Algorithms for Computing Numerical Optimal Feedback Motion Strategies*

Steven M. LaValle
Dept. of Computer Science
University of Illinois
Urbana, IL 61801 USA

Prashanth Konkimalla
Dept. of Computer Science
Iowa State University
Ames, IA 50011 USA

Abstract

We address the problem of computing a navigation function that serves as a feedback motion strategy for problems that involve generic differential constraints, nonconvex collision constraints, and the optimization of a specified criterion. The determination of analytical solutions to such problems is well beyond the state of the art; therefore, we focus on obtaining numerical solutions that are based on discretization of the state space (although we do *not* force trajectories to visit discretized points). Our work improves classical optimal control techniques for our problems of interest. By introducing a simplicial complex representation, we propose a novel interpolation scheme that reduces a key bottleneck in the techniques from $O(2^n)$ running time to $O(n \lg n)$, in which n is the state-space dimension. By exploiting local structure in the differential constraints, we present a progressive series of three improved algorithms that use dynamic programming constraints to compute an optimal navigation function. Each makes an assumption that is more restrictive than the previous one, and exploits that assumption to yield greater efficiency. These improvements yield a practical increase in the applicability of dynamic programming computations by one or two dimensions over classical techniques. Theoretical convergence to the optimal solution is established for these proposed algorithms. The algorithms have been implemented and evaluated on a variety of problems. Several computed results are presented.

Keywords: Motion planning, algorithms, nonholonomic planning, mobile robotics, navigation functions, dynamic programming.

*Earlier forms of this work were presented in [61] and [47].

1 Introduction

A central problem in robotics is the development of a motion strategy that brings a complex geometric body from an initial state to a goal while satisfying complicated, geometric constraints that model the environment. A classical approach is to decouple this development into a path planning phase and a path following phase. Although this approach works well in many cases, decoupling often leads to inefficient motions. Furthermore, an integrated approach might find solutions to problems that cannot be solved by a decoupled approach.

In this paper, we focus on the problem of computing feedback motion strategies for problems that involve the following assumptions:

1. A complete geometric model is known for the robot and the static part of its environment.
2. The executed trajectory of the robot might not be predictable due to avoidance maneuvers, random disturbances, modeling errors, etc.
3. An optimal strategy is requested that brings the robot from an initial state to a goal region.
4. Differential constraints exist for which there are no analytical solutions to the optimal path problem (e.g., there exist solutions in the case of Reeds-Shepp curves [81]).

A more detailed problem statement appears in Section 3. We allow the strategy to have discontinuous inputs, and do not necessarily require stabilization to a point. Under the restriction to smooth inputs, Brockett's condition implies that time-varying feedback control is needed for point stabilization. A recent overview of such issues for car-like robots is given in [71].

The first condition above is reasonable for many problems, and forms the basis for the vast majority of motion planning research (see, for example, [53]). The second condition motivates us to define a navigation function or potential function that is free of local minima and can be used to define a feedback motion strategy. Navigation functions have been proposed for this purpose in many robotics works (e.g., [43, 83, 90]). The continuous Dijkstra paradigm [34, 75] has been developed for the 2D shortest-path problem, using techniques from computational geometry. Level-set methods have also been proposed to compute navigation functions for motion planning [44]. Although existing methods apply to a broad class of problems, they do not directly apply to our problems because of inclusion of both the third and fourth conditions: *optimality* and *nonholonomy*. Many computational techniques exist for computing optimal solutions to holonomic problems, and many others exist for computing a path for nonholonomic problems. A recent survey of nonholonomic planning methods appears in [56]. Most of these methods do not produce optimal paths, and in general, these methods do not attempt to construct a navigation function or feedback strategy. The method presented in [7] is perhaps most related to ours because it is able to find optimal solutions for very general nonholonomic problems; however, one key distinction is that the method in [7] does not construct a navigation function. Additional related research is described in Section 2.

