Motion Planning and Control using RRTs

Submitted in partial fulfillment of the requirements for the degree of Master of Computer Science

Michael M. Curtiss
Electrical Engineering and Computer Science
Case Western Reserve University
mmc18@po.cwru.edu
Project Advisor: Prof. Michael S. Branicky
msb11@po.cwru.edu
May, 2002

Abstract

In this project, we used Rapidly-Exploring Random Trees (RRTs) to make advances in motion planning for several domains. In order to gauge the strengths and weaknesses of RRTs, we used the RRT algorithm to solve standard problems in machine learning and control. We also present results of applying RRTs to hybrid systems—we are the first researchers to present a general framework for such systems. Finally, we present and demonstrate an algorithm for prioritized multi-agent planning with RRTs.
1 Introduction and Overview

Our goal in this project was to make advances in motion planning and control algorithms. Motion planning is one of the most important and heavily researched areas in computer science and machine learning. With applications in countless fields of practical importance including robotics, air traffic control, computer animation, and drug design, researchers have been creating and testing new motion planning algorithms for over two decades. As the dimensionality of problems faced becomes even greater, modern computational resources are rendered ineffective in solving these problems of exponentially-increasing difficulty. Naive brute-force methods have long-since been abandoned, as even some of the most trivial problems would require more computation time or memory than even today’s fastest computers can provide.

Motion planning and control are important in hybrid systems. In a hybrid system, an agent is allowed to behave in different ways depending on which mode it is in at a given time. A simple example of such a system is a pogo stick. The differential equations that govern the motion of a pogo stick are different when it is in the air (free flight mode) than when it is “bouncing” on the ground (spring mode).

Another area of motion planning that is being heavily researched is multi-agent planning. In multi-agent planning, we attempt to coordinate the motions of several agents so that they do not intersect and they all reach their assigned goals. Because the difficulty of the problem increases exponentially as the number of agents increases, efficient, scalable algorithms for multi-agent planning are crucial.

1.1 Contributions of the Project

In this project, we investigated the applicability of Rapidly-Exploring Random Trees (RRTs) to several problem domains. RRTs were first proposed for finding configuration paths in robot motion planning problems, but the generality of the algorithm makes RRTs suitable for numerous
other domains. We have proposed, designed, implemented, and tested several new planners based on the RRT algorithm, and we have made specific advances with RRTs in the following domains:

- basic machine learning and control problems
- hybrid systems
- multi-agent planning

In order to better gauge the applicability, strengths, and weaknesses of RRTs, we applied the RRT algorithm to several standard problems in machine learning and control. Using an inverted pendulum with nonlinear differential constraints and a motor at the joint, we demonstrated the effectiveness of RRTs for finding solutions. We built on these results by applying RRTs to the “acrobot” problem.

Hybrid systems, in which the differential constraints of a robot or system vary depending upon which mode it is currently in, are another hot area of research. We showed that RRTs are adaptable to such systems.

Finally, we introduce an efficient and general algorithm for prioritized multi-agent planning with RRTs. Although not complete\footnote{A complete motion planning algorithm is one that is guaranteed to produce solution if one exists.} in terms of being guaranteed to always find a solution, we show that our algorithm is effective for solving complex air-traffic control problems. Further, we describe how it can be made the basis for a complete algorithm.

1.2 Preview

Section 1 introduces basic motion planning in addition to motion planning for hybrid and multi-agent systems. We also introduce holonomic and nonholonomic planning and briefly describe some general approaches to motion planning.

Section 2 introduces RRTs, including the basic algorithm and useful properties of RRTs.
Section 3 discusses results of applying RRTs to some basic machine learning and control problems.

Section 4 discusses a general hybrid-systems framework for RRTs and gives results of using a hybrid RRT to solve a simple hybrid-systems motion planning problem.

Section 5 presents an algorithm for prioritized multi-agent planning with RRTs. In Section 5, we also give results for applying this algorithm to holonomic and nonholonomic control problems. Further, we propose a theoretically-complete multi-agent algorithm based on the prioritized algorithm.

Section 6 describes the implementation details of our RRT planners.

Section 7 reviews our results and proposes future directions for RRT research.

1.3 Motion Planning Introduction

This section introduces a formal framework for describing motion planning problems. Although the following specification is not necessarily broad enough to encompass all possible motion planning problems of any number of agents, we can easily extend it to describe more complex problems. The reader is encouraged to consult [17] for more information; the specification below comes from [17].

Definition 1 The basic motion planning problem can be defined as follows:

We are given an initial position and orientation, A(start), and a goal position and orientation, A(goal), of A, a robot in a Euclidean space W, called the workspace and represented as \( \mathbb{R}^N \) with \( N = 2 \) or \( 3 \). Generate a path \( \tau \) specifying a continuous sequence of positions and orientations of A avoiding contact with all obstacles starting at A(start) and terminating at A(goal). Report failure if no such path exists.
The same planning problem can also be stated as a configuration problem: given configuration \( q_{\text{start}} \) of \( A \), a specification of the position and orientation of \( A \) with respect to some fixed reference frame, find a path from \( q_{\text{start}} \) to a goal configuration \( q_{\text{goal}} \). The configuration space of \( A \) is the space \( C \) of all configurations of \( A \). Essentially, the configuration space is a transformation from the physical space in which the robot is of finite-size into another space in which the robot is treated as a point. We can describe a configuration by a list of real parameters. In Figure 1, the configuration space of the L-shaped robot is three-dimensional, being described by three parameters, \((x, y, \theta)\).

In this context, we define a path \( \tau \) from the configuration \( q_{\text{start}} \) to the configuration \( q_{\text{goal}} \) as a continuous map \( \tau : [0,1] \rightarrow C \) with \( \tau(0) = q_{\text{start}} \) and \( \tau(1) = q_{\text{goal}} \). Free space is any portion of the configuration space that is not occupied by obstacles, and a free path between an initial configuration and a goal configuration is a path which lies completely in free space and does not come into contact with any obstacles.
### 1.3.1 Hybrid Systems

Here, we formalize the notion of a hybrid system. Researchers in the computer science and control theory communities have produced many models for describing the dynamics of hybrid systems (e.g., see [5, 23, 11, 24]). For the purpose of discussion in this document, we consider a simple illustrative case, in which the constituent continuous state and input spaces (in each mode) are the same. Thus, we have a hybrid system of the form

\[
\begin{align*}
\dot{x} &= f(x, u, q), \quad x \notin J(x, u, q) \\
(x, q)^+ &= D(x, u, q), \quad x \in J(x, u, q).
\end{align*}
\]

Here, \(x \in X\) is the continuous state, \(u \in U\) is the input, and \(q \in Q \simeq \{1, 2, \ldots, N\}\) is the discrete state or mode. Also, \(f(\cdot, \cdot, q)\) is the continuous dynamics, \(J(\cdot, \cdot, q)\) is the jump set, and \(D(\cdot, \cdot, q)\) is the discrete transition map, all for mode \(q\). The map \(D\) relates the post-jump hybrid state \((x, q)^+\) from the pre-jump hybrid state \((x, q)\). The input \(u\), which can include both continuous and discrete components, allows the introduction of non-determinism in the model, and can be used to represent the action of control algorithms and the effect of environmental disturbances. The evolution of the discrete state \(q\) models switches in the control laws and discrete events in the environment, such as failures.

