

Chapter 4

Path Planning: Incremental Searching Methods

Ariadne was the daughter of King Minos of Crete. Minos had Daedalus build a Labyrinth, a house of winding passages, to house the bull-man, the Minotaur, the beast that his wife Pasiphae bore after having intercourse with a bull. (Minos had refused to sacrifice a bull to Poseidon, as the king promised, so the god took revenge by causing his wife to desire the bull--but that's another story.) Minos required tribute from Athens in the form of young men and women to be sacrificed to the Minotaur.

Theseus, an Athenian, volunteered to accompany one of these groups of victims to deliver his country from the tribute to Minos. Ariadne fell in love with Theseus and gave him a thread which he let unwind through the Labyrinth so that he was able to kill the Minotaur and find his way back out again.

Ovid says that Daedalus built a house in which he confused the usual passages and deceived the eye with a conflicting maze of various wandering paths (in errorem variarum ambage viarum) (Metamorphoses 8.161):

"so Daedalus made the innumerable paths of deception [innumeras errore vias], and he was barely able to return to the entrance: so deceptive was the house [tanta est fallacia tecti]" (8.166-68).

Source: http://www.georgetown.edu/labyrinth/info_labyrinth/ariadne.html

4.1 Overview

This chapter presents computational approaches to the path planning problem. In general, it is assumed that a 2D or 3D world, \mathcal{W} , has been specified, which contains an obstacle region, \mathcal{O} , and a robot, \mathcal{A} . Based on possible transformations of \mathcal{A} , a configuration space, \mathcal{C} is defined. Given an initial configuration, q_{init} and a goal configuration, q_{goal} , the task is to compute a continuous, collision-free path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$.

There are numerous discouraging results in the literature which indicate the hardness of the general path planning problem. For example, it is known that path planning for:

- n polyhedral bodies that comprise a robot in a 3D world that contains a fixed number of polyhedral obstacles is PSPACE-hard.
- n rectangles with up, down, left, and right translation only is PSPACE-complete.
- a fixed robot among n moving obstacles with known trajectories is PSPACE-hard.
- a point robot with uncertainties in sensing and control in a 3D world is NEXPTIME-hard.

These and many other results generally indicate that it is highly unlikely or impossible that many general path planning problems will ever be solved by an efficient, polynomial-time algorithm.

These hardness results can be considered as lower bounds on the difficulty of the problem. These bounds indicate the complexity of the *problem*, as opposed to a particular *algorithm*. Even though it is believed that an efficient solution to the general path planning problem will never be found, it was important to develop a complete algorithm that actually solves the problem. This was first achieved in 1983 by Schwartz and Sharir in their application of cylindrical algebraic decomposition techniques to develop a cell decomposition algorithm that applies to any \mathcal{C}_{obs} that can be represented algebraically. Unfortunately, the time complexity of this algorithm was $O(2^{2^n})$, in which n is the dimension of the configuration space. In 1987, Canny proposed a general path planning algorithm that is the most efficient known to date, with a time complexity of $O(2^n)$. The technique involves powerful techniques from real algebraic geometry (particularly, the use of resultants and stratifications).

Although these complete approaches to path planning are very elegant, their running times would be too much for most practical path planning problems. This has led many researchers to develop efficient planning algorithms for more specific classes of problems, or to tolerate incomplete algorithms. Incomplete algorithms are not guaranteed to yield a solution, if a solution exists; however, any answer returned is guaranteed to be correct. In some applications this behavior is unacceptable, but in a wide variety of contexts, incompleteness can be tolerated. In this case it becomes particularly important to minimize the likelihood that the algorithm will fail, and in general to improve its computational performance as much as possible. Often times a very efficient algorithm can be developed if we are willing to pay a small price of occasionally missing the solution.

4.2 Metric Spaces

Distance is a fundamental notion in many path planning problems and approaches. It is generally useful to evaluate the distance from the robot to the goal, or to compare any two configurations. We are already quite familiar with Euclidean distance in the world, but how is distance measured in a general topological space, such as \mathcal{C} ?