It turns out that classical numerical dynamic programming techniques [8, 9, 11, 51, 52] can be applied to directly solve the problems of interest in this paper. It is well-known that such methods converge asymptotically to the optimal feedback solution as the resolution increases. These tools were applied previously in [60] to a variety of motion planning problems that involve feedback. The present paper addresses two

drawbacks of these classical methods: 1) the complexity of interpolation is exponential in dimension, 2) for typical robotics problems, most of the computation time is wasted on cost updates that have no effect. Section 4 introduces the basic computational goals and reviews a classical numerical dynamic programming approach to our problem. Section 5 introduces a new interpolation scheme that is based on decomposing the state space into a simplicial complex. This improves interpolation complexity from $O(2^n)$ to $O(n \lg n)$ in an n-dimensional state space. Section 6 presents several algorithms that dramatically focus the dynamic programming computations to obtain much improved performance in the synthesis of an optimal navigation function. The ideas are conceptually similar to the level set methods of [44, 86]; however, our approach is capable of handling differential system constraints in addition to global nonconvex constraints on the state space. Our algorithms are demonstrated in Section 7, and conclusions are presented in Section 8.

Several points are worth keeping in mind regarding this work:

1. We do not claim that the explicit construction of feedback motion strategies is the *only* appropriate way to achieve complicated robotics tasks. The classical decoupled approach of planning a path and then designing a tracking controller is successful in many applications. However, we do argue that a feedback strategy designed at the outset represents a valuable alternative that could solve problems in which tracking is not possible, leads to local minima, or is generally inefficient.

2. Our approach is numerical in the sense that the quality of the solution depends on the resolution. The convergence of our numerical approach to the optimal solution follows from the convergence established in classical works [8, 9, 11, 51, 52]. One key limitation is determining a sufficient resolution. Similar problems exist throughout motion planning research, for example in selecting the resolution of the nonholonomic planner in [7] or determining a sufficient number of milestones in the probabilistic roadmap planner in [41]. We offer no new techniques for determining the best resolution; we instead focus on improving computations for a given resolution.

3. We do not claim that our approach will be practical for more than five or six dimensions. Although the trend in path planning research has been toward high-dimensional problems, it is important to remember that dimensionality does not represent the only challenge in planning. The requirement of feedback, satisfaction of differential constraints, and optimality make our problem substantially more difficult than the classical path planning problem. Randomized planning techniques [4, 6, 32, 35, 41, 49, 64, 74, 95] have been able to solve high-dimensional path planning problems, but it is not known how to apply such ideas to the problem of interest in this paper.

4. In the same way that path planning algorithms are designed for generic configuration spaces and generic geometric models, we have designed our techniques for generic state spaces, generic nonlinear systems, and generic optimality criteria. If simplifying assumptions are made to address specialized problems, we expect that much more efficient approaches could be designed in some cases. For example, the philosophy in system theory and control is to exploit the particular structure of a system to design a better control law. It might be possible to combine particular system structure with our approach; however, we generally make few assumptions regarding the models. We apply a philosophy that is common in motion planning because we want to maximize the scope of application of our algorithms.

2 Related Research

Our work is inspired by research from three distinct areas: 1) nonholonomic path planning, 2) the design of feedback strategies in robotics, and 3) numerical optimal control.

Nonholonomic path planning The nonholonomic path planning problem was introduced in the robotics literature by Laumond [55]. For a detailed review of the research in nonholonomic motion planning, please refer to [56, 68]. The problem involves planning a collision-free path from an initial state to a goal state, while additionally satisfying nonintegrable differential constraints on the state space. Many results from nonlinear control theory are applicable to nonholonomic path planning by interpreting the problem as the design of an open loop control law; some of these results were introduced to the robotics literature in [57, 67].

A classic example of a nonholonomic planning problem involves attempting to parallel park a car that has a limited turning radius. Generally, nonholonomic planning problems have constraints of the form $f(x, \dot{x}) \leq 0$ or $f(x, \dot{x}) = 0$ (and are nonintegrable [36]) in which $\dot{x} = \frac{dx}{dt}$ and x denotes the state or configuration. The differential constraints exist in addition to algebraic constraints that encode stationary obstacles in the workspace. Although much of the work done so far on nonholonomic motion planning has been for car-like robots and extensions that involve trailers, there exists other important examples of nonholonomic robots such as space vehicle/manipulator system [78], dextrous manipulation when the spherical tips of the fingers of the robot's hand perform rolling motions in contact with an object [69], robotic manipulator that pushes objects from one place to another [73], and flying robots actuated by thrusters [72].