Briefly, the dynamics are as follows: the system starts at hybrid state \((x(t_0), q_0)\) and evolves according to \(f(\cdot, \cdot, q_0)\), until the set \(J(\cdot, \cdot, q_0)\) is reached. At this time, say \(t_1\), the continuous and/or discrete state instantaneously jump to the hybrid state \((x(t_1^+), q_1) = D(x(t_1), u, q_0)\), from which the evolution continues. While terse, the above model encompasses both autonomous and controlled switching and jumps, and allows modeling of a large class of embedded systems, including ground, air, and space vehicles and robots; see [6, 5] for more details.

Given a model of the system in the form (1), the motion planning problem for hybrid systems is essentially the same as the basic motion planning problem defined in Definition 1 above.
1.3.2 Multi-Agent Planning

We now build upon the basic motion planning problem stated above to deal with multiple agents. The following description is adapted from [1] and [17].

**Definition 2 Planning with Multiple Robots**

Given robots \( A_1, A_2, \ldots, A_n \) moving in the same \( k \)-dimensional workspace \( W \) with stationary obstacles \( B_1, B_2, \ldots, B_m \): find a collision-free path for each of the agents from an initial configuration to a goal configuration. The agents move independently of one another, but any two of them cannot occupy the same space at the same time. The composite configuration space of all robots consists of the product of the configuration spaces of the robots \( C_{\text{composite}} = C_1 \times C_2 \times \ldots \times C_n \).

The complexity of the motion planning problem increases exponentially as the number of dimensions in the configuration space increases. As the size of the configuration space increases linearly with the number of agents, multi-agent planners must employ clever techniques to cope with this growing complexity. We can classify most of these techniques under one of two general approaches: global planning and local planning. In global planning, also known as centralized planning, the coordinated paths of all agents in the workspace are planned together as a single path in their composite configuration space. Another way of looking at global planning is by imagining that all of the robots \( A_1 \) through \( A_n \) are actually one large robot \( A_{\text{composite}} \) with configuration space \( C_{\text{composite}} \). By looking at the problem this way, it reduces to the basic motion planning problem of one robot. Global methods, although simple and complete, suffer because they deal head-on with the exponential complexity inherent in multi-dimensional planning.

Local planning methods attempt to divide multi-agent planning problems into smaller problems involving fewer agents in order to minimize computational effort. In local planning (also known as decoupled planning), we attempt to plan the paths of the agents independently of each other first and later deal with any multi-agent conflicts that may arise. Local methods are not
guaranteed to generate a free path, even when such a path exists, but they are still useful for many practical situations. One of the difficulties with local planning is finding effective mechanisms for resolving agent-agent conflicts. For descriptions of conflict resolution mechanisms, see [17, p. 375].

**Prioritized planning** is one specific method for local planning. In prioritized planning, we plan each robot’s path independently of the others’ in some prioritized order. For example, if we are planning the paths of several airplanes from their respective source airports to their target airports, we might give aircraft with earlier take-off times a higher priority.

**Path coordination** is another method for local planning. In path coordination, we plan the paths of two robots in concert. This is done by first generating a free path for each robot independent of the other and then coordinating these two paths so that the robots do not collide [17, p. 379]. Latombe notes that planners based on path coordination are usually not as powerful as prioritized planners [17, p. 380], but Siméon *et al.* give a resolution-complete algorithm that is both efficient and complete for a large class of inputs [25]. Their planner, which is based on a bounding-box representation of obstacles, was able to coordinate the paths of 32 robots in 39 seconds and 150 robots in 250 seconds.²

### 1.4 Holonomic and Nonholonomic Planning

The following discussion is adapted from [17]. The control problems we dealt with in this project had either holonomic or nonholonomic constraints. A **holonomic** constraint is an equality relation among A’s configuration parameters that can be solved for at least one of the parameters. This type of constraint makes the planner’s job easier because *k* such constraints reduces the effective dimension of the actual configuration space by *k*. For example, if A is a three-dimensional body that can translate in any direction but is constrained to rotate only about a fixed axis rel-

---
²Planning was done on a Sun Ultra-1 170 MHz.
ative to itself, the problem is made much simpler. Although the configuration space of A could typically be described by six parameters, the holonomic constraint applied means that A’s configuration can be fully described by four parameters.

Unlike holonomic constraints, nonholonomic constraints usually make a planner’s job much more difficult. A nonholonomic constraint is a non-integrable equation involving the configuration parameters and their derivatives. As Latombe notes, nonholonomic constraints do not reduce the dimension of the space of configurations attainable by the robot, but they do reduce the dimensions of the space of possible differential motions at any given configuration [17].

As an example, let us consider the simple car-like robot modeled as a rectangular object moving in $\mathbb{R}^2$ shown in Figure 2. We can drive this car to any position with any orientation. Let us represent this configuration as $(x, y, \theta)$. At any instant during a motion, the velocity of the car has to point along its main axis. Therefore, the motion is constrained by the relation: $-\sin(\theta)dx + \cos(\theta)dy = 0$. This equation is non-integrable and does not prevent the car from ultimately reaching any desired configuration, and it is hence a nonholonomic equality constraint.

![Car-like robot with inherent nonholonomic constraints](image)

**Figure 2:** A car-like robot with inherent nonholonomic constraints (adapted from [19])

Nonholonomic constraints restrict the geometry of the feasible free paths between two configurations. The instantaneous motion of the car in Figure 2 is determined by two parameters: the linear velocity along its main axis and the steering angle $\phi$. When the steering angle is
non-zero, the robot’s orientation changes, as does its linear velocity. This implies that the car’s configurations can span a three-dimensional space. Although all configurations are therefore attainable, actions such as parallel parking are difficult in practice because the dynamics of the car restrict the motions it can make at any given instant—it cannot simply “scoot” sideways into a parking space.

1.5 Current Planning Approaches

Despite the bleak picture painted in Section 1, great strides have been made in motion-planning in the last two decades. Roadmap, cell decomposition, and potential field methods have all been proven effective in solving an array of motion planning problems (see [17]). Each technique encompasses a broad family of approaches to motion planning, and each tackles motion planning problems and copes with the “curse of dimensionality”—the exponential increase in computational power needed as the number of dimensions increases—in a different way. The following descriptions are adapted from [17].