A *metric space*, (X, ρ) , is a topological space, X , equipped with a function, $\rho : X \times X \rightarrow \mathbb{R}$ such that for any $x_1, x_2, x_3 \in X$:

1. $\rho(x_1, x_2) \geq 0$ (non-negativity)
2. $\rho(x_1, x_2) = 0$ if and only if $x_1 = x_2$ (reflexivity)
3. $\rho(x_1, x_2) = \rho(x_2, x_1)$ (symmetry)
4. $\rho(x_1, x_2) + \rho(x_2, x_3) \geq \rho(x_1, x_3)$ (triangle inequality) .

The function ρ defines distances between points in the metric space, and each of the four conditions on ρ agrees with our intuitions about distance. The final condition implies that ρ is optimal in the sense that the distance from x_1 to x_3 will always be less than or equal to the total distance obtained by traveling through an intermediate point x_2 , on the way from x_1 to x_3 .

Example 1 (*Euclidean space*) Suppose $X = \mathbb{R}^2$, and let a point in \mathbb{R}^2 be denoted by (x, y) . The Euclidean (or L^2) metric is defined as

$$\rho(x_1, y_1, x_2, y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Other metric spaces can be defined for \mathbb{R}^2 .

Example 2 (*Manhattan distance*) The Manhattan (or L^1) metric is defined over \mathbb{R}^2 as

$$\rho(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$

Example 3 (L^∞ distance) The L^∞ metric is defined over \mathbb{R}^2 as

$$\rho(x_1, y_1, x_2, y_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

Example 4 (*Distance in S^1 using θ*) In the case of $X = S^1$, the metric becomes slightly more complicated due to the topology of S^1 . If S^1 is represented by $\theta \in [0, 2\pi)$, then the metric may be defined as

$$\rho(\theta_1, \theta_2) = \min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|)$$

Example 5 (*Distance in S^1 using complex numbers*) If S^1 is represented by unit complex numbers, $a + bi$, then a metric may be defined as

$$\rho(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$$

A metric can be defined for a space such as $\mathcal{C} = \mathbb{R}^2 \times S^1$ by combining the metrics from \mathbb{R}^2 and S^1 . Suppose that two metric spaces, (X, ρ_x) , and (Y, ρ_y) , are defined. A metric space, (Z, ρ_z) , can be constructed for the Cartesian product $Z = X \times Y$ by defining the metric ρ_z as

$$\rho_z(z_1, z_2) = \rho(x_1, y_1, x_2, y_2) = c_1 \rho_x(x_1, x_2) + c_2 \rho_y(y_1, y_2),$$

in which $c_1 > 0$ and $c_2 > 0$ are any positive, real constants. Other combinations lead to a metric for Z , such as

$$\rho_z(z_1, z_2) = \sqrt{c_1 \rho_x^2(x_1, x_2) + c_2 \rho_y^2(y_1, y_2)}.$$

In either of these cases, two positive constants must be chosen. It is important to understand that many choices are possible, and there is no natural notion of a “correct” choice. A metric defined on $\mathcal{C} = \mathbb{R}^2 \times S^1$ must compare both distance in the plane and an angular quantity. For example, even if $c_1 = c_2 = 1$, the range for S^1 is $[0, 2\pi)$ using radians, but $[0, 360)$ using degrees. If the same constant c_2 is used in either case, two very different metrics will be obtained. The units that one are applied to \mathbb{R}^2 and S^1 are completely incompatible. In terms of path planning, this implies that no natural metric exists for configuration spaces.

A final metric will be given, which is perhaps the most appropriate for path planning problems; however, it is also the most complicated to evaluate. For any two configurations $q_1, q_2 \in \mathcal{C}$, for any configuration space, \mathcal{C} , the distance is

Example 6 (*Robot displacement metric*) $\rho(q_1, q_2) = \max_{a \in \mathcal{A}} \|a(q_1) - a(q_2)\|$,

in which $a(q_i)$ is the position of the point a in the world, when the robot, \mathcal{A} is at configuration q_i . Intuitively, the robot displacement metric yields the maximum amount in the world that any part of the robot is displaced from configuration q_1 to q_2 .