The *kinodynamic planning* problem [25] can be considered as a case of nonholonomic planning in which both velocity and acceleration constraints are enforced. One of the earliest algorithms is presented in [85], in which minimum-time trajectories were designed by tesellating the joint space of a manipulator, and performing a dynamic programming-based search that uses time-scaling ideas to reduce the search dimension. Algebraic approaches solve for the trajectory exactly, though the only known solutions are for point masses with velocity and acceleration bounds in one dimension [80] and two dimensions [18]. Approximately-optimal kinodynamic trajectories are computed in [25] by a dynamic programming-based search on the state space by systematically applying control inputs. Other papers have extended or modified this technique [24, 23, 33, 82]. A dynamic programming-based approach to kinodynamic planning for all-terrain vehicles was presented in [19]. In [28], an incremental, variational approach is presented to perform state-space search. An approach to kinodynamic planning based on Hamiltonian mechanics is presented in [20]. An efficient approach to kinodynamic planning was developed by adopting as sensor-based philosophy that maintains an emergency stopping path which accounts for robot inertia [87]. Randomized approaches to kinodynamic planning in high-dimensional state spaces have been proposed in [64], and later in [45].

In nonholonomic planning literature, the *steering problem* has received considerable attention. The task is to compute an open-loop trajectory that brings a nonholonomic system from an initial state to a goal state, without the presence of obstacles. Given the general difficulty of this problem, most methods apply to purely kinematic models (i.e., systems without drift or momentum). To illustrate the challenges of nonholonomic planning, we describe some of the related work on three degree-of-freedom car models that consider only kinematic constraints due to the rolling of the wheels and a limited steering angle. The problem of finding the shortest path between any two states in the absence of obstacles was first addressed by Dubins [26] for

a car moving forward with a constant velocity. He proved the existence of shortest paths and provided a set of six types of curves that contains an optimal path that connects any two states. Reeds and Shepp [81] extended Dubins' result to a car that can move both forward and backward. They proved that the shortest path is always one of 48 types of curves. Each curve is constituted by at most five segments which are either straight or arcs of a circle with minimal radius. Sussmann and Tang [91] later reduced the set to 46 curves. All of the work above shows that the shortest path can be limited to a simple family of trajectories. Souéres and Laumond [88] provided a method to select inside this family an optimal path to link any two states for path planning. All of these results above meet the optimality conditions, but their applicability is limited to car-like robots, or a differential drive robot [5]. Moreover they consider a workspace with no obstacles. For more complicated kinematic models, non-optimal steering techniques have been introduced, which includes a car pulling trailers [77], and firetrucks [17]. Techniques also exist for general system classes, such as nilpotent [50], differentially flat [76, 29], and chained form [17, 77, 89]. For systems with drift and/or obstacles, the steering problem remains a formidable challenge. The series methods introduced in [15] address a broad class of systems of drift.

For a point car that can go only forward (the Dubins car), in the presence of polygonal obstacles, Fortune and Wilfong [30] proposed a complete algorithm for deciding whether there exists a feasible path between any two states. Jacobs and Canny [37] proposed an approximate solution of the complete problem. Exact solutions have been proposed in [3] and [13] if the workspace contains only *moderated* obstacles, which are generalized polygons whose boundaries are admissible paths for the Dubins car. However, if the robot is a polygon (instead of a point) the decision problem is still open.

In [58], a two-phase planner is presented for the Reeds-Shepp car which finds a holonomic path in the first phase and transforms this path into a feasible free path that satisfies the nonholonomic constraints in the second phase. A standard, holonomic path planning algorithm can be used for the first phase. For the second phase, maneuvers as discussed in [55] can be used to transform the path. However, this solution is generally not optimal. Later, a three-phase planner was developed in [38, 92], which used the above approach and also the Reeds and Shepp curves [81]. In the first phase, a holonomic path is generated as in the two-phase planner. In the second phase, the path is recursively decomposed into subpaths until all of the sub-paths can be replaced by collision-free shortest feasible paths. In the third phase, an attempt to optimize the path is done by replacing randomly selected feasible subpaths by collision-free shortest feasible paths. This planner tries to optimize the path, but since the transformations in second and third phases are local, the generated path might be far from optimal.

Most of the results discussed above are specific to a particular nonholonomic system, and are not applicable to other problems. Barraquand and Latombe [7] proposed a planner that is applicable to a wide range of problems. In fact, our solution shares many similarities with this planner. This planner uses the dynamic programming principle to compute the path. The algorithm computes optimal paths and is *asymptotically complete*. That is, for any given problem that admits a solution path, the planner is guaranteed to generate a solution path, provided that the discretization has been set fine enough. A general-purpose planner based on dynamic programming has also been proposed for kinodynamic planning [25]. These approaches suffer from the curse of dimensionality; however, this appears to be the price one must pay to obtain optimal solutions in a very general setting. For example, by sacrificing the requirement of optimality, LaValle and Kuffner

proposed a general planner based on Rapidly-exploring Random Trees (RRTs) [62, 64] for nonholonomic path planning problems in high dimensional spaces; however, the result is a feasible trajectory that satisfies the global constraints, but is not optimal. Note that the general-purpose methods in [7, 25, 64] only generate a path, as opposed to a feedback motion strategy, which is considerably more challenging.

