properties, and we would prefer to find a path that does not include nodes in this region, we could negatively bias that region, in order to select points from that region with a lower frequency than would random selection. For example, for the Hovercraft example in Chapter 5, we could negatively bias the windy region while replanning, in an attempt to avoid the extra work that it requires in replanning.
6.2.5 Model Refinement

Chapter 5 talked about using an incorrect model to plan a path for a correct model. It claimed that it was not possible to discover the real model because it would require more resources than replanning. Perhaps it would be possible to discover the real model by incrementally refining the path after each invalidation until the planning system was sufficiently close to the real system. This would help in situations where the model was inaccurate about movement, but would not help for obstacles, which are outside of the control of the rover.
Bibliography