4.3 A General Framework

There are two kinds of incremental search methods: 1) single-tree and 2) bidirectional (two trees). In a single-tree method, a tree, $T(V, E)$ represents the status of the search. The tree, T , includes a set, V , of vertices that each represent a configuration, and a set, E , of edges. An edge, e , from vertex v_1 to v_2 represents a path from v_1 to v_2 . For simplicity, v will denote both the vertex in V and the corresponding configuration, $v \in \mathcal{C}$. Also, e will denote both the edge in E and the path, e , such that $e(0) = v_1$ and $e(1) = v_2$.

SINGLE_TREE_SEARCH

- 1 $V = \{q_{init}\}; E = \{\};$
 - 2 $v \leftarrow \text{VSF}(V);$
 - 3 $e \leftarrow \text{PCF}(v);$
 - 4 $V = V \cup \{e(1)\}; E = E \cup \{e\};$ (assuming PCF successful)
 - 5 **if** $e(1) \neq q_{goal}$ **goto** Step 2;
-

In bidirectional search, there are two trees. The tree $T_i(V_i, E_i)$ starts at q_{init} , and $T_g(V_g, E_g)$ starts at q_{goal} . The path planning problem is solved if the two trees reach a common grid point.

BIDIRECTIONAL_SEARCH

- 1 $V_i = \{q_{init}\}; E_i = \{\};$
 - 2 $V_g = \{q_{goal}\}; E_g = \{\};$
 - 3 $v_i \leftarrow \text{VSF}(V_i);$
 - 4 $e_i \leftarrow \text{PCF}(v_i);$
 - 5 $V_i = V_i \cup \{e_i(1)\}; E_i = E_i \cup \{e_i\};$ (assuming PCF successful)
 - 6 $v_g \leftarrow \text{VSF}(V_g);$
 - 7 $e_g \leftarrow \text{PCF}(v_g);$
 - 8 $V_g = V_g \cup \{e_g(1)\}; E_g = E_g \cup \{e_g\};$ (assuming PCF successful)
 - 9 **if not** $\text{CONNECTED}(T_i, T_g)$ **goto** Step 3;
-

4.4 Classical Grid-Based Search

Assume that a d -dimensional configuration space is represented as a unit cube, $\mathcal{C} = [0, 1]^d \setminus \sim$, in which \sim indicates that identifications of the sides of the cube are made to reflect the \mathcal{C} -space topology. Representing \mathcal{C} as a unit cube usually requires a reparamterization. For example, an angle $\theta \in [0, 2\pi)$ parameter would be replaced with $\theta/2\pi$ to make the range lie within $[0, 1]$.

Discretization Assume that \mathcal{C} is *discretized* by using the following resolutions $[k_1 \ k_2 \ \dots \ k_d]$, in which each k_i is a positive integer.

Let $\Delta q_i = [0 \ \dots \ 0 \ \frac{1}{k_i} \ 0 \ \dots \ 0]$.

A *grid point* is a configuration $q \in \mathcal{C}$ that can be expressed in the form

$$\sum_{i=1}^d j_i \Delta q_i,$$

in which each $j_i \in \{0, 1, \dots, k_i\}$. The integers j_1, \dots, j_d can be imagined as array indices for the grid. Let the term *boundary grid point* refer to a grid point that has $j_i = 0$ or $j_i = k_i$ for some i . Note that due to identification, boundary grid points might have more than one representation.

For simplicity, assume that q_{init} and q_{goal} are grid points. If they are not, then for each one, a short path would have to be defined which connects it to a nearby grid point.

Neighborhoods For each grid point, q , we want to describe nearby grid points.

