

RRTs for Nonlinear, Discrete, and Hybrid Planning and Control*

Michael S. Branicky, Michael M. Curtiss, Joshua A. Levine, and Stuart B. Morgan
 Electrical Engineering and Computer Science Dept.
 Case Western Reserve University
 {msb11,mmc18,jal31,sbm5}@cwru.edu

Abstract—In this paper, we describe a planning and control approach in terms of sampling using Rapidly-exploring Random Trees (RRTs), which were introduced by LaValle. We review RRTs for motion planning and show how to use them to solve standard nonlinear control problems. We extend them to the case of hybrid systems and describe our modifications to LaValle’s Motion Strategy Library to allow for hybrid motion planning. Finally, we extend them to purely discrete spaces (using heuristic evaluation as a distance metric) and provide computational experiments comparing them to conventional methods, such as A*.

I. INTRODUCTION AND OVERVIEW

Often, one is interested in solving path planning and control problems of the form: given a system model, find a (controlled) trajectory of the system that leads from a start to a goal configuration. Whether one uses dynamic programming or complete motion planning algorithms, such problems are exponential in the state-space (and control) dimensions. Attempts to fight the curse of dimensionality have led to the introduction of randomized (or Monte-Carlo) approaches to path planning that are capable of solving many challenging problems efficiently, at the expense of being able to guarantee that a solution will be found in finite time. Most randomized planning methods are designed for the generalized mover’s problem, including randomized potential fields, probabilistic roadmaps, Ariande’s clew algorithm, and other planners (see [6] for citations). Derandomization of some of these algorithms has been explored in [6].

Many path planning methods, including dynamic programming and most of the randomized planning methods can be categorized as incremental search methods. In these methods a tree (or two trees in the bidirectional version) is grown incrementally from the initial state by adding a new edge and vertex in each iteration after performing some local motion. There are generally two decision problems at each iteration: 1) which vertex should be selected for expansion? 2) what local motion should be executed? The answers to these questions are given by the particular method. For example, dynamic programming selects the “active” vertex with the lowest cost, and the local motion attempts to move in a direction not yet considered. Among path planning techniques, incremental search methods are most amenable to the inclusion of differential constraints.

An incremental search method that has achieved considerable success in the design of feasible trajectories is

```

BUILD_RRT( $x_{init}$ )
1   $T.init(x_{init});$ 
2  for  $k = 1$  to  $K$  do
3       $x_{rand} \leftarrow RANDOM\_STATE();$ 
4      EXTEND( $T, x_{rand}$ );
5  Return  $T$ 
    
```

```

EXTEND( $T, x$ )
1   $x_{near} \leftarrow NEAREST\_NEIGHBOR(x, T);$ 
2  if NEW_STATE( $x, x_{near}, x_{new}, u_{new}$ ) then
3       $T.add\_vertex(x_{new});$ 
4       $T.add\_edge(x_{near}, x_{new}, u_{new});$ 
    
```

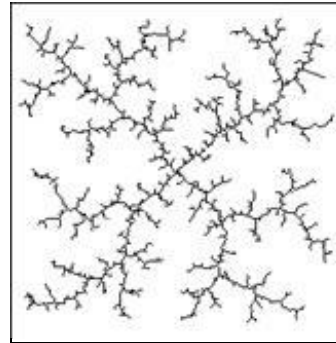


Fig. 1. The basic RRT construction algorithm (top) and an example RRT (bottom)

the Rapidly-exploring Random Tree (RRT) [7]. An RRT is an exploration algorithm for quickly searching high-dimensional spaces that have both global constraints (arising from workspace obstacles and velocity bounds) and differential constraints (arising from kinematics and dynamics). The key idea is to bias the exploration toward unexplored portions of the space by randomly sampling points in the state space, and incrementally “pulling” the search tree toward them. The resulting method is much more efficient than brute-force exploration of the state space. This method can solve challenging problems that involve state spaces of up to twelve dimensions with the inclusion of both differential constraints and complicated obstacle constraints. In [3], [4], we used RRTs to solve nonlinear control problems and extended them to the case of hybrid systems. Herein, we review those results, extend RRTs to fully discrete domains, and describe computational experiments and tools using RRTs for nonlinear, discrete, and hybrid planning and control.