1.5.1 Roadmap Methods

Roadmap methods consist of capturing the connectivity of the robot’s free space in a network of curves called a roadmap. Once constructed, the roadmap $R$ is used as a set of standardized paths. Path planning is thus reduced to connecting the initial configuration $A(\text{start})$ to $R$, connecting the goal configuration $A(\text{goal})$ to $R$, and searching $R$ for a path between these points. Numerous roadmap methods have been proposed, including visibility graph, freeway net, and silhouette roadmaps. Figure 3 shows a visibility roadmap with obstacles CB1, CB2, and CB3. This is a visibility roadmap because the curves in the roadmap are formed by connecting each vertex of the obstacles to all other vertices that are visible. In this case, planning would consist of finding a free path from the initial configuration to the roadmap, finding a free path from the
goal configuration to the roadmap, and searching the roadmap for a path between these two connection points. In a probabilistic roadmap planner, the vertices of the roadmap are generated at random in the collision-free subset of the configuration space, and edges connect vertices within a specified distance threshold ([9, 16]; see Figure 27 on p. 39).

1.5.2 Cell Decomposition

Cell decomposition methods can be separated into two categories: exact and approximate methods. Each category involves decomposing the robot’s free space into simple regions, called cells. Cells are constructed so that we can easily generate a path between any two configurations in a cell. To plan a robot’s path, we simply find a channel of consecutively adjacent cells connecting the start and goal configurations. Exact cell decomposition methods split the free space into cells whose union is exactly the free space. Generally, exact cell composition methods are computationally difficult because of the numerous geometries they confront. Approximate cell decomposition methods produce cells of a predefined shape, and the union of these cells does not necessarily compose the entire free space. In Figure 4, we find a free path from a start
configuration in Cell 1 to a goal configuration in Cell 19 by searching the corresponding graph (right) for a path connecting vertices 1 and 19.

![Cell Decomposition example](image)

Figure 4: Cell Decomposition example (from [2])

### 1.5.3 Potential Fields

In potential field methods, a robot in its configuration space is treated as a particle moving under the influence of an artificial potential produced by the start and goal configurations along with the obstacles. Treating the goal configuration as a basin of attraction and the start configuration and obstacles as repulsive forces, potential fields are sometimes used in combination with other planning methods. Figure 5 shows a potential field for a workspace with start configuration $q_{init}$ and goal configuration $q_{goal}$.

![Potential field example](image)

Figure 5: Potential field example (from [17])
Unlike the two other planning approaches, potential fields do not require a pre-planning step because they do not need to capture the connectivity in the free space before the initial and goal configurations are given. This is advantageous because it makes potential field methods more straightforward than methods based on the two other approaches, but it also presents a disadvantage: when a new start and goal configuration are given, no previously computed information can be used in solving the problem—we are forced to start from scratch. This is different from roadmaps and cell decomposition, where the data from the pre-planning step can be reused for any given start and goal configuration pairs.
2 RRT Introduction

Rapidly-Exploring Random Trees (RRT’s; see Figure 6) are a stochastic approach to dealing with the curse of dimensionality. Designed to be applicable to many domains including both holonomic and nonholonomic planning, a RRT is a search tree grown from an initial state that makes incremental steps toward unexplored regions of the configuration space. Based on a simple algorithm, RRT’s are exempt from some of the problems facing other planning methods including local minima (potential fields), overcrowded roads (roadmaps with multiple agents), and complex geometry calculations (cell decomposition).

Figure 6: Growth of typical RRT over time, from left to right

To construct an RRT from an initial configuration \( q_{\text{start}} \), the algorithm selects a random point in the configuration space, \( q_{\text{rand}} \) during each iteration (see Figure 7). Next, we find \( q_{\text{near}} \), the vertex in the tree that is closest to \( q_{\text{rand}} \). Finally, we find \( q_{\text{new}} \), the new vertex to add to the tree, by taking a finite step from \( q_{\text{near}} \) toward \( q_{\text{rand}} \). If \( \epsilon \), the step-length, is greater than the distance from \( q_{\text{near}} \) to \( q_{\text{rand}} \), \( q_{\text{new}} \) is set equal to \( q_{\text{rand}} \). The algorithm finishes when \( q_{\text{new}} \) is close enough to \( q_{\text{goal}} \). At this point, we can say that we have found a free path from the initial configuration \( q_{\text{init}} \) to the goal configuration \( q_{\text{goal}} \). The algorithm can continue to grow the tree until a solution of the desired optimality is found (see Figure 6).

We shall see later that distance metrics are one of the most important considerations in
**Figure 7:** The basic RRT construction algorithm (left) and diagram (right) (Adapted from [21]) implementing a successful RRT planner. A perfect distance metric would yield a solution in optimal time, but a bad distance metric may never yield a solution in a finite amount of time. In addition to growing a tree from the starting state, many RRT implementations grow a second tree from the goal state. Such trees expand in four steps:

1. Grow start-tree toward a random unexplored configuration.
2. Grow goal-tree toward a random unexplored configuration.
3. Grow start tree toward goal tree. At each iteration, select a random node in the goal tree to grow toward it.
4. Grow goal tree toward start tree.

A solution path is found when the two trees finally connect. Such dual-RRT’s can potentially reduce the time complexity of a search from $O(b^n)$ to $O(2 \times b^{n/2})$, where $n$ is the number of steps from the start configuration to the goal configuration and $b$ is the average branching factor of each RRT. In the case of no obstacles, the minimum number of steps from start to goal is $D(q_{start}, q_{goal}) / \epsilon$ where the numerator is the Euclidean distance from the start configuration to the goal configuration and $\epsilon$ is the step-size.
2.1 Voronoi Interpretation

The Voronoi diagram of a set of points is the locus of all points that are no nearer to one point than another. The Voronoi region of a point, \( P \), is the set of all points that are closer to \( P \) than to any other point (see Figure 8).

![Voronoi diagrams](image)

Figure 8: Examples of Voronoi diagrams (in red) for figures of two points (left), three points (middle), and several points (right). The diagram for the lattice (right) image is from [9].

Although RRTs are nondeterministic, LaValle and Kuffner [21] have shown that RRTs possess probabilistic completeness—they will eventually find a solution to any configuration problem if such a solution exists. RRTs are also biased towards selecting the largest Voronoi regions in exploring the configuration space for future expansion. This is the so-called “rapidly-exploring” property of RRTs, and it explains why RRTs tend to fill unexplored regions of a space.

![RRT interpretation](image)

Figure 9: Voronoi interpretation of RRT (adapted from [21]). Because the probability that a Voronoi region will be selected for growth is directly proportional to its size, larger (more empty) regions are more likely to be selected. This explains the “rapidly-exploring” property of RRTs.
3 Applying RRTs to Control Problems

Other researchers have applied RRT’s to planning problems of various types including path-steering, manipulation planning for digital actors, varieties of holonomic planning, and attitude control (kinodynamic planning) [20, 22, 10]. To our knowledge, we are the first experimenters to test RRT’s on standard control problems. It is our hope that by studying the RRT’s performance in these common problems, we will be able to gauge the strengths and weaknesses of RRT’s compared to other approaches and introduce RRTs to a wider range of applications.