If q is not a boundary grid point, then the 1 -neighborhood is defined as

$$N_1(q) = \{q + \Delta q_1, \dots, q + \Delta q_d, q - \Delta q_1, \dots, q - \Delta q_d\}.$$

For a d -dimensional configuration space there are at most $2d$ 1-neighbors. Special care must be given to defining the neighborhood of a boundary grid point to ensure that the identification is respected.

A 2 -neighborhood is defined as

$$N_2(q) = \{q \pm \Delta q_i \pm \Delta q_j \mid 1 \leq i, j \leq d, i \neq j\} \cup N_1(q).$$

In a similar manner, a k -neighborhood can be defined for any positive integer $k \leq d$. For a d -neighborhood, there are up to 2^{d+1} neighbors.

4.4.1 Dynamic programming

Let Q denote a priority queue of *active* vertices. The vertices in Q are sorted in increasing order by using the following cost, $c(q)$. This cost measures the total path length from $c(q)$ to the configuration at the root of the tree, T . The cost of each edge of the path is given by the metric on \mathcal{C} .

Let N denote a stack of grid points.

Initially, $Q = \{q_{init}\}$, which corresponds to the root of T . Also, $N = \{\}$.

Vertex Selection Function (VSF): The vertex selection function returns the first element of Q , without deleting it.

Path Creation Function (PCF): Assume that k has been selected for the definition of neighborhoods, N_k . Let v denote the vertex returned by the VSF. The PCF is:

PATH_CREATION_FUNCTION(v)

```

1  if  $N = \{\}$  then
2     $N \leftarrow N_k(v)$ ;
3  while ( $N \neq \{\}$ )
4     $q \leftarrow N.pop()$ ;
5    if ( $N = \{\}$ ) then
6       $Q.delete(v)$ ;
7    if ( $(q \notin V)$  and ( $CollisionFree(v, q)$ )) then
8       $Q.insert(q)$ ;
9    return  $q$ ;
10 return FAILURE
```

The test $q \notin V$ (is q already a vertex in T ?) can be performed in constant time by storing flags in a d -dimensional array in which each element corresponds to a grid point.

The CollisionFree function uses a collision detection or distance computation algorithm to ensure that the configurations along the shortest path from v to q are not in collision.

The dynamic programming algorithm is guaranteed to find shortest paths on the grid (the shortest that can be obtained under the assumption of a neighborhood type is used).

4.4.2 Other grid-based search algorithms

A* search This is the same as dynamic programming, except that the cost for sorting Q is

$$c(v) = c_i(v) + \hat{c}_g(v),$$

in which $c_i(v)$ denotes the cost from v to the root, and $\hat{c}_g(v)$ denotes an underestimate of the cost to the goal.

If $\hat{c}_g(v)$ is always an underestimate of the true cost, then the algorithm is still guaranteed to find shortest paths on the grid, but usually with much less exploration required. If $\hat{c}_g(v) = 0$, then dynamic programming is obtained.

BF* search The cost for sorting Q is

$$\hat{c}_g(v).$$

In this case, there is usually even less exploration, but optimal paths are not obtained.

Bidirectional dynamic programming Bidirectional approaches maintain two trees, T_i and T_g . Let Q_i and N_i be associated with T_i , and let Q_g and N_g be associated with T_g .

For bidirectional dynamic programming, the cost for sorting Q_i is $c_i(v)$, and the cost for sorting Q_g is $c_g(v)$, which is path length from q_{goal} to v .

For bidirectional A^* search, the cost for Q_i is $c_i(v) + \hat{c}_g(v)$, and the cost for Q_g is $\hat{c}_i(v) + c_g(v)$.

For bidirectional BF^* search, the cost for Q_i is $\hat{c}_g(v)$, and the cost for Q_g is $\hat{c}_i(v)$.

Bidirectional dynamic programming generally solves problems with less exploration than single-tree dynamic programming. However, bidirectional A^* and BF^* might or might not be faster than their single-tree counterparts. Problems sometimes occur because the trees are not biased to grow into each other. The tree T_i is biased toward q_{goal} , and T_g is biased to grow toward the q_{init} .