Computing feedback strategies It has been widely recognized throughout the history of control theory that feedback is essential for reliable control of most systems. Generally, a state-feedback law provides a control input for each possible state. In traditional path planning approaches, the problem is decomposed into three parts. First, a collision-free path is planned; then this path is transformed into a trajectory that satisfies velocity bounds. Finally, a feedback control law is applied that will attempt to follow the trajectory as closely as possible.

In the context of robotics, the concept of a real-valued potential function or navigation function has been proposed [42, 83] for encoding feedback motion strategies. The *potential field* approach solves the problem as a whole in one step. This approach was first proposed in robotics by Khatib [42] in his Ph.D. dissertation. Later several approaches were developed to compute the potential function [1, 14, 93]. Most of these techniques suffer from the presence of undesired local minima. Ideally, a potential function should have only one local minimum, which is located at the goal. Consequently, several approaches were proposed to avoid local minima [6, 21, 48]. Rimon and Koditschek introduced potential fields that have only one global minimum and no local minima for a *generalized sphere world* for a point-mass robot (a simplest member of a generalized sphere world family is a space obtained by puncturing a disk by an arbitrary number of smaller disjoint disks that represent obstacles) [83]. Approaches have appeared recently in which a feedback motion strategy is constructed by populating the state space with a collection of neighborhoods over which feedback control laws are defined [16, 94]. Sundar and Shiller developed a potential function with no local minima for a point robot in a workspace that contains circular obstacles [90]. The potential field approach has also been applied to problems in which the obstacles are moving [79, 43].

Once a navigation function is given, a control input is determined by selecting an input that locally reduces the function value. Eventually, the state is driven into the goal region. The particular path executed could vary from trial to trial due to unpredictable deviations; however, this does not pose a problem because the navigation function always provides an input that makes progress toward the goal from any state. An optimal navigation function provides an input that serves as a first step along the optimal path.

Numerical optimal control Since our problem can be characterized in control-theoretic terms, there are natural connections to numerical algorithms that compute optimal feedback control laws. A recent survey of numerical optimal control techniques appears in [12]. The majority of modern techniques, such as collocation, transcription, and multiple shooting are designed for typical optimization problems in control for which substantial nonlinear dynamical constraints exist. Although these approaches include the ability to consider algebraic state-space constraints, the constraints considered in practice are usually simpler than collision constraints that arise in robotics for a complicated robot in a geometrically-complicated environment. The problems considered in our paper contain a substantial motion planning component, in which complicated nonconvex constraints form one of the major challenges, in addition to differential constraints.

Our approach is inspired by the classical dynamic programming framework [8, 9, 11, 51, 52], which is very general, and can be well-adapted to problems with complicated state-space constraints. Several decades ago, the amount of storage required prohibited their use in most realistic applications; however, with greater computation power and their great flexibility, this framework appears quite advantageous. The approach computes a solution in the form of an optimal cost-to-go function, which can be considered in robotics as a navigation function. Note that in addition to these methods, many classical motion planning techniques rely on the dynamic programming principle. About a decade ago, successful dynamic programming approaches to nonholonomic planning were proposed [7, 25] for problems with several degrees of freedom. In contrast to these methods and classical motion planning tools, such as grid-based search and graph search using Dijkstra's algorithm, it is important to note that the methods in [8, 9, 11, 51, 52] are designed to produce control laws over continuous spaces using interpolation. The techniques yield numerical solutions to differential equations over continuous state spaces, as opposed to a graph-based solution that typically arises in path planning research. This difference becomes crucial when both differential constraints and feedback exist in the problem statement.

The continuous dynamic programming techniques were adapted and applied to a variety of robotics problems in [60, 63, 65], and their convergence is shown in [11]. In spite of the high level of generality of these methods, their practical application is limited to problems with only a few dimensions. This motivates the techniques in this paper, which substantially improve the classical approach.