3.1 Pendulum Swing-Up

The first experiment we conducted was applying the RRT to the swing-up problem for a nonlinear pendulum:

Definition 3 Pendulum Swing-Up

Given:

- A pendulum of mass $m$ and length $l$ with equation of motion
  \[ \ddot{\theta} = \frac{-3g}{2l} \sin \theta - \frac{3\tau}{ml^2} \]
- Motor at the joint which can apply torques of $\tau \in \{-1, 0, 1\}$ units
- Initial state of $\theta = 0$ (down) and $\dot{\theta} = 0$
- Goal state of $\theta = \pi$ (up) and $\dot{\theta} = 0$

![Figure 10: Pendulum](image)

The goal for the planner is to find a series of torque-time pairs that get the pendulum to the goal state. In all but the most trivial cases, the motor is unable to lift the pendulum to the goal state in one smooth motion. The pendulum therefore must be swung back and forth until it achieves sufficient velocity to reach the goal configuration.
Figure 11: Single- and Dual-RRT Solutions to the Pendulum Swing-Up Problem. The horizontal axis corresponds to $\theta$ and the vertical axis to $\dot{\theta}$. The left image shows a single-tree RRT solution for the pendulum problem after 5600 iterations. The right image shows a dual-tree RRT solution after 3300 iterations (solution in dark).

Our first try at solving the problem, a single-tree RRT using the straightforward Euclidean metric, $\rho = \sqrt{(\Delta \theta)^2 + (\Delta \dot{\theta})^2}$, proved to be quite successful. Usually finding a solution in less than 10,000 iterations (only a few seconds of computation on most modern computers), our implementation showed that the RRT algorithm is both fast and adaptable to many problem domains. See Figure 11 (left).

The dual-tree solution to the same problem was also impressive, sometimes finding a path to the goal state in close to half the time of its single-tree relative. One interesting characteristic of the solution trees is how clearly each demonstrates the dynamics of the system (see right of Figure 11). In Figure 12–14, we see how the behavior of the systems varies as progressively higher torques are allowed. The torques used and the number of iterations until a solution was found are listed. With torques of $\{-4, -1, 0, 1, 4\}$, we are able to reach the goal configuration without swinging the pendulum back and forth. Figure 15 shows how the RRT covers the configuration space, even with nonholonomic constraints.
Figure 12: Torques of $\{-1, 0, 1\}$. 6800 iterations.

Figure 13: Torques of $\{-2, -1, 0, 1, 2\}$. 3800 iterations.

Figure 14: Torques of $\{-4, -1, 0, 1, 4\}$. 2300 iterations.
Figure 15: Single-tree RRT run of inverted pendulum problem after 4000 iterations. Torques used were \{-4, -2, -1, 2, 4\}. Notice how well the RRT covers the space.

3.2 Acrobot

For our second experiment, we tested the RRT algorithm on a problem of higher dimensionality. The acrobot has gained attention in recent literature as an interesting control task in the area of reinforcement learning [9]. Analogous to a gymnast swinging on a high-bar, the acrobot has been studied by both control engineers and machine learning researchers. The equations of motion for the acrobot in Figure 16 come from [27, p. 271]. A time step of 0.05 seconds was used in the simulation, with actions chosen after every four time steps. The torque applied at the second joint is denoted by \( \tau \in \{-1, 0, 1\} \). Joint positions could be in the range \([-\pi, \pi]\) but \( \theta_2 \) was not allowed to pass through \( \pi \) (an acrobat’s legs cannot pass through his head). Angular velocities were limited to \( \dot{\theta}_1 \in [-4\pi, 4\pi] \) and \( \dot{\theta}_2 \in [-9\pi, 9\pi] \). The constants were \( m_1 = m_2 = 1 \) (masses of the links), \( l_1 = l_2 = 1 \) (lengths of links), \( l_{c1} = l_{c2} = .5 \) (lengths to center of mass of links), \( I_1 = I_2 = 1 \) (moments of inertia of links), and \( g = 9.8 \) (gravitational constant).

There are numerous goals that planning systems can attempt to reach when controlling the acrobot, but most involve reaching various vertical levels. In our testing, we attempted to swing the tip of the acrobot above some vertical level, \( y = y_{\text{goal}} \). The single-tree RRT had no problem finding a solution to the acrobot tip-goal problem, but the dual-tree version did not fare as well.
The reason for the dual-tree RRT’s failure is simple: dual-tree RRTs require us to specify a finite set of goal-configurations from which to grow the goal-tree. In the acrobot, there are an infinite number of configurations with the desired tip-height. Therefore, by limiting the RRT to a finite number of goal configurations, we make its task much harder.

One question that we may investigate further is how well the RRT-based solution compares quantitatively to other planners. Unlike some of the competing planners, the RRT is based on virtually no domain-specific knowledge except for the acrobot’s equations of motion, yet the RRT planner was able to perform well compared to published metrics of energy efficiency and time efficiency [27].

\[
\begin{align*}
\ddot{\phi}_1 &= -d_1^{-1}(d_2 \ddot{\theta}_2 + \phi_1) \\
\ddot{\theta}_2 &= (m_2 l_2^2 + l_2 - \frac{d_2}{d_1})^{-1}\left(d + \frac{d_2}{d_1} \phi_1 - m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_2) - \phi_2\right) \\
d_1 &= m_1 l_1^2 + m_2 (l_1^2 + l_2^2 + 2l_1 l_2 \cos(\theta_2)) + I_1 + I_2 \\
d_2 &= m_2 (l_2^2 + l_1 l_2 \cos(\theta_2)) + I_2 \\
\phi_1 &= -m_2 l_1 l_2 \dot{\theta}_2^2 \sin(\theta_2) - 2m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_1 \sin(\theta_2) + (m_1 l_1 + m_2 l_2)g \cos(\theta_1 - \pi/2) + \phi_2 \\
\phi_2 &= m_2 l_2 g \cos(\theta_1 + \theta_2 - \pi/2)
\end{align*}
\]

Figure 16: Acrobot equations of motion (left) and diagram (adapted from [27])

Figure 17 shows the vertical position of the tip versus time for one run of one version of the RRT-controlled acrobot, and the figure below it shows the positions corresponding to locally-maximal tip positions. Like the inverted pendulum, the acrobot had to swing back and forth multiple times in order to reach the goal state. The example had starting position \(y = -2.0\) and \(y_{goal} = 1.0\).
Figure 17: Acrobot tip-height vs. time.