4.5 Randomized Potential Field

The methods of the previous section have great difficulty in escaping local minima in the search of high-dimensional configuration spaces. The randomized potential field approach was the first randomized planning method. It yields good performance for many problems, but it unfortunately requires many heuristic choices of parameters to help escape local minima. The basic steps involved in the approach are

- Define a metric, ρ , over \mathcal{C} , and consider $\rho(q, q_{goal})$ as a real-valued potential function.
- Define a DESCEND procedure that iteratively attempts to reduce the potential function by taking small steps in a direction that reduces $\rho(q, q_{goal})$.
- If the DESCEND procedure allows the configuration to fall into a local minimum of ρ without reaching the goal, then perform a random walk for a while, and then try DESCEND again. If this repeatedly fails, then randomly backtrack to a configuration already visited and try DESCEND again.

Define a mode variable, m with three possible values:

D Descend

E Escape

B Backtrack

4.6 Rapidly-Exploring Random Trees

Rapidly-Exploring Random Trees (RRTs) offer an alternative way to build path planning algorithms. They are similar in some ways to the Randomized Roadmap approach; however, RRTs can be easily extended to more challenging problems, such as nonholonomic planning and kinodynamic planning.

An RRT that is rooted at a configuration q_{init} and has K vertices is constructed using the following:

The basic RRT construction algorithm is given in Figure 4.1. A simple iteration is performed in which each step attempts to extend the RRT by adding a new vertex that is biased by a randomly-selected configuration. The EXTEND procedure, illustrated in Figure 4.2, selects the nearest vertex already in the

```

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

```

```

EXTEND( $\mathcal{T}, q$ )
1   $q_{near} \leftarrow \text{NEAREST\_NEIGHBOR}(q, \mathcal{T});$ 
2  if  $\text{NEW\_CONFIG}(q, q_{near}, q_{new}, u_{new})$  then
3       $\mathcal{T}.add\_vertex(q_{new});$ 
4       $\mathcal{T}.add\_edge(q_{near}, q_{new}, u_{new});$ 
5      if  $q_{new} = q$  then
6          Return Reached;
7      else
8          Return Advanced;
9  Return Trapped;

```

Figure 4.1: The basic RRT construction algorithm.

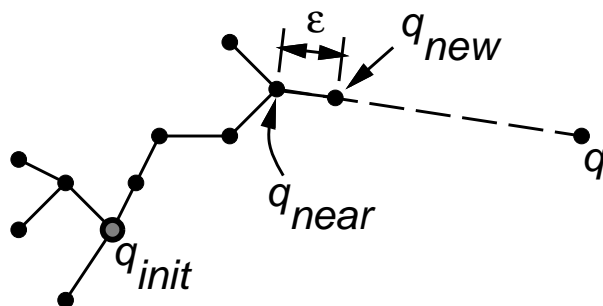


Figure 4.2: The EXTEND operation.

RRT to the given sample configuration. The “nearest” vertex is chosen according to the metric, ρ . The procedure `NEW_CONFIG` makes a motion toward q 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 configuration as close as possible to the sample, q (if U is infinite, then an approximation or analytical technique can be used). In the case of holonomic planning, the optimal value for u can be chosen easily by a simple vector calculation. `NEW_CONFIG` also implicitly uses the collision detection function to determine whether the new configuration (and all intermediate configurations) satisfy the global constraints. For many problems, this can be performed quickly (“almost constant time”) using incremental distance computation algorithms [6, 15, 17] by storing the relevant invariants with each of the RRT vertices. If `NEW_CONFIG` is successful, the new configuration and input are represented in q_{new} and u_{new} , respectively. Three situations can occur: *Reached*, in which the new vertex reaches the sample q (for the nonholonomic planning case, we might instead have a threshold, $\|q_{new} - q\| < \epsilon$ for a small $\epsilon > 0$); *Advanced*, in which a new vertex $q_{new} \neq q$ is added to the RRT; *Trapped*, in which `NEW_CONFIG` fails to produce a configuration that lies in \mathcal{C}_{free} . The top row of Figure 4.3 shows an RRT for a holonomic planning problem, constructed in a 2D square space. The lower figure shows the Voronoi diagram of the RRT vertices; note that the probability that a vertex is selected for extension is proportional to the area of its Voronoi region. This biases the RRT to rapidly explore. It has been shown that RRTs also arrive at a uniform coverage of the space, which is also a desirable property of the probabilistic roadmap planner.

