# From Dynamic Programming to RRTs: Algorithmic Design of Feasible Trajectories

Steven M. LaValle

Department of Computer Science, University of Illinois, Urbana, IL 61801, USA

**Abstract.** This paper summarizes our recent development of algorithms that construct feasible trajectories for problems that involve both differential constraints (typically in the form of an underactuated nonlinear system), and global constraints (typically arising from robot collisions). Dynamic programming approaches are described that produce approximately-optimal solutions for low-dimensional problems. Rapidly-exploring Random Tree (RRT) approaches are described that can find feasible, non-optimal solutions for higher-dimensional problems. Several key issues for future research are discussed.

## 1 Introduction

In robotics, we frequently study the interaction between two elements: *local constraints*, e.g., a nonlinear, underactuated, mechanical system and *global constraints*, e.g., state space constraints that arise from a complicated environment that is often modeled with thousands of piecewise-linear or piecewise-algebraic constraints. Even considering one of these elements in isolation, designing motions automatically that bring a system from an initial state to a goal region represents a formidable research challenge, even if optimality is abandoned. Whether appearing in control literature or algorithms (computer science, robotics) literature, the problem is generally considered as a form of *motion planning*. To emphasize the consideration of both nonlinear systems and complicated environments, this paper refers to the problem as *trajectory design*. If algorithms can be designed that are able to efficiently find and possibly optimize feasible trajectories for broad classes of problems, many application areas would be greatly impacted, including robotics, aeronautics, automotive design, and computer graphics. The successful development of such algorithms will most likely depend on a culmination of ideas from both the algorithms and control communities.

In the algorithms community, focus has been primarily on global constraints, such as computing collision free paths in the presence of complicated, global constraints on the configuration space of one or more movable bodies. In this case, $\dot{x} = u$, and the configuration, $x$, simply represents the set of all rigid or articulated body transformations. It is widely known that the class of problems is NP-hard [50], which has caused the focus of research in this area to move from exact, complete algorithms to sampling-based algorithms that can solve many challenging high-dimensional problems efficiently at the expense of completeness. Within the past decade, sampling-based versions of earlier ideas were developed. The classic notion of a *roadmap* [12,29,48], is a network of collision-free paths that captures the configuration-space topology, and is generated by preprocessing the configuration space independently of any initial-goal query. The most popular sampling-based variation is termed a probabilistic roadmap (PRM) [24], which is formed by selecting numerous configurations at random, and generating a network of paths by attempting to connect nearby points. In contrast to roadmaps, classical *incremental search* ideas are based heavily on a particular initial-goal query, and include methods such as dynamic programming, $A^*$ search, or bidirectional search. Randomized approaches, such as the randomized potential field approach [3], Ariadne's clew algorithm [44], the planner in [23][1], and Rapidly-exploring Random Trees (RRTs) [26], were introduced to handle higher-dimensional planning problems through clever sampling.

A separate but extremely challenging problem is overcoming local constraints, such as the design of open-loop controls that bring an underactuated nonlinear system from an initial state to a goal state or region, even with no global constraints on the state space (e.g., [10,46]). Much of this work is surveyed in [31].

This paper considers the problem of designing trajectories under both local and global constraints on the state space. These problems are often referred to as *nonholonomic planning* [4,30,31] or in the case of systems with drift, *kinodynamic planning* [11,15,16,19,18,20,25,36]. Many of these methods, such as those in [4,18], follow closely the incremental search paradigm because it is generally more challenging to design a roadmap-based algorithm due to the increased difficulty of connecting numerous pairs of states in the presence of differential constraints (often referred to as the steering problem [31]).

Sections 3 and 4 summarize recent approaches to which I have contributed, based on dynamic programming and Rapidly-exploring Random Trees, respectively. Section 5 helps identify interesting directions for future research.

## 2 Generic Problem Formulation

The problem formulation given in this section is intended to provide the general idea; however, additional definitions will be needed in each of Sections 3 and 4.

1. **State Space:** A bounded manifold, $X \subset \Re^n$
2. **Boundary Values:** $x_{init} \in X$ and $X_{goal} \subset X$

[1] We note that the method introduced here is termed a PRM by the authors. Since the method is based on incremental search, it is included here to help categorize the methods based on their conceptual similarities.

3. **Constraint Satisfaction:** A function, $D : X \rightarrow \{true, false\}$, that determines whether global constraints are satisfied from state $x$. This could alternatively be a real-valued function that indicates distance from the constraint boundary. Let $X_{free} \subseteq X$ denote the set of states, $x$, such that $D(x) = true$ (i.e., the constraints are satisfied).
4. **Inputs:** A set, $U(x) \subset \Re^m$, for each $x \in X$, which specifies the set of controls. This set may be finite, as in the case of a quantized system [17,42], or a compact subset of $\Re^m$, for some $m \leq n$.
5. **State Transition Equation:** $\dot{x} = f(x, u)$.
6. **Incremental Simulator:** Given the current state, $x(t)$, and inputs applied over a time interval, $\{u(t')|t \leq t' \leq t + \Delta t\}$, the incremental simulator yields $x(t + \Delta t)$ through numerical integration of $f$ for a fixed input.