II. RRT BACKGROUND

The basic RRT construction algorithm is given in Figure 1 (top). A simple iteration is performed in which each step attempts to extend the RRT by adding a new vertex that is

*This work was supported in part by NSF award 0208919.

biased by a randomly-selected state, $x \in X$. The EXTEND function selects the nearest vertex already in the RRT to x . The “nearest” vertex is chosen according to the metric, ρ . The function NEW_STATE makes a motion toward x by applying an input $u \in U$ for some time increment Δt . This input can be chosen at random, or selected by trying all possible inputs and choosing the one that yields a new state as close as possible to the sample, x (if U is infinite, then a finite approximation or analytical technique can be used). NEW_STATE implicitly uses the collision detection function to determine whether the new state (and all intermediate states) satisfy the global constraints. For many problems, this can be performed quickly (“almost constant time”) using incremental distance computation algorithms by storing the relevant invariants with each of the RRT vertices. If NEW_STATE is successful, the new state and input are represented in x_{new} and u_{new} , respectively. The bottom of Figure 1 shows an RRT grown from the center of a square region in the plane. In this example, there are no differential constraints (motion in any direction is possible from any point). The incremental construction method biases the RRT to rapidly explore in the beginning, and then converge to a uniform coverage of the space [7]. The exploration is naturally biased towards vertices that have larger Voronoi regions. This causes the exploration to occur mostly on the unexplored portion of the state space.

In addition to growing a tree from the starting state, many RRT implementations grow a second tree from the goal state. Such trees grow in four steps:

- 1) Grow start-tree towards a random unexplored configuration.
- 2) Grow goal-tree towards a random unexplored configuration.
- 3) Grow start tree towards goal tree. At each iteration, select a random node in the goal tree to grow towards it.
- 4) Grow goal tree towards start tree. A solution path is found when the two trees finally connect.

III. RRTS FOR NONLINEAR CONTROL

Other researchers have applied RRTs to planning problems of various types including path-steering, manipulation planning for digital actors, varieties of holonomic planning, and attitude control (kinodynamic planning) [7]. To our knowledge, we are the first experimenters to test RRTs on standard control problems [3], [4]. 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.

A. Pendulum Swing-Up

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

- 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 tip 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 $\dot{\theta} = 0$

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. 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 2 (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 it demonstrates the dynamics of the system. See Figure 2 (right).

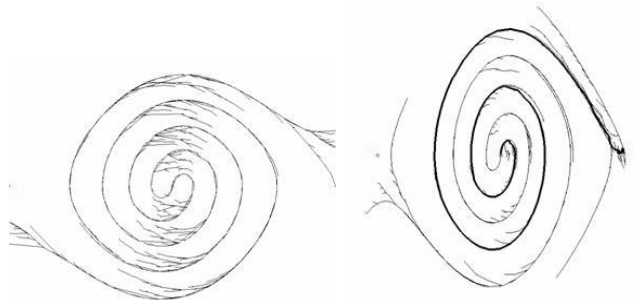


Fig. 2. Single- and Dual-RRT Solutions to the Pendulum Swing-Up Problem. The x -axis corresponds to θ and the y -axis to $\dot{\theta}$. The left image shows a single-tree RRT solution for the pendulum problem after 5600 iterations. The right image shows a dual-tree RRT search after 3300 iterations (solution in dark).

B. 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 [10]. 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 used come from [11, 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 $\tau \in \{-1, 0, 1\}$.

There were no constraints on the joint positions, but the 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, $y = y_{goal}$. The single-tree RRT had no problem finding a solution to the acrobot tip-goal problem. 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 [10], [11].

Figure 3 shows the vertical position of the tip of one version of the RRT-controlled acrobot versus time. Like the inverted pendulum, the acrobot had to swing back and forth multiple times in order to reach the goal state. The starting position was $y = -2.0$ and $y_{goal} = 1.0$. As for the time-lapse behavior, the RRT-controlled acrobot showed similar behavior to that shown in [11, p. 274].