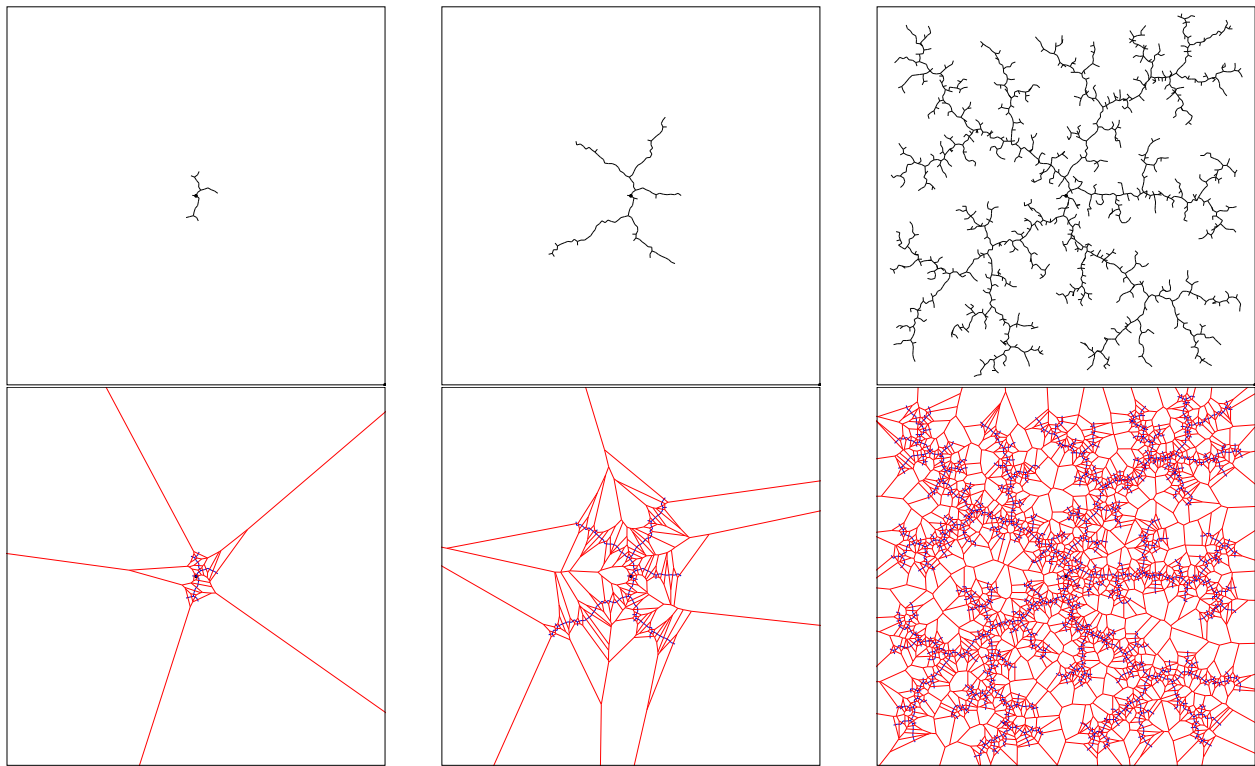


Figure 4.3: An RRT is biased by large Voronoi regions to rapidly explore, before uniformly covering the space.