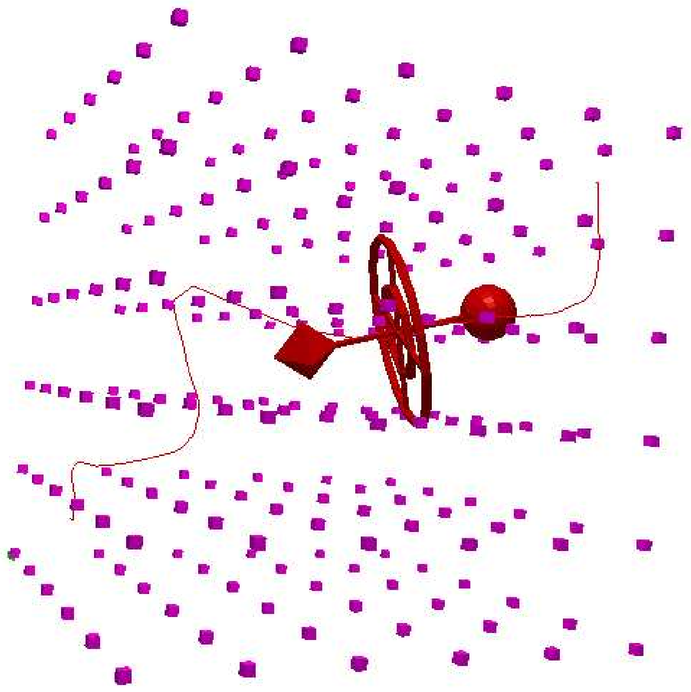

**Fig. 1.** A challenging example: firing three off-center thrusters to bring a spacecraft through an obstacle course.

An example that illustrates the problem formulation is shown in Fig. 1 (this was solved in [14]). The robot is a multiply-connected spacecraft modeled with 869 triangles, and the environment is a collection of 216 small squares, arranged in a grid formation. The robot is considered as a free-floating rigid body in a vacuum. There are three thrusters on the robot, each of which can provide a impulse thrust which yields a force with direction that is not through the center of mass. The task is to compute a sequence of firings for the thrusters that brings the robot from an initial state at rest to goal state at rest, on the other side of the grid.

## 3 Dynamic Programming

Whether in control theory or the design of algorithms, Bellman's principle of optimality [5] has represented a powerful constraint on the set of candidate solutions to an optimization problem. Assume that a discrete-time approximation is made to the original problem described in Section 2. Let $k$ refer to a *stage* or *time step*, and let $x_k$ and $u_k$ refer to the state and input at stage $k$. Let $K+1$ denote the final stage (as described later, this does not have to be explicitly chosen).

Define a *cost functional* of the form

$$L(x_1, \ldots, x_{K+1}, u_1, \ldots, u_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_{K+1}(x_{K+1}), \tag{1}$$

in which $l_{K+1}(x_{K+1}) = 0$ if $x_{K+1} \in X_{goal}$, and $l_{K+1}(x_{K+1}) = \infty$ otherwise. Furthermore, $l$ is a nonnegative real-valued function such that $l(x_k, u_k) = 0$ if and only if $x_k \in X_{goal}$ (one might also require that $u_k$ represents an input that expends no energy in this case). Assume that once the state reaches $X_{goal}$, it remains there until stage $K+1$ without any further cost (other variants are possible, of course).

### 3.1 Classical Cost-to-go Iterations

Classical numerical dynamic programming iterations [5–7,27,28] can numerically solve the problems of interest in this paper. Let the *cost-to-go* function $L_k^* : X_{free} \rightarrow [0, \infty]$ represent the cost if the optimal trajectory is executed from stage $k$ until stage $K+1$,

$$L_k^*(x_k) = \min_{u_k, \ldots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\}. \tag{2}$$

The approach computes representations of the cost-to-go functions iteratively from stage $K+1$ to 1. In each iteration, $L_k^*$ is computed using the representation of $L_{k+1}^*$, by using the following dynamic programming equation, which involves a local optimization over the inputs:

$$L_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + L_{k+1}^*(x_{K+1}) \right\}. \tag{3}$$

Note that $L^*_{K+1} = l_{K+1}$ from (1), which implies that the final stage-dependent cost-to-go function is immediately determined. If $K$ is sufficiently large, then for reasonably-behaved planning problems there exists an $i < K$ such that $L^*_k = L^*_i$ for all $k$ satisfying $k > i$. This will hold for problems in which: 1) all optimal trajectories that reach the goal arrive within a bounded amount of time; 2) infinite cost is obtained for a trajectory that fails to reach the goal; 3) no cost accumulates while the robot "waits" in the goal region; 4) the environment and system are stationary.

The cost-to-go values are computed over a finite set, $P \subset X_{free}$, of *sample points* on which the cost-to-go function $L^*$ is defined through interpolation. In $d$-dimensions, the complexity of classical interpolation is $O(2^d)$; however, in [35], we presented a method based on barycentric subdivision that reduces this to $O(n \lg n)$. The dynamic programming equation (3) is applied at each sample point to compute the next cost-to-go function, $\tilde{L}^*_K$. Each subsequent cost-to-go function is similarly computed. Consider the computation of $\tilde{L}^*_k$. For a finite-input model, the right side of (3) is an optimization over all inputs $u_k \in U$. The values $\tilde{L}^*_{k+1}(x_{k+1})$ are computed by using the incremental simulator and interpolation. For the compact input model, a set of sample points are defined in $U$, and the right side of (3) is an optimization over all inputs $u_k \in U$. When the inputs are tried, the global constraints (e.g., collision checking) can be directly evaluated each time to determine whether each $x_{k+1}$ lies in $X_{free}$.