Fig. 3. Acrobot Swing-Up Problem: vertical position versus time

C. Multi-Aircraft Planning

We also investigated prioritized RRT algorithms to plan for multiple aircraft in two-dimensions, traveling among six airports, with simple flight dynamics [4]. We were able to generate plans for up to 800 holonomic agents in the air at one time. We also generated plans for tens of nonholonomic agents (with unicycle dynamics) in the air at one time.

IV. RRTs FOR HYBRID SYSTEMS

A. Hybrid Systems

Researchers in the computer science and control theory communities have produced many models for describing the dynamics of hybrid systems [1], [2]. For the purpose of the 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

$$\begin{aligned} \dot{x} &= f(x, u, q), & x &\notin J(x, u, q) \\ (x, q)^+ &= D(x, u, q), & x &\in J(x, u, q). \end{aligned} \quad (1)$$

Here, $x \in X$ is the continuous state, $u \in U$ is the input, and $q \in Q \simeq \{1, 2, \dots, N\}$ 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 $(x(t_1^+), q_1) = D(x(t_1), u(t_1), q_0)$, 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 [1], [2] for more details. Below, we describe an approach to hybrid planning and control based on RRTs.

B. Extending RRTs to 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 [5]. However, we believe our work [3], [4] was the first general description of a hybrid RRT. We summarize it here.

A general, hybrid RRT 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). A 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 *hybrid-RRT 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 can be exploited by the algorithm.

In the case of unrestricted switching, the **hybrid-RRT algorithm** is exactly the same as outlined above, except that the control set is augmented to allow mode changes: $U \mapsto U \times Q$. The other cases are non-trivial. In the case of homogeneous specifications, x_{rand} lives, and distances are measured in, the continuous state space X ; in the general case, x_{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 of current 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_{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_{near}, q_{near}) if there are no intersections with the jump set $J(\cdot, q_{near})$. If there are, evolution continues from the destination point(s), using the same or different u , until the desired amount of time elapses.

In Figure 4 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 $s = (x, q) \in [-20, 20]^2 \times \{1, 2, 3, 4\}$. The metric used is $\rho(s_1, s_2) = \|x_1 - x_2\|_2 + 20|q_1 - q_2|$.

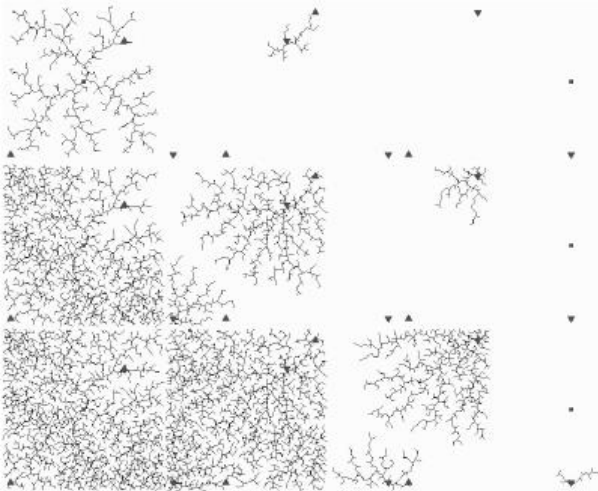


Fig. 4. Stair Climbing: an example hybrid RRT.

C. Computational Tool for Hybrid Systems

We have been working on a visual tool for manipulating and studying hybrid systems. (See [9] for more details throughout.) Our tool builds on the Motion Strategy Library (MSL) developed by Steve LaValle et al. [8]. The intended purpose of the MSL is to provide a generalized framework for development and testing of motion planning algorithms. Despite differences between our problem definition and the goals of the MSL, this framework is applicable to our needs. We want to use planning algorithms, particularly RRTs, to “walk” through hybrid systems, a similar task to planning paths through a motion strategy problem. The MSL provides straightforward extensibility for all aspects of its design, and serves as the basis for our tool.

1) *MSL Overview:* The MSL is composed of three subsystems: (1) an interface to input problems of arbitrary dimensions and geometries, as well as the dynamics of these problems; (2) a set of planning algorithms, ranging from PRMs to numerous RRT-variants; and (3) a graphical means for a user to study how effectively the planning algorithms solve these problems. A typical session involves running the MSL against a particular problem, selecting a planner, and using the interface to plan and view paths through the system. Figure 5 shows the MSL’s seven main objects, interactions between them, and the three main subsystems.