Figure 18: Time-lapse of acrobot motion planned by RRT. Each local maximum in Figure 17 corresponds to one of the instantaneous positions labeled C–K.
4 RRTs for Hybrid Systems

Emilio Frazzoli and his co-workers have used random search in the context of a hybrid “maneuver automaton” to plan motions for aerospace vehicles [13, 7, 12, 14]. To our knowledge, our work is the first general description of a hybrid RRT.

General, hybrid RRTs can be achieved in various ways, depending on the underlying hybrid systems model and specifics of the continuous and discrete dynamics (and symmetries therein). We now wish to give a taste of the way a hybrid RRT might work for the model (1) in Section 1.3.1. Below, we assume that the planning/control problem will have a target set $T \subset X \times Q$.

The simplest algorithm one might envision would explore reachable space by growing a forest of RRTs one in each mode, with jump points among various trees in the forest identified. In the more general case, evolution will start from a set of seeds in a start set $S \subset X \times Q$, encompassing one or more modes, and proceed from there according to the algorithm outlined below. One may think of the resulting tree as (a) growing in the hybrid state space, $X \times Q$, or (b) as growing in $X$, with nodes and arcs colored/labeled by the current mode.

Even under this setup, there are several cases to consider:

1. General specifications; $S$, $T$, $J$, and $D$ are arbitrary.

2. Homogeneous specifications: $S = B \times Q$ and $T = G \times Q$. i.e., the start and target sets are independent of mode.

3. Homogeneous switching: $J(x, q) \equiv J(x)$ and $D(x, q) \equiv D(x)$, independent of $q$.

4. Unrestricted switching: $J(\cdot, q) = X$ for all $q$ and $D(x, q) = x$ for all $x$, $q$.

While the above is not exhaustive, it provides a sense of a few types of symmetries in the discrete dynamics that might be exploited by the algorithm.

In the case of unrestricted switching, the hybrid RRT algorithm is exactly the same as outlined in Figure 7 above, except that the control set is augmented to allow mode changes:
$U \ni U \times Q$. The other cases are non-trivial. In the case of homogeneous specifications, $x_{\text{rand}}$ lives, and distances are measured in, the continuous state space $X$; in the general case, $x_{\text{rand}}$ lives, and distances are measured in, the hybrid state space $X \times Q$. The latter brings up the issue of designing metrics for combined continuous and discrete space, which is a topic for future research. In either case, the New-State function must respect the hybrid dynamics. Typically, for purely continuous RRTs, the states examined come from extending the state $x_{\text{near}}$ according to the dynamics $f(x, \cdot)$ for a fixed time and for various (sampled) $u \in U$. In the hybrid case, this continues to hold for $(x_{\text{near}}, q_{\text{near}})$ if there are no intersections with the jump set $J(\cdot, q_{\text{near}})$. If there are, evolution continues from the destination point(s), using the same or different $u$, until the desired amount of time elapses.

![Diagram](image)

Figure 19: Stair Climbing: an example hybrid RRT

In Figure 19 we give an example of a hybrid RRT. Pictured from left to right in each row are four square floors, 1 through 4. Stairs (jumps) are given by triangles, with destinations given by
inverted triangles in the next highest floor. The tree started in the gray square in the center of floor 1, and the target set is the gray square on floor 4. Successive rows represent different stages in the expansion process. The hybrid state is \( s = (x, y, q) \in [-20, 20] \times [-20, 20] \times \{1, 2, 3, 4\} \). The distance metric used is \( \rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + 20|q_1 - q_2|} \). This example shows that RRTs are adaptable to hybrid systems.
5 Multi-agent Planning with RRTs

In the sections that follow, we describe an algorithm for prioritized multi-agent planning based on RRTs. We will see that this algorithm is highly scalable and that its running time is impressively fast. Finally, we will make conflict-resolution suggestions for future implementations.

5.1 Prioritized Multi-Agent RRT Algorithm

See Figure 20 for a detailed description of the algorithm. At a high level, prioritized planning with RRTs can be achieved by using one tree for each agent. Once it is a robot’s turn, we start its tree from the robot’s initial configuration and grow it until it reaches the goal configuration (see Figure 21). Note that we do not grow trees for multiple robots concurrently—one robot’s tree is grown from start to finish before we begin building another robot’s tree (see Figure 21). Planning proceeds as in the normal RRT algorithm except for intersection testing. Each new vertex in a tree must be tested against vertices in all other trees whose global creation times are identical to the new vertex. For holonomic trees, a vertex’s global creation time is the global creation time of its parent (source) vertex plus one. The global creation time of the start vertex is equal to the start time of the robot. We need not test against vertices in our own tree, since a robot cannot run into another copy of itself. Testing against static obstacles is done as usual. $N_{\text{max}}$ is the largest number of agents allowed to operate in the workspace at once.

In all of our implementations, which were inspired by air-traffic control, the actual number of agents operating in the workspace at once was sometimes less than $N_{\text{max}}$. For a transient period at the beginning of each run, the number of agents allowed to operate in the workspace was gradually increased until it reached $N_{\text{max}}$. In addition, there was a certain probability that an airplane would take off if the current number of airplanes operating in the workspace, $N_{\text{current}}$, was less than $N_{\text{max}}$. Practically speaking, this means that $N_{\text{current}}$ was usually 95%–100% of $N_{\text{max}}$.

---

3This was done to prevent $N_{\text{max}}$ “airplanes” from all taking off at the same time.
PLAN-MULTI-AGENTS($N_{max}$)
1 repeat
2 if CAN-START-NEW-TREE($N_{max}$)
3 then
4 $T_{new} \leftarrow$ CREATE-TREE()
5 $T_{grown} \leftarrow$ GROW-TREE-TO-COMPLETION($T_{new}$)
6 REMOVE-NON-SOLUTION-VERTICES($T_{grown}$)
7 STORE($AllTrees$, $T_{grown}$)
8 else
9 $currenttime \leftarrow currenttime + 1$
10 REMOVE-OLD-TREES()
11 until DONE

GROW-TREE-TO-COMPLETION($T$)
1 $q_{start} \leftarrow q_{start}[T]$
2 $q_{goal} \leftarrow q_{goal}[T]$
3 repeat
4 $q_{rand} \leftarrow$ RAND-CONFIGURATION
5 $q_{near} \leftarrow$ NEAREST-NEIGHBOR-VERTEX($q_{rand}$, $T$)
6 $u \leftarrow$ SELECT-INPUT($q_{rand}$, $q_{near}$)
7 $q_{new} \leftarrow$ NEW-CONFIGURATION($q_{near}$, $u$)
8 $time[q_{new}] \leftarrow time[q_{near}] + 1$
9 if IS-NOT-SAFE-CONFIGURATION($q_{new}$)
10 then repeat
11 ADD-VERTEX($T$, $q_{new}$)
12 ADD-EDGE($T$, $q_{near}$, $q_{new}$, $u$)
13 until IS-AT-GOAL($q_{new}$, $q_{goal}$)
14 return $T$

IS-NOT-SAFE-CONFIGURATION($q_{test}$)
1 for each $T$ in $AllTrees$
2 do $q_{T} = \text{VERTEX-CREATED-AT-TIME}(T, time[q_{test}])$
3 if TOO-CLOSE($q_{test}$, $q_{T}$)
4 then return $FALSE$
5 return $TRUE$

Figure 20: Prioritized multi-agent RRT algorithm. Note that the algorithm does not use dual-tree RRTs. Also, we can alternate between selecting a random configuration and selecting the goal configuration (to grow towards) in line 4 of GROW-TREE-TO-COMPLETION.
$N_{\text{max}}$ after the transient start-up period.