Note that $L^*_K$ represents the cost of the optimal one-stage strategy from each state $x_K$. More generally, $L^*_{K-i}$ represents the cost of the optimal $(i+1)$-stage strategy from each state $x_{K-i}$. It was assumed that all optimal trajectories require no more than a bounded number of stages before terminating in $X_{goal}$. For a state $x$, let $I(x) \subset P$ denote the set of sample points that are used to compute the cost-to-go for $x$ by interpolation. For the algorithm to succeed, the resolution must be set so that $I(x_{k+1}) \subset X_{goal}$ for sample points near the goal region; otherwise, interpolation with infinity will be attempted, and the algorithm will fail to progress. For a small, positive $\delta$ the dynamic programming iterations are terminated when $|\tilde{L}^*_k(x_k) - \tilde{L}^*_{k+1}(x_{k+1})| < \delta$ for all sample points. Note that no original choice of $K$ was necessary because termination occurs when the cost values have stabilized. The resulting stabilized cost-to-go function, $\tilde{L}^*_1$, can be considered as a representation of the optimal feedback strategy, and is simply denoted as $\tilde{L}^*$.

From any state, $x$, the optimal input in this strategy is obtained by selecting $u_k$ to minimize

$$\tilde{L}^*(x) = \min_{u_k} \left\{ l(x, u_k) + \tilde{L}^*(x') \right\}, \tag{4}$$

in which $x'$ is obtained by applying the incremental simulator to $x$ and $u_k$. In the compact input case, only the inputs that are sample points in $U(x)$ are considered. Starting from any initial state, $x_1 \in X_{free}$, a trajectory can be computed by iteratively applying (4) to compute and apply inputs until termination in $X_{goal}$ is achieved. This results in a sequence of inputs, $u_1, u_2, \ldots, u_k$, and a sequence of states, $x_1, x_2, \ldots, x_k$, in which $x_k \in X_{goal}$. The cost functional (1) can be used to compute the cost of this trajectory. Without numerical error, the cost would be $L^*(x_1)$. Let $\hat{L}^*(x_1)$ denote the actual computed cost by applying $\tilde{L}^*$ to guide the state from $x_1$ to $X_{goal}$.

## 3.2 Improved Algorithms

In each pass over the state space in the method above, most of the values remain unchanged because they either have not been reached, or the optimal value is already known. In [35], we introduced three improved variations that focus the computation only on the active portion of the state space, much in the same way as Dijkstra's algorithm on a graph. The first of the three is summarized here.

For a sample point, $p \in P$, consider the values $\tilde{L}^*_{k+1}(p)$ and $\tilde{L}^*_k(p)$, which are the cost-to-go values at $p$ from iteration $k+1$ to iteration $k$ of the classical algorithm. Note that for any $k \in \{2, \ldots, K+1\}$ and any $p \in P$, the cost-to-go is monotonically nonincreasing, $\tilde{L}^*_k(p) \leq \tilde{L}^*_{k+1}(p)$. If $\tilde{L}^*_k(p) = \tilde{L}^*_{k+1}(p)$, then there are two possible interpretations: 1) $\tilde{L}^*_k(p)$ is infinite,[2] which implies that no trajectories exist which can reach $X_g$ from $p$ in stages $k$ to $K+1$; 2) $\tilde{L}^*_k(p)$ is finite, which implies that the cost-to-go has been correctly computed for $p$, and it will not decrease further in subsequent iterations. For each of these two cases, the costly evaluation of (3) performs no useful work. Furthermore, if $p$ belongs to the second case, it never needs to be considered in future iterations. Let $P_f$ be called the *finalized set*, which is the set of all sample points for which the second condition is satisfied. Let $P_u$ denote the *unreached set*, which is the set of all sample points for which the first condition is satisfied. One more situation remains. If $\tilde{L}^*_k(p) < \tilde{L}^*_{k+1}(p)$, then in iteration $k$ the evaluation of (3) is useful because it reduces the estimate of the true cost-to-go at $p$. Let $P_a$ denote the *active set*, which denotes these remaining sample points. Note that $P_f$, $P_u$, and $P_a$ define a partition of $P$. We assume that $P_a$ is small relative to $P$ in each iteration.

For a set of sample points, $P_1$, let $R(P_1) \subseteq X_{free}$ denote the set of all states, $x$, such that $I(x) \subseteq P_1$. For any subset $C \subset X_{free}$, let $Pre(C)$ denote a *preimage*, which is the set of all $x_{k+1} \in X_{free}$ such that there exists some $u_k \in U$ with $x_{k+1}$ obtained by integration of $f$ over $\Delta t$, starting with $x_k \in C$ and applying input $u_k$. In other words, $Pre(C)$ gives the set of states from which the set $C$ is reachable in a single stage.

Figure 2 shows an algorithm based on preimages that avoids most of the wasted computations of the classical algorithm. Steps 2 to 4 perform a cost-

[2] In practice, a large positive floating point number represents this cost. In this case, the cost-to-go actually increases in each iteration. This does not pose a problem, however, because this is the only case in which an increase can occur, and it is correctly interpreted

```
1  $P_a \leftarrow \{\}$; $P_f \leftarrow P \cap X_{goal}$
2  for each $p \in Pre(R(P_f)) \setminus P_f$
3      Compute $lub(p)$
4      INSERT$(p, P_a)$
5  while $P_a \neq \emptyset$ do
6      for each $p \in P_a$
7          Recompute $lub(p)$
8          if $lub(p)$ is unchanged
9              DELETE$(p, P_a)$
10         if $lub(p)$ is unchanged and finite
11             INSERT$(p, P_f)$
12     $P_a = Pre(R(P_a)) \setminus P_f$
```

**Fig. 2.** This algorithm computes the optimal navigation function while avoiding most of the wasted computations of the classical algorithm.

to-go computation for every sample point outside of $P_f$ that can reach the finalized region in one stage. Step 3 computes and stores the cost-to-go for a sample point using interpolation; this value is referred to as $lub(p)$, which indicates that it represents lowest upper bound on the optimal cost-to-go. Over time, the value is repeatedly updated until $lub(p) = \tilde{L}^*(p)$. Steps 5-12 generate a loop that terminates when $P_a$ is empty. Within each iteration, an updated $lub(p)$ is computed for each $p \in P_a$. If the dynamic programming computation does not change, then one of two possibilities exists: $p$ is finalized or $p$ is unreached. In either case, it should not belong to $P_a$, and is therefore deleted.