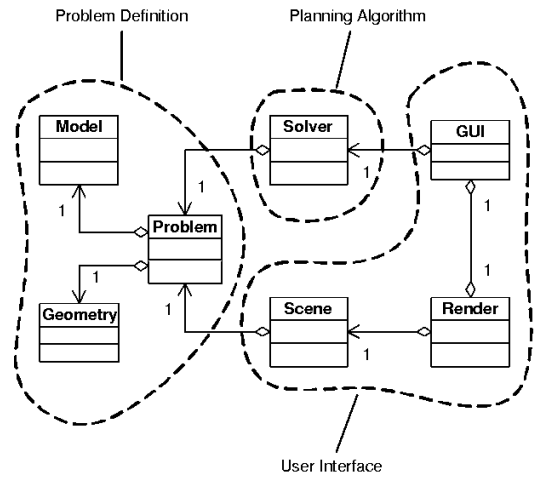


Fig. 5. MSL Class Hierarchy, Derived from [8]

The interface to a problem definition in the MSL is divided into two objects—a Geometry object and a Model object. The Geometry object contains physical representations of objects in the state space; in a motion-planning problem these include the robot and the obstacles among which it moves. Primarily, its role is to do collision detection checks, to prevent potential paths for the agent from intersecting the space of the obstacles. The Model object encapsulates the dynamics of the system, including a metric function for determining distance in the space as well as algorithms to

calculate future states for the planner given a current state, a time increment, and a control input. Given these two objects, a `Problem` object provides an encapsulation of both; it provides an interface for the planning and graphical objects.

The `Solver` hierarchy is a collection of different planning objects that are used to find motion planning solutions to problems. The largest subset of these solvers is the RRT branch, and it includes variants to grow two trees together as well as ones biased towards the goal. The important thing to note regarding the solvers is that there are a significant number of variants; hence in the MSL we are provided a large amount of extensibility to control how we actually do sampling-based planning.

The third main subsystem of the MSL is the graphical front end provided to the user. This part is composed of three main objects, the `Scene`, `Render`, and `GUI` objects and their respective hierarchies. The `Scene` object computes the physical locations of objects based on information provided by the `Problem` object. The `Render` object provides a hierarchy of different ways to draw objects based on different graphical libraries, including subclasses for SGI Iris Performer, Open Inventor, and OpenGL. A `Render` object receives most of its information from the `Scene` and `Problem` objects, and given this input it does the both the drawing of the problem and the animation of the solution. The final part of the user interface is the `GUI` object that provides a graphical control window for the user. In this window, users can select different types of planners, modify properties of the planning algorithm, and execute motion planning. In addition, the `GUI` object also provides an interface for the user to the commands of the `Render` object, by providing a set of animation controls that allow the user to start and stop animation, change the speed of the animation, and the rotate or translate the viewpoint.

2) *Extending the MSL:* Our needs require extending all three of the main areas discussed above—the problem definition, the planning algorithm, and the user interface. In terms of the problem definition, we implement both a new `Geometry` object and a new `Model` object. Our `Geometry` object contains the same information as a regular `Geometry` object, but also includes geometries for the state transitions of the hybrid system. Each transition can be modeled as a pair of (P, q) tuples, (P_s, q_s) and (P_f, q_f) , where P_i is a polygon in continuous space and q_i is its discrete state. In addition to collision detection, we include algorithms for “state transition detection.” That is, we use the same geometric algorithms for detecting if the planner intersects with an obstacle to detect if we intersected with a state transition.

The implementation of the `Model` object for hybrid systems is accomplished similarly. We override the methods for determining the metric to include some metric between two elements of the hybrid state. In some cases this includes the discrete information, but is not always required to do so. In the example we show later, we chose a metric that

was dependent on the continuous as well as the hybrid state. Also, the `Model` object is used for determining future states or “taking steps” throughout the continuous state. Since the `Model` object is independent of the `Geometry` object, the planning algorithm itself determines when state transitions occur, and reacts accordingly.