In order to make intersection testing fast, we can remove all non-solution vertices from a tree once a solution has been found, since the robot will not pass through these vertices on its way to the goal. This implies that, for each completed tree, there is exactly one vertex created at each discrete time in the range $[t_{\text{start vertex}}, t_{\text{goal vertex}}]$. Therefore, to test whether a new vertex created at time $t_{\text{new}}$ conflicts with any existing trees, we can simply query each tree for vertices created at $t_{\text{new}}$. Since each tree has at most one vertex created at $t_{\text{new}}$, the running time of intersection-testing is linear in the number of trees. This is crucial to the performance of the overall algorithm.

### 5.2 Analysis of Prioritized Multi-Agent RRT Algorithm

Our first test of the prioritized RRT was based on an air-traffic control model. In our workspace, we created six “airports” at which “airplanes” could start and finish their paths. The first implementation used holonomic airplanes with two configuration-space dimensions, $x$ and $y$. Airplanes were not allowed to fly within a certain fixed distance, $d_{\text{safe}}$, of other airplanes in the workspace except within a small radius, $d_{\text{airport}}$, around each airport inside of which planes could be closer. Figures 22 and 23 show a snapshot of the operation of the airplanes and a closeup of conflict avoidance, respectively.

In general, the running time and scalability results were impressive. With $d_{\text{safe}} = .03$, we were able to reach $N_{\text{max}} = 800$, although planning 3000 paths took nearly a day of computation. Figure 25 shows running time versus $N_{\text{max}}$ for $d_{\text{safe}} = .06$.

---

4This query by creation time is a constant-time operation if implemented using a simple hashing scheme. See Section 6 for more details.
Figure 21: Prioritized planning RRT algorithm for 3 simple holonomic robots. The RRT for each robot is shown in the top row, with the corresponding overall solution after each step shown on the bottom. First, we find a free path for the top-priority robot (left). Next, we find a free path for the second robot that does not conflict with the path for the first robot (middle). Finally, we plan a path for the third robot that does not conflict with the first or second robots’ paths. Note that although the solution paths cross in the $x$-$y$ domain, these intersections are non-conflicting because the robots reach the crossings at different times.
Figure 22: Holonomic air-traffic control model planned by prioritized RRT. Airplanes have dots in the middle, and airports are filled. The color of each airplane corresponds to the color of its destination airport. The maximum number of aircraft operating at any time was 50. The workspace was $[0, 1] \times [0, 1]$ and $d_{soe} = 0.06$ (each disk therefore has a radius of .03).

Figure 23: Close-up view of conflict avoidance. The highest-priority agent is black, and the lowest-priority agent is white. Numbers on circles correspond to the instantaneous time. In this situation, notice how the lowest-priority (white) agent must avoid the other agents at $t = 9–11$ by moving to the right.
5.2.1 Nonholonomic Constraints

The algorithm also handled nonholonomic agents admirably. We modified the above air traffic model to incorporate nonholonomic constraints by giving each agent dynamics as follows:

\[
\begin{align*}
\dot{x} &= u_1 \cos(\phi) \\
\dot{y} &= u_1 \sin(\phi) \\
\dot{\phi} &= u_2
\end{align*}
\]

(2)

In the above equations, \( u_1 \) and \( u_2 \) are the inputs chosen by the RRT from a finite set of possible inputs. The time step was chosen such that \( d_{safe} \) could still be maintained given maximum speed \( u_1 \). Given the above equations and the same workspace as described in Figure 22, the planner was able to plan for as many as forty nonholonomic airplanes.

5.2.2 Dynamic Envelopes

In many applications, it is important to have fail-safe mechanisms in place in case the agents ever need to make a sudden (emergency) stop. In such domains, we must be able to guarantee at all times that all agents, with maximum negative acceleration \( A_{max} \), would be able to completely stop if necessary without hitting any other robots or obstacles. We have tested the RRT for such cases, using a dynamically-expanding safety region, or \textit{envelope}, to ensure safe stopping ability. Instead of merely using a static disk or “hockey puck” to enforce safety constraints, the safety region’s length in the direction of the agent’s velocity is computed to be at least as long as the agent’s braking distance (see Figure 24). For RRT implementations of dynamic safety regions, we need only alter the intersection testing function. In Figure 24, which demonstrates agents
with dynamic envelopes operating in the same workspace, the dynamics of each robot are:

\[
\begin{align*}
    \dot{x} &= v \cos(\phi) \\
    \dot{y} &= v \sin(\phi) \\
    \dot{v} &= u_1 \\
    \dot{\phi} &= u_2
\end{align*}
\]  

Figure 24: Dynamic safety region for a nonholonomic agent (left) and multiple agents operating in workspace (right). The agents can always stop within their safety regions, and the size of an agent’s safety region is proportional to its instantaneous speed. Thus, unforeseen collisions can be avoided by stopping.

5.2.3 Running Time

The primary factor in determining the overall speed and success of the algorithm was \( N_{\text{max}} \). As noted above, intersection testing ran in \( O(N_{\text{max}}) \) because each new tree’s vertices had to be queried against at most one vertex from each other tree. In future implementations, it may be possible to increase the speed of this operation to \( O(\log N_{\text{max}}) \) by using a tree-based spatial querying data structure to access all vertices created at time \( t_{\text{query}} \) instead of linearly testing each vertex for proximity to the new vertex. Another factor contributing to running time was
the "fullness" of the workspace. On the one hand, if only a few planes are operating, few intersections occur in finding a free path for a new plane. On the other hand, if the workspace is already full of planes (say, 50% of the workspace is filled by planes and their safety-disks), then we will encounter a high percentage of intersections in planning the path of another plane, leading to a longer running time. In test runs, the overall running time was $O(N_{\text{max}}^3)$, but as noted above, future implementations might be able to reduce this to $O(N_{\text{max}}^2 \log N_{\text{max}})$. Figure 25 shows running time versus $N_{\text{max}}$ for $d_{\text{safe}} = .06$ with holonomic constraints. A similar trend was seen for $d_{\text{safe}} = .03$, except planning 3000 free paths took approximately 15 minutes with $N_{\text{max}} = 200$ and nearly 18 hours with $N_{\text{max}} = 800$.