The second dynamic programming variation in [35] additionally assumes that the state, $x_{k+1}$, obtained through integration of the state transition equation from $x_k$ over $\Delta t$, always lies in a different interpolation region as $x_k$. In this case, a Dijkstra-like algorithm results, which is able to compute the optimal cost-to-go in a single pass over the state space. The third variation in [35] additionally assumes time optimal solutions are requested, which results in an efficient wavefront propagation algorithm.

## 3.3 Convergence Conditions

One advantage of numerical dynamic programming is that convergence to the optimal solution can be guaranteed for some suitable choice of constants. This section gives conditions such that both $\tilde{L}^*(x)$ and $\hat{L}(x)$ converge to $L^*(x)$ for all $X \in X_{free}$, which are proved in [35], based on extending earlier analysis from [7]. For the *finite input model*, $U(x)$ is finite for all $x \in X_{free}$. Furthermore, it is assumed that there exist positive constants $\alpha_1$ and $\alpha_2$, such that for all $x, x' \in X_{free}$, and for all $u \in U(x) \cap U(x')$,

$$\|f(x,u) - f(x',u)\| \leq \alpha_1 \|x - x'\|$$

and

$$\|l(x,u) - l(x',u)\| \leq \alpha_2 \|x - x'\|.$$

These represent Lipschitz conditions which are needed to establish convergence to the optimal solution in the algorithms.

For the *compact input model*, $U(x)$, is a compact subset of $\Re^m$ for some $m \leq n$. Furthermore,

$$U = \bigcup_{x \in X_{free}} U(x)$$

is compact. It is assumed that there exist positive constants $\alpha_1$ and $\alpha_2$, such that for all $x, x' \in X$, there exists a positive constant $\beta$ such that

$$U(x) \subset U(x') + \{u \mid \|u\| \leq \beta \|x - x'\|\}.$$

It is also assumed that for all $x, x' \in X_{free}$, and for all $u, u' \in U(x)$,

$$\|f(x,u) - f(x',u)\| \leq \alpha_1 (\|x - x'\| + \|u - u'\|)$$

and

$$\|l(x,u) - l(x',u)\| \leq \alpha_2 (\|x - x'\| + \|u - u'\|).$$

Let $d_x$ denote the *dispersion* of the set, $P$, of sample points over $X_{free}$, which is defined as:

$$d_x = \max_{x \in X_{free}} \min_{p \in P} \|x - p\|.$$

Intuitively, the dispersion measures the furthest distance possible in which a state can be placed away from the nearest sample point. For the compact input model, a dispersion, $d_u$, can similarly be defined for a set of samples defined over $U$.

As the number of samples increases, the dispersion becomes smaller. The convergence result is therefore stated in terms of dispersion:

**Proposition 1.** *For classical dynamic programming and the three variants mentioned in this section, there exists a positive constant $\epsilon > 0$ such that*

$$\|L^*(x) - \tilde{L}^*(x)\| < \epsilon \, (d_x + d_u) \quad \forall x \in X_{free}$$

*and*

$$\|L^*(x) - \hat{L}^*(x)\| < \epsilon \, (d_x + d_u) \quad \forall x \in X_{free}.$$

*Under the finite input model, the proposition holds true by setting $d_u = 0$.*

### 3.4 Barraquand and Latombe's Method

To ease the transition to the topic of Section 4, it is interesting to consider a dynamic programming variant introduced in [4]. In contrast to maintaining cost-to-go functions over $X_{free}$, as in the methods above, the approach in [4] is to grow a tree of trajectories. Each vertex of the tree represents a state, and each edge represents a piece of the trajectory over which an input is applied.

The state space, $X$, is partitioned into a rectangular grid in which each element is called a *cell*, to which one of three labels may be applied:

- OBST: The cell contains points in $X \setminus X_{free}$. These are usually precomputed.
- FREE: The cell has not yet been visited by the algorithm, and it lies entirely in $X_{free}$.
- VISITED: The cell has been visited, and it lies entirely in $X_{free}$.

Initially, all cells are labeled either FREE or OBST.

Let $Q$ represent a priority queue in which the elements are states, sorted in increasing order according to $L$, which represents the cost accumulated along the path constructed so far from $x_{init}$ to $x$.

```
FORWARD_DYNAMIC_PROGRAMMING($x_{init}, x_{goal}$)
1  $Q$.insert($x_{init}, L$);
2  $\mathcal{T}$.init($x_{init}$);
3  while $Q \neq \emptyset$ and FREE($x_{goal}$)
4      $x_{cur} \rightarrow Q$.pop();
5      for each $x \in$ NBHD($x_{cur}$)
6          if FREE($x$)
7              $Q$.insert($x, L$);
8              $\mathcal{T}$.add_vertex($x$);
9              $\mathcal{T}$.add_edge($x_{cur}, x$);
10             Label cell that contains $x$ as VISITED;
11 Return G;
```

The algorithm iteratively grows a tree, $\mathcal{T}$, which it rooted at $x_{init}$. The NHBD function tries all inputs, and returns a set of states that can be reached in time $\Delta t$. For each of these states, if the cell that contains it is FREE, then $\mathcal{T}$ is extended. At any given time, there is at most one vertex per cell. The algorithm terminates when the cell that contains the goal has been reached. Convergence of the algorithm is argued in [4] for car-like like robot systems; however, its general convergence remains to be established.