In addition to the `Model` and `Geometry` objects, we also must implement a new type of `Solver` class that plans against hybrid systems. Our hybrid RRT [3], [4] extends directly from the RRT branch of the `Solver` hierarchy, and plans in a similar manner to all other RRTs used in the MSL. However, one important difference is that at each iteration of the algorithm, the RRT also does a check to see if a state-transition has occurred. In effect, it queries the `Geometry` object, and if the newly planned state for the RRT “collides” with a state transition polygon, then it adds an additional node to the RRT, assuming that any time contact is made with a state transition, the hybrid automaton follows the jump.

The final addition to the MSL is developing the user interface to include information for hybrid systems. Our needs require extending both the `Render` and `GUI` objects. Specifically, in the `Render` object we needed to modify the input language to include discrete state information. Currently, the MSL uses a set of ASCII text files as a convenient means for input, extending the `Render` object to read our input language involved reading an extra discrete value for state information. In terms of drawing bodies, we chose to draw them on a one-state-at-a-time basis; hence, displaying or animating each body involves checking if that body’s state is the current state, and drawing them. A big addition to the `Render` object including drawing not only the path planned for the problem, but also including drawing the RRT as it gets planned as an optional way to view the problem. Our modified `GUI` object includes abilities to turn on and off these new features, as well as a means to select what discrete state is currently being viewed. See Figure 6.

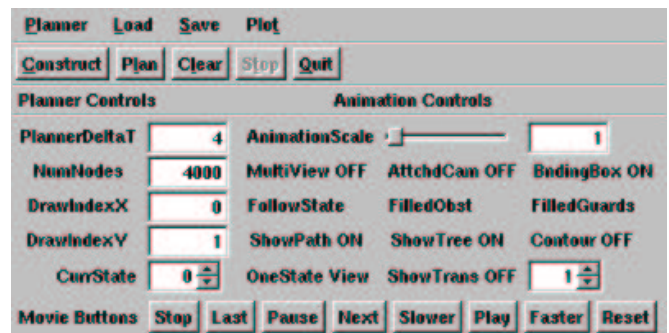


Fig. 6. Modified GUI Window

3) *Experimentation:* To test our extended MSL for hybrid systems, we have done example work using a four-story building, similar to the one used above. Specifically, states in our system consist of a two-dimensional coordinate combined with a discrete floor. Our hybrid state space is

$s = (x, q) \in [0, 50]^2 \times \{1, 2, 3, 4\}$. We use distance metric $\rho(s_1, s_2) = \|x_1 - x_2\|_2 + 50|q_1 - q_2|$. Given these as inputs to our model, we grow an RRT via the MSL to get a result like the one shown in Figure 7 (left).

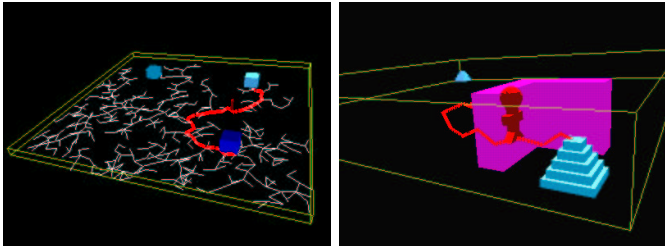


Fig. 7. Stair climber Examples 2d (left) 3d (right)

In this figure, the red (medium gray) peg represents a simple point objects translated from 2d space to 3d. The dark blue (dark gray) block is a state transition or “down stairs” in which q is decremented while the light blue (light gray) blocks are “up stairs” where q is incremented. The white line segments represent the RRT that has been grown through the system and the overlaid red (medium gray) segments represent the path determined by the RRT.

Figure 7 (right) shows a screen capture of a stair climber in a 3d space ($q = 1$), where the hybrid state space is now $s = (x, q) \in [0, 50]^2 \times [0, 10] \times \{1, 2, 3, 4\}$. Here we use distance metric $\rho(s_1, s_2) = \|x_1 - x_2\|_2 + 200|q_1 - q_2|$. This example takes full advantage of the MSL’s 3d drawing capabilities, as our agent is a stick-figure shape and the state transitions are now tiny pyramids. In addition, we make use of the MSL’s collision detection capabilities by providing obstacles, shown as the pink (light gray) L-shaped wall.