Figure 25: Running Time vs. Maximum Number of Agents ($N_{\text{max}}$). These results are for holonomic planning of 3000 agent free paths with $d_{\text{safe}} = .06$. Planning was run on an AMD Athlon Thunderbird 1400 with 512 MB RAM, and times above are given in terms of approximate overall CPU time.

### 5.3 Open Issues and Future Work

While the running time mentioned above is satisfactory considering the exponential complexity of complete multi-agent planning (cf. page 7), there are drawbacks to the above algorithm related
to its prioritized nature. As noted in Section 1.3.2, prioritized planning is not guaranteed to find a solution for certain planning problems, even when a solution exists. In this subsection, we first introduce one technique for avoiding conflicts using a configuration space-time metric [17, p. 22]. Finally, we suggest a way of implementing a “complete” planner based on the above algorithm and the notion of a hybrid global-local planner [1].

Prioritized planners often fail when given narrow passageways in the configuration space. This is especially true with a prioritized RRT that does not incorporate time into its distance metric. In such implementations, if we are given a sufficiently narrow corridor and two agents that need to pass through in opposite directions, the probability of failure is high. If we ever add a vertex inside the narrow corridor to the lower-priority agent’s tree, this tree will fail to ever find a free path if this new vertex corresponds to a time at which the higher-priority agent is also passing through the corridor. The lower priority agent’s tree will continually “run into” the higher-priority agent’s tree.

![Figure 26: Conflict situation where prioritized planning with naive RRT is likely to fail. Since the path of A is planned without regard to B, it is unlikely that we will ever find a path for B that is not blocked by A in the narrow corridor.](image)

The easiest solution to the above problem is to incorporate vertex creation time into the distance metric. If, instead of using the Euclidean $x$-$y$ distance, we use the metric $\rho = \sqrt{(\Delta x)^2 + (\Delta y)^2 + \eta(t_{v-in-tree} - t_{query})}$ where $0 < \eta \ll 1$ (for example $\eta = .0001$), we are more likely to find a solution in a finite amount of time.\footnote{In using the configuration space-time metric, the time for random configurations $t(q_{rand})$ is chosen from the}
ward newer vertices, thus eventually selecting vertices created at times after the higher-priority agent has already passed through the narrow corridor.

Although helpful, the above metric cannot guarantee to eliminate all possible conflicts. In order to alleviate this problem, we propose a hybrid local-global motion planner based on the above prioritized RRT algorithm. A similar planner based on potential fields is demonstrated in [1]. The basic concept is to run the prioritized algorithm as presented above but to respond to unresolvable conflicts by planning the paths of the conflicting agents concurrently in their coordinated configuration space. In Figure 26, if we find that robot B is unable to find a free path because of conflicts with robot A, we will delete the free-path we found for A and find a free path for the two robots together in their four-dimensional, coordinated configuration space \( C_{AB} \). If there is another robot \( A' \) (with higher priority than \( A \)) that we find conflicts with the coordinated path of \( A \) and \( B \), we will delete the free path for \( A' \) and plan the paths for \( A, B, \) and \( A' \) in their six-dimensional, coordinated configuration space \( C_{A'AB} \). In the worst case, such switching could continue to occur until we are planning in the composite configuration space of all robots.

The above algorithm is guaranteed to find a free path if such a path exists, but its running time could potentially be exponential if we are forced to plan in the composite configuration space of all robots. Such an occurrence is unlikely, however, and real-world performance of the algorithm would probably be close to that of our initial prioritized RRT algorithm for most situations. In many ways, the “C-subsumption” hybrid algorithm just described strikes a good balance between efficiency and completeness.

random range \([0,1,\ldots,t_{oldest}+1]\) where \( t_{oldest} \) is the creation time of the oldest node in the tree so far.
6 Building Our RRT Planner

Now that we have discussed the results of our RRT planner, let us pause to consider some of the theoretical and practical issues that affected our RRT implementations.

6.1 Design Issues

Our planners were all implemented in C++ on the Microsoft Windows platform. We used OpenGL to generate images, and we edited our source code in Microsoft Visual C++ 7.0. C++ was chosen because of its efficiency compared to other languages like Matlab and Java and because it allowed object-oriented programming.

Although our implementation was not fully object-oriented, we used classes to implement our RRT class, CRert. Each CRert contained a vector of Edge’s and a CNearestNeighborContainer to hold vertices. CNearestNeighborContainer merely defined an interface, and both the linear and tree-based NN container implementations conformed to this interface. We also implemented vertices as a class CVertex, and each CVertex contained a vector of double’s corresponding to the configuration at the vertex. Another class used by CRert was the quasi-random number generator, CHaltonGenerator. CHaltonGenerator can generate random configurations for any number of dimensions less than 100.

In order to make the code easily portable to different problem domains, most of the important numerical constants were defined in a separate, domain-specific section of a header file. The most difficult aspect of porting the code to different problem domains was the numerical solution of nonholonomic constraints.

Nonholonomic constraints were also difficult to implement in dual-tree RRT implementations. We usually had to use a smaller step size for the goal-tree than the start-tree because of the inherent inaccuracy in stepping backward using the differential equations for the nonholonomic constraints. Both Euler’s method and Runge-Kutta were used at different periods in the
6.2 Distance Metrics

Distance metrics play a crucial role in the RRT algorithm, and the success of the search for a free path depends heavily on the distance metric used [21]. A perfect metric is one that is always correct in assessing how “close” a candidate state is to a desired state relative to another candidate state. Unfortunately, having knowledge of a perfect distance metric implies having knowledge of an optimal solution,\(^6\) which is exactly what we are seeking. Therefore, the best we can hope for is a metric that closely approximates a perfect metric.

The most common metric used for our applications was the weighted Euclidean distance. In the two-dimensional case, this is simply \(\sqrt{\alpha_1 x_1^2 + \alpha_2 x_2^2}\). The two constant coefficients \(\alpha_1\) and \(\alpha_2\) give certain dimensions more weight than others. In most applications, this metric fares reasonably well and can be generalized to any number of dimensions.

The Euclidean distance is a special case of the Minkowski \(L_m\) distance metric. For any integer \(m \geq 1\), the \(L_m\)-distance between points \(p = (p_1, p_2, \ldots, p_d)\) and \(q = (q_1, q_2, \ldots, q_d)\) in \(\mathbb{R}^d\) is defined as \(m \left( \sum_{i=1}^d |p_i - q_i|^m \right)^{1/m}\). In the limiting case where \(m = \infty\), this is equivalent to \(\max_{1 \leq i \leq d} |p_i - q_i|\). The \(L_1\), \(L_2\), and \(L_\infty\) metrics are known as the Manhattan, Euclidean, and max metrics, respectively [3]. Each of these metrics has potential applications to RRTs. For an excellent discussion of RRT fundamentals, including distance metrics, see [21].