## 4 Rapidly-Exploring Random Trees

The dynamic programming-based methods are limited to state spaces of only a few dimensions. The Rapidly-exploring Random Tree (RRT) was introduced in [33] as 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. Based on RRTs, a randomized, sampling-based approach to trajectory design was introduced in [36]. In that paper, RRTs were applied to trajectory design problems for hovercrafts and rigid spacecrafts that move in a cluttered 2D or 3D environment. Theoretical performance bounds are presented in [37]. In [21], RRTs

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

```
EXTEND($\mathcal{T}, x$)
1 $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x, \mathcal{T}$);
2 if NEW_STATE($x, x_{near}, x_{new}, u_{new}$) then
3     $\mathcal{T}$.add_vertex($x_{new}$);
4     $\mathcal{T}$.add_edge($x_{near}, x_{new}, u_{new}$);
5     if $x_{new} = x$ then
6         Return *Reached*;
7     else
8         Return *Advanced*;
9 Return *Trapped*;
```

**Fig. 3.** The basic RRT construction algorithm.

were applied to the design of collision-free trajectories for a helicopter in a cluttered 3D environment. In [53], RRTs were applied to the design of trajectories for underactuated vehicles. In [25], another tree-based randomized approach to trajectory design was introduced.

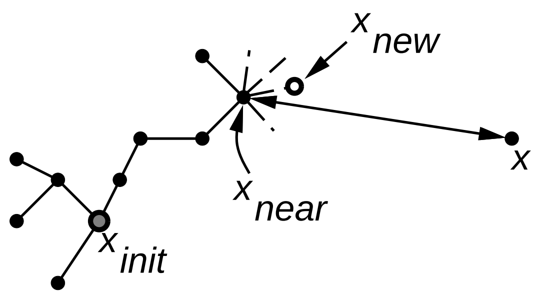

**Fig. 4.** The EXTEND operation.

The basic RRT construction algorithm is given in Fig. 3. 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 state, $x \in X$. The EXTEND function, illustrated in Fig. 4, selects the nearest vertex already in the RRT to $x$. The "nearest" vertex is chosen according to some metric, $\rho$. This can be accomplished naively in linear time, or algorithms that perform near logarithmic time can be applied (see [1] for a discussion of these issues in the context of RRTs).

The function NEW_STATE makes a motion toward $x$ by applying an input $u \in U$ for some time increment $\Delta t$. The input, $u$, 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 checks whether the new state (and all intermediate states) satisfies the global constraints. For many problems, this can be performed quickly ("almost constant time") using incremental distance computation algorithms [22,39,45] 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 first row of Fig. 5 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 [37]. Note that the exploration is naturally biased towards vertices that have larger Voronoi regions. This causes the exploration to occur mostly in the unexplored portion of the state space.

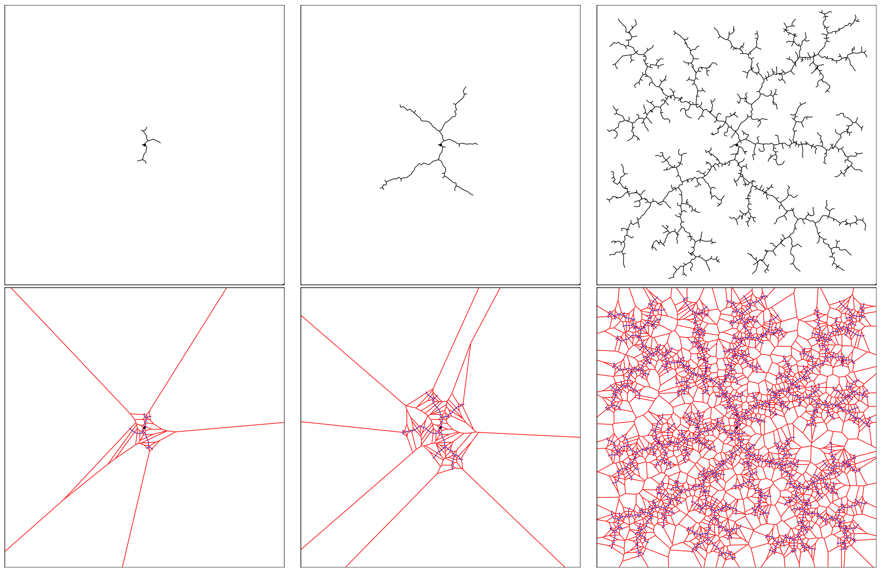

**Fig. 5.** The RRT rapidly explores in the beginning, before converging to the sampling distribution. Below each frame, the corresponding Voronoi regions are shown to indicate the exploration bias.

### 4.1 RRT-Based Trajectory Design Algorithms

In principle, the basic RRT can be used in isolation as a path planner because its vertices will eventually cover a connected component of $X_{free}$, coming arbitrarily close to any specified $x_{goal}$. The problem is that without any bias toward the goal, convergence might be slow. This can be overcome by altering the sampling strategy to concentrate some samples at or near $x_{goal}$.

```
RRT_BIDIR($x_{init}, x_{goal}$)
1 $\mathcal{T}_a$.init($x_{init}$); $\mathcal{T}_b$.init($x_{goal}$);
2 for $k = 1$ to $K$ do
3     $x_{rand} \leftarrow$ RANDOM_STATE();
4     if (EXTEND($\mathcal{T}_a, x_{rand}$) = Trapped) then
5         if (EXTEND($\mathcal{T}_b, x_{new}$) = Reached) then
6             Return PATH($\mathcal{T}_a, \mathcal{T}_b$);
7     SWAP($\mathcal{T}_a, \mathcal{T}_b$);
8 Return Failure
```

**Fig. 6.** A bidirectional RRT-based planner.

The best performance has been obtained so far by conducting a bidirectional search [49] using two RRTs, one from $x_{init}$ and the other from $x_{goal}$; a solution is found if the two RRTs meet. Figure 6 shows the RRT_BIDIR algorithm, which may be compared to the BUILD_RRT algorithm of Fig. 3. RRT_BIDIR divides the computation time between two processes: 1) exploring the state 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.