The examples above demonstrate a system with constant dynamics. That is, each step the RRT takes is governed by a simple constant function. However, we have continued our work by implementing examples that use more complicated dynamics. Specifically, hybrid dynamics can be included in the `Model` object for a given example problem. In the functions that determine the next state given a current state, a time step, and a control input, we include information regarding the explicit dynamics of the system. Our first steps have been using systems with different speeds on each “floor” in the stair-climbing problem; but the implementation is open enough to accept more complicated forms of dynamics in different hybrid system contexts. For example, we have studied rectangular and nonlinear hybrid systems [9].

Our initial progress using the MSL has been very positive. Given the framework we used above, combined with Emilio Frazzoli’s success using random searches for hybrid systems, we see no reason why any class of differential equations could not be applied in a similar manner. Additionally, by relying on the framework of the MSL, we gain the ability to apply any form of generalized planner (e.g. PRMs, RRT variants, etc.) to hybrid systems as well.

V. RRTs IN DISCRETE SPACES

A. Discrete Space Problems

In general, a discrete space problem is one where, given a finite set of states \mathbf{S} , with operators mapping $q \in \mathbf{S} \mapsto \mathbf{Q}' \subseteq \mathbf{S}$, one seeks a path from a start state s to a goal state $g \in \mathbf{G}$, the set of all goal states. Although optimal algorithms (such as A*) exist for such problems, they become infeasible for large problems. In addition, they are intrinsically goal-directed, making them ill-suited to road-mapping to solve repeated queries.

B. Discretized RRT Algorithms

By applying the same sort of heuristic information used in general informed search strategies, the RRT algorithm can be adapted for use in discrete state spaces. As in the original RRT algorithm, the tree begins with the initial state s . At each step, we select a random state q_r from the state-space which is not already in the tree, find the nearest state q_n already in the tree based on a heuristic estimate of distance to q_r , consider each possible operation from q_n , select the one which yields the state closest to q_r , and add an edge from q_n to it.

The primary difference between construction of continuous and discrete RRTs is in the necessity of selecting from a small set of possible operators when taking a step toward a random discrete state. Since it is not improbable that the best of the operators yields a new state which is no closer to q_r than q_n , and can in fact be further away, the performance of the RRT in discrete space is degraded by a decreased bias toward unexplored areas.

A new algorithm, which mitigates this issue, is the Rapidly-Exploring Random Leafy Tree (RRLT). The RRLT algorithm keeps a separate open list of all states reachable in one step from the current tree; the “leaves” of the tree nodes. When q_r is selected, the nearest leaf is located directly, using the same heuristic information, and added to the tree, while all of its successors are added to the open list. In this way the algorithm guarantees that at each step the tree grows as far as possible (heuristically) toward q_r . Although this algorithm is more memory-intensive than the simple discrete RRT algorithm, early results indicate that it is generally faster and finds better solutions. The RRLT algorithm also lends itself to optimizations which further reduce running-times.

C. Experiments and Results

In order to evaluate the potential of the RRT algorithms in discrete spaces, we explored their performance on several relatively simple discrete space problems.

The first problem we attempted was the 8-Puzzle (or more generally the $(k^2 - 1)$ -Puzzle), where the goal is to order a randomly arranged set of tiles, labeled 1 to 8, by number. At each step, it is possible to swap the positions of the empty space, and one of the adjacent tiles. Since this problem is easily solvable, with its low branching factor and average

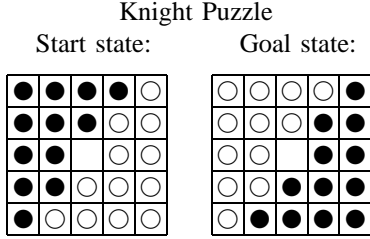


Fig. 8. The 5×5 Knight Puzzle

optimal solution length of approximately 22, it provides a simple test-case for the algorithms. Manhattan distance of each tile’s position in one board relative to the other was used as a distance heuristic for all tests.

A more difficult puzzle is the Knight Puzzle, which entails swapping the positions of 12 white knights and 12 black knights on a 5×5 chess board, using only valid knight moves. The branching factor is larger, and the optimal solution length is 36, leading to much larger search trees. Here, the heuristic used was the number of knight moves for each out of place knight to reach a destination position occupied by another out of place knight (an admissible but non-metric heuristic).