6.3 Nearest-Neighbors

Because the RRT algorithm finds the nearest-neighboring (NN) vertex to a random state at every iteration, the computational efficiency of the nearest-neighbors search is crucial for implementing a fast RRT-based planner. In the simplest NN implementations, a search takes \(O(dn)\) time, where \(d\) is

\(^6\)If we had the perfect distance metric, we could find the optimal solution using gradient descent. Also see [21] for more information.
\(d\) is the configuration space dimension and \(n\) is the number of nodes in the tree. Unfortunately, the performance of the linear version becomes intolerable as the tree grows large.

Implementation of a fast NN data structure was not a specification for this project, and most of the problem domains encountered were easily tractable with linear NN searches. Real-time and near-real-time RRT planners must use faster NN algorithms, and some of our RRT implementations use a data structure presented in [15] that can retrieve the nearest neighbor to a search point in \(O(\log n \log d)\) time.

Since exactness is not necessary for RRT’s to work properly, the RRT algorithm may be adaptable to use an approximate nearest neighbor algorithm. Such an algorithm is given in \([3]\) that runs in \(O(c_{d, \epsilon} \log n)\) time, where \(c_{d, \epsilon} \leq d[1 + 6d/\epsilon]^d\), and \(\epsilon \in \mathbb{R}\) indicates that the approximate nearest neighbor will be within a factor of \((1 + \epsilon)\) of the true nearest neighbor. Since the NN search is the speed bottleneck of the RRT algorithm, approximate NN searching may be vital for solving more complex problems.

As we mentioned above, the prioritized RRT algorithm must be able to quickly access nodes by global creation time. Since these times are discrete, and there are a finite number of times between \(t_{\text{algorithm-start}}\) and \(t_{\text{current}}\), we can implement this fast access using hashing. One option is to simply store a table of pointers, indexed by global creation time, that each point to the head of a linked list holding all vertices created at that time.

### 6.4 Pseudo-Random and Quasi-Random Numbers

Path-planning systems of the past used pseudo-random numbers as the foundation for stochastic algorithms. Such numbers, which are designed to have statistical properties identical to truly random numbers, suffer from some of the same problems as their truly random counterparts. Specifically, pseudo-random numbers often appear to have a non-homogeneous local distribution—or “clumpiness”—that causes some local regions to contain significantly more
points than others.

If pseudo-random numbers are merely statistical distributions designed to mimic certain properties, there is no reason why we cannot design other distributions that still capture what we mean by “random” but avoid some of the problems found in pseudo-random numbers. So-called quasi-random numbers minimize the quantitative analogues of “clumpiness”—the dispersion and the discrepancy of a sample.\footnote{In the standard Euclidean case, the dispersion of a set of points is the radius of the largest empty sphere that we can draw within the field of points (not including edge-cases). For an explanation of discrepancy, see [9].} For a discussion of the mathematics behind quasi-random numbers, see [8, 9].

![Figure 27: Probabilistic roadmap path planner using pseudo-random numbers (left), and quasi-random numbers (right). Both solutions are formed from 1000 generated points, and the source points are shown on the outside. Notice the better uniformity of the quasi-random (Halton) points. Reproduced from [8, 9].](image)

Quasi-random numbers already have well-known applications in optimization, integration, and image processing, and Branicky et al. have taken the first steps in showing the usefulness of quasi-random numbers in path planning (see [8, 9]). The above images are taken from a probabilistic roadmap planner using pseudo-random numbers (left) and quasi-random numbers (right). Although they both use exactly 1000 samples, the quasi-random version is clearly superior in both state-space exploration and in finding a path through the narrow corridor.

Experiments with quasi-random points for RRT’s show little gain so far. Using Halton points
instead of pseudo-random points has produced disappointing results, often requiring 10% more iterations before finding a solution. However, future efforts may employ Hammersley and other quasi-random sampling techniques, along with “lazy” versions of the RRT similar to the lazy probabilistic roadmaps discussed in [4].
7 Conclusions and Future Work

7.1 Summary of the Project

In this project, we used RRTs to make advances in the domains of machine learning, hybrid systems, and multi-agent planning. We showed how planners based on the RRT algorithm can be successful for solving problems in these domains.

In Section 3, we used RRTs to find solutions for two basic problems in machine learning and control: the inverted pendulum and the acrobot. For the inverted pendulum, both the single and dual-tree RRTs were able to find solutions, with the dual-tree version taking approximately half as much time as the single-tree version. For the acrobot, while the single-tree RRT was consistently able to find solutions for reaching the desired tip-height, the dual-version was usually unable to find solutions. The dual-tree version did not fare as well because dual-tree RRTs require us to specify a finite set of goal-configurations from which to grow the goal-tree(s). In the acrobot, there are an infinite number of configurations with the desired tip-height. Therefore, by limiting the RRT to a finite number of goal configurations, we may make its task harder.

In Section 4, we introduced a general hybrid-systems framework for RRTs and presented results of applying the RRT to a simple, holonomic hybrid system.

In Section 5, we presented a prioritized algorithm based on RRTs for multi-agent planning. Although not complete, the algorithm was able to solve both holonomic and nonholonomic air-traffic control problems. We also suggested and implemented a similar planner based on a configuration space-time metric that was able to resolve certain agent-agent conflicts. Finally, we proposed, but did not implement, a complete hybrid local-global planning algorithm based on the prioritized planner.
7.2 Future Work

There is still much to be done in RRT research, and we have merely scratched the surface of potential applications for RRTs. Future research may include looking into the following areas:

**Different quasi-random techniques:** While Halton point results have been discouraging thus far, use of Hammersley points, rotated lattices, and other sampling techniques may yield better results (see [4] and [9]). Initial experiments with lazy evaluation of the RRT have also been promising.

**Complete multi-agent RRT planner:** As described above, we are hopeful that a hybrid local-global planner (cf. [1]) based on the prioritized multi-agent RRT algorithm will be successful. Such a planner might only be a fraction slower than the prioritized algorithm above for many practical cases.

**Better metrics and optimal solutions:** As we noted above, a perfect metric for motion planning is impractical, because finding such a metric is just as hard as finding an optimal free path. Although the Euclidean distance fared well for the problems we worked with in our research, there may be other simple metrics that give better solutions. Also, it may be useful to compare the optimality of RRT solutions to the acrobot problem with published solutions based on more domain-specific planning methods (see [27]). RRTs are a general technique based on little domain-specific knowledge. If they can perform as well as or better than planners based on domain-specific knowledge, this would give us a glimpse into the power of RRTs as a general-purpose planning method.
References