Through extensive experimentation over a wide variety of problems with state spaces up to twelve dimensions, 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 state, no connections are necessary using a single-RRT approach. The gaps between the two trajectories can be closed in practice by applying steering methods [31], if possible, or classical shooting methods [8], which are often used for two-point boundary value problems.

## 5 Research Challenges

This section compares the algorithms presented in Sections 3 and 4, and identifies key directions for future research. Dynamic programming and RRTs offer complementary advantages, but each has its drawbacks.

### 5.1 Optimality vs. Feasibility

An advantage of the dynamic programming methods is that approximately-optimal solutions are obtained. Even though the dynamic programming equation provides a powerful constraint that significantly reduces the amount of work required to find optimal solutions, it is sometimes much easier to find feasible, but non-optimal solutions. Although much of modern control theory is concerned with optimal decision-making, this requirement is not necessary in many robotics applications. It is certainly desirable in many instances; however, the additional computation cost might outweigh the benefits. Given the complexity of the global constraints considered due to obstacles in the environment, feasibility becomes challenging enough. This philosophy has been followed through most classical algorithmic motion planning work due to the difficulty of finding shortest paths (see the methods in [29]). Algorithms such as RRTs provide solutions that only guarantee feasibility, but they can solve problems that would be too challenging for dynamic programming. In many cases, the solutions can be used in practice after performing some gradient-based optimization techniques (such as those in [8]) on the result, to at least obtain a locally-optimal solution. It remains to be seen whether it is worthwhile to pay the additional cost that appears to be required to obtain optimal solutions, or if there there are ways to avoid this cost.

### 5.2 Random vs. Deterministic Sampling

All of the methods presented in this paper require some form of sampling on the state space. One important issue is whether randomization offers any advantages when applied to sampling strategies. Over the past decade of algorithmic motion planning work, it has been argued by many that randomization is a key to overcoming the curse of dimensionality; however, in related work [9,34], my co-authors and I have shown that deterministic sampling offers performance advantages over random sampling. Excellent overviews of deterministic sampling methods include [43,47]. The key idea is to view sampling as an optimization problem in which a set of points is chosen to optimize some criterion of uniformity (as opposed to a statistical test). One of the most popular measures is discrepancy. Let $\mu(R)$ denote the Lebesgue measure of subset $R$. If the samples in $P$ are uniform in some ideal sense, then it seems reasonable that the fraction of these samples that lie in any subset $R$ should be roughly $\mu(R)$ (divided by $\mu(X)$, which is simply one). Suppose $X = [0,1]^n$. Define the *discrepancy* [54] to measure how far from ideal the point set $P$ is:

$$D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \left| \frac{|P \cap R|}{N} - \mu(R) \right| \tag{5}$$

in which $|\cdot|$ applied to a finite set denotes its cardinality, and $\mathcal{R}$ denotes the set of all axis-aligned rectangular subsets of $X$.

At first glance, the progression from deterministic to randomized, and then back to deterministic might appear absurd; thus, some explanation is required. There appear to be two prevailing reasons for the preference of randomized methods over classical deterministic techniques: 1) they fight the curse of dimensionality by allowing a problem to be solved without prior, systematic exploration of all alternatives; 2) if the "problem maker" is viewed as an opponent in a game, then one can often avoid defeat by employing a random strategy (imagine defeating a deterministic strategy by designing a problem that causes worst-case performance).

The first reason is often motivated by considering that a grid with a fixed number of points per axis will require a number of points that is exponential in the dimension of the space. However, this result is not the fault of grids or even deterministic sampling. It was proved long ago by Sukharev [51] that *any* sampling method that constructs a good covering of the space requires an exponential number of samples. The "goodness" is in terms of point dispersion, as defined in Section 3.3, but using an $\ell^\infty$ metric. We believe the explanation for good performance of path planners in solving challenging high-dimensional problems is that they are able to either exploit some greedy heuristics and/or find solutions to easier problems early by using low-resolution sampling. These benefits are independent of the issue of randomization versus determinism. Thus, multiresolution, deterministic sampling methods are certainly worth exploring in this context. We have already taken steps in this direction in [40]. Randomization appears to be useful in the RRT because it avoids the expensive computation of Voronoi region volumes; however, it may be possible to construct a practical derandomized version.

The second reason (defeating an opponent) might be valid in the case of "true" random numbers; however, any machine implementation generates a deterministic sequence of pseudo-random numbers. These numbers are designed to meet performance criteria that are based on uniform probability densities; however, once it is understood that these numbers are deterministic and being used to solve a particular task, why not design a deterministic sequence that can solve the task more efficiently, instead of worrying about statistical closeness to a uniform probability density? Even if we suppose that true random numbers exist, it seems unlikely that practical examples drawn from applications will contain state space constraints that are designed to break a specific deterministic sampling strategy. Furthermore, randomization can even be introduced back into a deterministic sampling strategy for precisely the reason of fooling an adversary while still maintaining quasi-random sample distribution properties that are superior to pseudo-random sampling [43].