To introduce the idea of “obstacles,” we also tried variations of the Knight Puzzle with constrained moves. In the second version, we allow a piece to move only when a piece of the same color borders the destination on one of the four sides. In the third version, a move is allowed only if it leaves no pieces without an adjacent piece of the same color.

D. Results

Results for various RRT algorithms on the 8-Puzzle are shown below. In the case of the biased trees, the algorithm was modified to select the goal node as q_r half the time. The bi-directional search uses the method of growing each tree first toward a random point, then toward a random point in the other tree.

Algorithm	Nodes Exp'd	Open Nodes	Time (sec)	Soln. Len.
A*	1150	14850	22	2.00
Biased RRT	610	-	65	0.42
Biased RRLT	380	270	48	0.26
Bi-Dir. RRT	380	-	31	0.15
Bi-Dir. RRLT	290	220	30	0.14
Best-First	270	230	60	0.02

Results for the the Knight-Puzzle with obstacles show the RRLT’s effectiveness in a large and constrained space. In these trials, the RRLT’s are grown bidirectionally (best-first search is unidirectional, as it exhibits classic worst-case bidirectional search performance). Also, we made an improvement to the nearest-neighbor query algorithm as follows: by storing at each node the distance to its maximally distant descendant, we may use A* to search the tree for

nearest neighbors, pruning subtrees which could not yield a neighbor nearer than the best already found.¹ Version 1 is without obstacles; versions 2 and 3 are as described above. Although best-first search is much faster for the simple version of the problem, excessive memory paging prevents it from terminating after over 5 hours of computation on a machine with 512 MB of RAM. The fact the time and space requirements grow much more slowly for the RRLT as the space becomes less open shows that it is well-suited to finding paths around obstacles in the discrete space.

Algorithm	Nodes Exp'd	Open Nodes	Soln. Len.	Time (sec)
9x9 v 1 Best-First	1040	3930	316	0.5
9x9 v 2 Best-First	9680	23200	620	3.2
9x9 v 3 Best-First	-	-	-	-
9x9 v 1 RRLT	7100	27500	271	123.0
9x9 v 2 RRLT	9150	30200	315	171.0
9x9 v 3 RRLT	19900	57260	354	665.0

VI. REFERENCES

- [1] M.S. Branicky. *Studies in Hybrid Systems: Modeling, Analysis, and Control*. Sc.D. thesis, M.I.T., June 1995.
- [2] M.S. Branicky, V.S. Borkar, and S.K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE T-AC*, **43**(1):31–45, 1998.
- [3] M.S. Branicky and M.M. Curtiss. Nonlinear and Hybrid Control Via RRTs. *Proc. Intl. Symp. Math. Theory of Networks and Systems*, South Bend, August 2002.
- [4] M.M. Curtiss, *Motion Planning and Control using RRTs*, M.S. thesis, EECS, CWRU, May 2002. <http://dora.cwru.edu/msb/pubs/mmcMS.pdf>
- [5] E. Frazzoli, M.A. Dahleh, and E. Feron. A hybrid control architecture for aggressive maneuvering of autonomous helicopters. In *IEEE CDC*, December 1999.
- [6] S.M. LaValle and M.S. Branicky. On the relationship between classical grid search and probabilistic roadmaps. *Proc. Workshop Algorithmic Foundations of Robotics*, December 2002.
- [7] S.M. LaValle and J.J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Proc. Workshop Algorithmic Foundations of Robotics*, 2000.
- [8] S.M. LaValle *et al.*, Motion Strategy Library. <http://msl.cs.uiuc.edu/msl/>
- [9] J.A. Levine, *Sampling-Based Planning for Hybrid Systems*, M.S. thesis, EECS, CWRU, August 2003.
- [10] M.W. Spong. Swing up control of the acrobot. *Proc. IEEE Intl. Conf. Robotics Automation*, 1994.
- [11] R.S. Sutton and A.G. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

¹While this approach is only provably correct for admissible metric heuristics, our results show promise as an approximate nearest-neighbor algorithm with admissible but non-metric heuristics.