4.6.1 Designing Path Planners

Now the focus is on developing path planners using RRTs. We generally consider the RRT as a building block that can be used to construct an efficient planner, as opposed to a path planning algorithm by itself. For example, one might use an RRT to escape local minima in a randomized potential field path planner. In [26], an RRT was used as the local planner for the probabilistic roadmap planner. We present several alternative RRT-based planners in this section. The recommended choice depends on several factors, such as whether differential constraints exist, the type of collision detection algorithm, or the efficiency of nearest neighbor computations.

Single-RRT Planners In principle, the basic RRT can be used in isolation as a path planner because its vertices will eventually cover a connected component of \mathcal{C}_{free} , coming arbitrarily close to any specified q_{goal} . The problem is that without any bias toward the goal, convergence might be slow. An improved planner, called RRT-GoalBias, can be obtained by replacing `RANDOM_CONFIG` in Figure 4.2 with a function that tosses a biased coin to determine what should be returned. If the coin toss yields “heads”, then q_{goal} is returned; otherwise, a random configuration is returned. Even with a small probability of returning heads (such as 0.05), RRT-GoalBias usually converges to the goal much faster than the basic RRT. If too much bias is introduced; however, the planner begins to behave like a randomized potential field planner that is trapped in a local minimum. An improvement called RRT-GoalZoom replaces `RANDOM_CONFIG` with a decision, based on a biased coin toss, that chooses a random sample from either a region around the goal or the whole configuration space. The size of the region around the goal is controlled by the closest RRT vertex to the goal at any iteration. The effect is that the focus of samples gradually increases around the goal as the RRT draws nearer. This planner has performed quite well in practice; however, it is still possible that performance is degraded due to local minima. In general, it seems best to replace `RANDOM_CONFIG`

```

CONNECT( $\mathcal{T}, q$ )
1  repeat
2     $S \leftarrow \text{EXTEND}(\mathcal{T}, q)$ ;
3  until not ( $S = \text{Advanced}$ )
4  Return  $S$ ;

```

Figure 4.4: The CONNECT function.

```

RRT_BIDIRECTIONAL( $q_{init}, q_{goal}$ )
1   $\mathcal{T}_a.\text{init}(x_{init}); \mathcal{T}_b.\text{init}(q_{goal})$ ;
2  for  $k = 1$  to  $K$  do
3     $q_{rand} \leftarrow \text{RANDOM\_CONFIG}()$ ;
4    if not ( $\text{EXTEND}(\mathcal{T}_a, q_{rand}) = \text{Trapped}$ ) then
5      if ( $\text{EXTEND}(\mathcal{T}_b, q_{new}) = \text{Reached}$ ) then
6        Return  $\text{PATH}(\mathcal{T}_a, \mathcal{T}_b)$ ;
7     $\text{SWAP}(\mathcal{T}_a, \mathcal{T}_b)$ ;
8  Return Failure

```

Figure 4.5: A bidirectional RRT-based planner.

with a sampling scheme that draws configurations from a nonuniform probability density function that has a “gradual” bias toward the goal. There are still many interesting research issues regarding the problem of sampling. It might be possible to use some of the sampling methods that were proposed to improve the performance of probabilistic roadmaps [1, 4].

One more issue to consider is the size of the step that is used for RRT construction. This could be chosen dynamically during execution on the basis of a distance computation function that is used for collision detection. If the bodies are far from colliding, then larger steps can be taken. Aside from following this idea to obtain an incremental step, how far should the new configuration, q_{new} appear from q_{near} ? Should we try to connect q_{near} to q_{rand} ? Instead of attempting to extend an RRT by an incremental step, EXTEND can be iterated until the random configuration or an obstacle is reached, as shown in the CONNECT algorithm description in Figure 4.4. CONNECT can replace EXTEND, yielding an RRT that grows very quickly, if permitted by collision detection constraints and the differential constraints. One of the key advantages of the CONNECT function is that a long path can be constructed with only a single call to the NEAREST_NEIGHBOR algorithm. This advantage motivates the choice of a greedier algorithm; however, if an efficient nearest-neighbor algorithm [3, 9] is used, as opposed to the obvious linear-time method, then it might make sense to be less greedy. After performing dozens of experiments on a variety of problems, we have found CONNECT to yield the best performance for holonomic planning problems, and EXTEND seems to be the best for nonholonomic problems. One reason for this difference is that CONNECT places more faith in the metric, and for nonholonomic problems it becomes more challenging to design good metrics.