### 5.3 Convergence Conditions

For the algorithms presented in this paper, there are many interesting issues regarding conditions that guarantee convergence to a solution, if a solution exists. In the case of dynamic programming, it is additionally important to show that optimality is obtained. When considering dynamic programming as an approximation of the cost-to-go function, it is possible to guarantee convergence by specifying Lipschitz conditions. In the case of the dynamic programming algorithm by Barraquand and Latombe from Section 3.4, it is not clear whether convergence can be guaranteed if the algorithm is applied to nonlinear systems other than those intended by the authors. In general, for convergence of the algorithms, it appears that it would be useful to have multiple, independent conditions of convergence for a given algorithm. For example, suppose a trajectory design algorithm is known to converge if either a Lipschitz condition is met, or if the system is controllable. It might be possible to specify different rates of convergence depending on which of the two conditions are met.

### 5.4 Designing Metrics

The primary drawback with the RRT-based methods is the sensitivity of the performance on the choice of the metric, $\rho$. In [13], an RRT variant was introduced that exhibits less sensitivity, but the problem still remains in many instances. All of the results presented [38] were obtained by assigning a simple, weighted Euclidean metric for each model (the same metric was used for different collections of obstacles). Nevertheless, we observed that the computation time varies dramatically for some problems as the metric is varied. This behavior warrants careful investigation into the effects of metrics.

In general, we can characterize the ideal choice of a metric (technically this could be called a pseudometric due to the violation of some metric properties). Consider a cost or loss functional, $L$, defined as

$$L = \int_0^T l(x(t), u(t))dt + l_f(x(T)).$$

As examples, this could correspond to the distance traveled, the energy consumed, or the time elapsed during the execution of a trajectory. The optimal cost to go from $x$ to $x'$ can be expressed as

$$\rho^*(x, x') = \min_{u(t)} \left\{ \int_0^T l(x(t), u(t))dt + l_f(x(T)) \right\}.$$

Ideally, $\rho^*$ would make an ideal metric because it indicates "closeness" as the ability to bring the state from $x$ to $x'$ while incurring little cost. For holonomic planning, nearby states in terms of a weighted Euclidean metric are easy to reach, but for nonholonomic problems, it can be difficult to design a good metric. The ideal metric has appeared in similar contexts as the nonholonomic metric (see [31]), the value function [52], and the cost-to-go function [2,32]. In [21] excellent performance was obtained for designing helicopter trajectories by using the cost-to-go for a hybrid system based on trim trajectories as the RRT metric. Of course, computing $\rho^*$ is as difficult as solving the original planning problem! It is generally useful, however, to consider $\rho^*$ because the performance of RRT-based planners seems to generally degrade as $\rho$ and $\rho^*$ diverge. An effort to make a crude approximation to $\rho^*$, even if obstacles are neglected, will probably lead to great improvements in performance.

### 5.5 Constructing Motion Primitives

For the most challenging problems, many challenges remain. For example, there is currently great interest in the design of motions of humanoid robots and also human characters in computer graphics. These problems typically have state spaces of more than 100 dimensions. One reasonable way to deal with the complexity of these systems is to build a family of higher-level motion primitives. A single primitive might be applicable over a large region of the state space due to Lie group symmetries. As an example, the work in [21] develops a hybrid system for designing helicopter trajectories by sequencing trim trajectories. Each trim trajectory is an input function that would typically be applied by a pilot, and can be considered as a mode of the system. If certain conditions are met, it is possible to execute a transition into another mode. When constructing higher-level primitives, it is important to ensure that as much as possible of the expressiveness of the original system is preserved. In the case of [21], it was established that the resulting system is controllable (although not in the presence of global constraints on $X$). In graphics work, researchers have begun to develop libraries of motion primitives for human characters, and are designing search algorithms that sequence these motions [41]. These efforts represent a promising direction of research for handling larger systems that arise in robotics.

## 6 Conclusions

A summary of our approaches to the trajectory design problem in robotics was presented. This problem considers both local and global constraints on the state space. Approximately-optimal dynamic programming approaches and heuristic-based Rapidly-exploring Random Tree approaches were discussed. Although there have been several successes, many interesting and challenging questions remain in this area.

**Acknowledgments** This work was funded in part by NSF CAREER IRI-9875304 and NSF IIS-0118146. I am grateful for the help of Michael Branicky, Peng Cheng, Prashanth Konkimalla, James Kuffner, Andrew Olson, and Libo Yang.

## References

1. A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 632–637, 2002.
2. T. Başar and G. J. Olsder. *Dynamic Noncooperative Game Theory.* Academic Press, London, 1982.
3. J. Barraquand, B. Langlois, and J. C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Trans. Syst., Man, Cybern.*, 22(2):224–241, 1992.
4. J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.
5. R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.
6. R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, Princeton, NJ, 1962.
7. D. P. Bertsekas. Convergence in discretization procedures in dynamic programming. *IEEE Trans. Autom. Control*, 20(3):415–419, June 1975.
8. J. T. Betts. Survey of numerical methods for trajectory optimization. *J. of Guidance, Control, and Dynamics*, 21(2):193–207, March-April 1998.
9. M. Branicky, S. M. LaValle, K. Olsen, and L. Yang. Quasi-randomized path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 1481–1487, 2001.
10. F. Bullo. Series expansions for the evolution of mechanical control systems. *SIAM J. Control and Optimization*, 40(1):166–190, 2001.
11. J. Canny, A. Rege, and J. Reif. An exact algorithm for kinodynamic planning in the plane. *Discrete and Computational Geometry*, 6:461–484, 1991.
12. J. F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.
13. P. Cheng and S. M. LaValle. Reducing metric sensitivity in randomized trajectory design. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, pages 43–48, 2001.
14. P. Cheng and S. M. LaValle. Resolution complete rapidly-exploring random trees. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 267–272, 2002.
15. M. Cherif. Kinodynamic motion planning for all-terrain wheeled vehicles. In *IEEE Int. Conf. Robot. & Autom.*, 1999.
16. C. Connolly, R. Grupen, and K. Souccar. A Hamiltonian framework for kinodynamic planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA'95)*, Nagoya, Japan, 1995.
17. D. F. Delchamps. Stabilizing a linear system with quantized output record. *IEEE Trans. Autom. Control*, 35(8):916–926, 1990.