Bidirectional Planners Inspired by classical bidirectional search techniques [23], it seems reasonable to expect that improved performance can be obtained by growing two RRTs, one from q_{init} and the other from q_{goal} ; a solution is found if the two RRTs meet. For a simple grid search, it is straightforward to implement a bidirectional search; however, RRT construction must be biased to ensure that the trees meet well before covering the entire space, and to allow efficient detection of meeting.

Figure 4.4 shows the RRT_BIDIRECTIONAL algorithm, which may be compared to the BUILD_RRT algorithm of Figure 4.1. RRT_BIDIRECTIONAL divides the computation time between two processes: 1) exploring the sconfiguration space; 2) trying to grow the trees into each other. Two trees, \mathcal{T}_a and \mathcal{T}_b are maintained at all times until they become connected and a solution is found. In each iteration, one tree is extended, and an attempt is made to connect the nearest vertex of the other tree to the new vertex.

Then, the roles are reversed by swapping the two trees. Growth of two RRTs was also proposed in [14] for kinodynamic planning; however, in each iteration both trees were incrementally extended toward a random configuration. The current algorithm attempts to grow the trees into each other half of the time, which has been found to yield much better performance.

Several variations of the above planner can also be considered. Either occurrence of EXTEND may be replaced by CONNECT in RRT_BIDIRECTIONAL. Each replacement makes the operation more aggressive. If the EXTEND in Line 4 is replaced with CONNECT, then the planner aggressively explores the configuration space, with the same tradeoffs that existed for the single-RRT planner. If the EXTEND in Line 5 is replaced with CONNECT, the planner aggressively attempts to connect the two trees in each iteration. This particular variant was very successful at solving holonomic planning problems. For convenience, we refer to this variant as RRT-ExtCon, and the original bidirectional algorithm as RRT-ExtExt. Among the variants discussed thus far, we have found RRT-ExtCon to be most successful for holonomic planning [12], and RRT-ExtExt to be best for nonholonomic problems. The most aggressive planner can be constructed by replacing EXTEND with CONNECT in both Lines 4 and 5, to yield RRT-ConCon. We are currently evaluating the performance of this variant.

Through extensive experimentation over a wide variety of examples, we have concluded that, when applicable, the bidirectional approach is much more efficient than a single RRT approach. One shortcoming of using the bidirectional approach for nonholonomic and kinodynamic planning problems is the need to make a connection between a pair of vertices, one from each RRT. For a planning problem that involves reaching a goal region from an initial configuration, no connections are necessary using a single-RRT approach. The gaps between the two trajectories can be closed in practice by applying steering methods [13], if possible, or classical shooting methods, which are often used for numerical boundary value problems.

Other Approaches If a dual-tree approach offers advantages over a single tree, then it is natural to ask whether growing three or more RRTs might be even better. These additional RRTs could be started at random configurations. Of course, the connection problem will become more difficult for nonholonomic problems. Also, as more trees are considered, a complicated decision problem arises. The computation time must be divided between attempting to explore the space and attempting to connect RRTs to each other. It is also not clear which connections should be attempted. Many research issues remain in the development of this and other RRT-based planners.

It is interesting to consider the limiting case in which a new RRT is started for every random sample, q_{rand} . Once the single-vertex RRT is generated, the CONNECT function from Figure 4.4 can be applied to every other RRT. To improve performance, one might only consider connections to vertices that are within a fixed distance of q_{rand} , according to the metric. If a connection succeeds, then the two RRTs are merged into a single graph. The resulting algorithm simulates the behavior of the probabilistic roadmap approach to path planning [10]. Thus, the probabilistic roadmap can be considered as an extreme version of an RRT-based algorithm in which a maximum number of separate RRTs are constructed and merged.