18. B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning: Robots with decoupled dynamics bounds. *Algorithmica*, 14(6):443–479, 1995.
19. B. R. Donald, P. G. Xavier, J. Canny, and J. Reif. Kinodynamic planning. *Journal of the ACM*, 40:1048–66, November 1993.
20. Th. Fraichard and C. Laugier. Kinodynamic planning in a structured and time-varying 2d workspace. In *IEEE Int. Conf. Robot. & Autom.*, pages 2: 1500–1505, 1992.
21. E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance and Control*, 25(1):116–129, 2002.
22. L. J. Guibas, D. Hsu, and L. Zhang. H-Walk: Hierarchical distance computation for moving convex bodies. In *Proc. ACM Symposium on Computational Geometry*, pages 265–273, 1999.
23. D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. Comput. Geom. & Appl.*, 4:495–512, 1999.
24. L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. & Autom.*, 12(4):566–580, June 1996.
25. R. Kindel, D. Hsu, J.-C. Latombe, and S. Rock. Kinodynamic motion planning amidst moving obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 2000.
26. J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 995–1001, 2000.
27. R. E. Larson. A survey of dynamic programming computational procedures. *IEEE Trans. Autom. Control*, 12(6):767–774, December 1967.
28. R. E. Larson and J. L. Casti. *Principles of Dynamic Programming, Part II.* Dekker, New York, NY, 1982.
29. J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, MA, 1991.
30. J.-P. Laumond. Finding collision-free smooth trajectories for a non-holonomic mobile robot. In *Proc. Int. Joint Conf. on Artif. Intell.*, pages 1120–1123, 1987.
31. J. P. Laumond, S. Sekhavat, and F. Lamiraux. Guidelines in nonholonomic motion planning for mobile robots. In J.-P. Laumond, editor, *Robot Motion Plannning and Control*, pages 1–53. Springer-Verlag, Berlin, 1998.
32. S. M. LaValle. *A Game-Theoretic Framework for Robot Motion Planning.* PhD thesis, University of Illinois, Urbana, IL, July 1995.
33. S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.
34. S. M. LaValle and M. S. Branicky. On the relationship between classical grid search and probabilistic roadmaps. In *Proc. Workshop on the Algorithmic Foundations of Robotics (to appear)*, December 2002.
35. S. M. LaValle and P. Konkimalla. Algorithms for computing numerical optimal feedback motion strategies. *International Journal of Robotics Research*, 20(9):729–752, September 2001.
36. S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 473–479, 1999.
37. S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Workshop on the Algorithmic Foundations of Robotics*, 2000.

38. S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pages 293–308. A K Peters, Wellesley, MA, 2001.
39. M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Int. Conf. Robot. & Autom.*, 1991.
40. S. R. Lindemann and S. M. LaValle. Incremental low-discrepancy lattice methods for motion planning. In *Submitted to IEEE International Conference on Robotics and Automation*, 2003.
41. C. K. Liu and Z. Popovic. Synthesis of complex dynamic character motion from simple animations. In *SIGGRAPH*, 2002.
42. A. Marigo, B. Piccoli, and A. Bicchi. Reachability analysis for a class of quantized control systems. In *Proc. IEEE Conf. on Decision and Control*, 2000.
43. J. Matousek. *Geometric Discrepancy.* Springer-Verlag, Berlin, 1999.
44. E. Mazer, G. Talbi, J. M. Ahuactzin, and P. Bessière. The Ariadne's clew algorithm. In *Proc. Int. Conf. of Society of Adaptive Behavior*, Honolulu, 1992.
45. B. Mirtich. V-Clip: Fast and robust polyhedral collision detection. Technical Report TR97-05, Mitsubishi Electronics Research Laboratory, 1997.
46. R. M. Murray and S. Sastry. Nonholonomic motion planning: Steering using sinusoids. *Trans. Automatic Control*, 38(5):700–716, 1993.
47. H. Niederreiter. *Random Number Generation and Quasi-Monte-Carlo Methods.* Society for Industrial and Applied Mathematics, Philadelphia, USA, 1992.
48. C. O'Dunlaing and C. K. Yap. A retraction method for planning the motion of a disc. *Journal of Algorithms*, 6:104–111, 1982.
49. I. Pohl. Bi-directional and heuristic search in path problems. Technical report, Stanford Linear Accelerator Center, 1969.
50. J. H. Reif. Complexity of the mover's problem and generalizations. In *Proc. of IEEE Symp. on Foundat. of Comp. Sci.*, pages 421–427, 1979.
51. A. G. Sukharev. Optimal strategies of the search for an extremum. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 11(4), 1971. Translated from Russian, *Zh. Vychisl. Mat. i Mat. Fiz.*, 11, 4, 910-924, 1971.
52. S. Sundar and Z. Shiller. Optimal obstacle avoidance based on the Hamilton-Jacobi-Bellman equation. *IEEE Trans. Robot. & Autom.*, 13(2):305–310, April 1997.
53. G. J. Toussaint, T. Başar, and F. Bullo. Motion planning for nonlinear underactuated vehicles using hinfinity techniques. Coordinated Science Lab, University of Illinois, September 2000.
54. H. Weyl. Über die Gleichverteilung von Zahlen mod Eins. *Math. Ann.*, 77:313–352, 1916.