3 Problem Formulation

The problem is defined over a *state space*, X, which is compact and $X \subset \mathbb{R}^n$. A *state*, $x \in X$, could capture a configuration, or both configuration and velocities, of a robot. A set of global, nonconvex constraints exist, which can be combined to yield a *constraint function* of the form $D : X \to \mathbb{R}$. The state will be required to remain in

$$X_{free} = \{x \in X \mid D(x) > 0\},$$

which represents the set of all states that satisfy collision constraints and velocity bounds, if relevant. It is assumed that the interior of X_{free} is am n-dimensional manifold.

A *state transition equation*, $x_{k+1} = f(x_k, u_k)$, models a discrete-time stationary dynamical system, in which k denotes a *stage* (or time step) which occurs at time $t = (k+1)\Delta t$ for some small $\Delta t > 0$. The vector u_k denotes an *input* taken from a state-dependent input space $U(x_k)$. The stages go from $k = 1$ to $k = K + 1$, in which $K + 1$ is a positive integer that represents the *final stage*. It is assumed that $f : X_{free} \times U(x_k) \to X_{free}$, which implies that the state remains in X_{free}.

A compact *goal region*, X_{goal} is given. A sequence of inputs, u_1, u_2, ..., u_K leads to a state trajectory or path, x_1, x_2, x_3, ..., x_{K+1}. To evaluate a trajectory, a *cost functional* is given 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). We assume that once the state reaches X_{goal}, it remains there until stage $K+1$. In the actual mechanics of a system, it might overshoot the goal; however, we consider the goal to be achieved once the requested state is reached. If overshoot is a concern, then a goal region should be chosen which requires the robot to come to rest.

We allow two different models, depending on the number of controls. For the *finite input model*, $U(x)$ is finite for all $x \in X_{free}$. Furthermore, it is assumed that there exist positive constants α_1 and α_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 $\mathbb{R}^m$ for some $m \leq n$. Furthermore,

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

is compact. It is assumed that there exist positive constants α_1 and α_2, such that for all $x, x' \in X$, there exists a positive constant β 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

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

A feedback solution If the task were merely to design an open-loop trajectory, one would only have to specify the inputs, $u_1, \ldots, u_K$, to yield the resulting solution path. We assume, however, that a feedback motion strategy will be required because of unpredictability of the state. For our problems of interest, we generally assume that the feedback strategy is not stage-dependent because our system and constraints are stationary. Let $\gamma : X_{free} \rightarrow U$, which yields an input u_k from any state x_k. The remainder of this paper focuses on the design of an optimal feedback motion strategy by constructing a navigation function. Note that the task of computing a *function of state* is more challenging than computing a *path*.

4 A Classical Dynamic Programming Approach

Numerical dynamic programming approaches to solve the problem introduced in Section 3 have existed for several decades. The foundational work of Bellman [8, 9], the numerical computation algorithms of Larson [51, 52], and the later work of Bertsekas [11] represent some of the seminal works in this area. In this section,

we illustrate how this classical work can be adapted to our problem by interpolation of cost-to-go functions on a grid. It was shown that this method converges to the optimal solution as the resolution increases [11]. The particular approach described in this section is adapted from [60, 63, 65]. It is presented here to introduce the general computational ideas, establish asymptotic convergence to the true optimum, and to provide motivation for the improved methods proposed in Sections 5 and 6.

4.1 Optimal Navigation Functions

The first step is to define a representation of the feedback strategy, $\gamma : X_{free} \rightarrow U$, in terms of a navigation function. In optimal control theory, the *cost-to-go* function has served this purpose. In our context, this is defined as $L^* : X_{free} \rightarrow [0, \infty]$. Each value, $L^*(x)$, yields the cost given by the cost functional (1) under the execution of the optimal trajectory from x until the goal is reached. If termination in the goal cannot be achieved, then $L^*(x) = \infty$.

Note that γ maps to an input, and L^* maps to a real value (or infinity). The feedback strategy requires an input for each state. If L^* is used to encode the feedback strategy, then for a given $x \in X_{free}$, an input is chosen that moves locally in the direction that minimizes L^* over all possible inputs (note that in general, this is not necessarily a gradient descent on L^*). In practice, this is obtained numerically by computing

$$\gamma(x_k) = \min_{u_k \in U} L^*(x_{k+1}) = \min_{u_k \in U} L^*(f(x_k, u_k)), \tag{2}$$

under the assumption that L^* is given. For our problems of interest, we generally assume that L^* is not stage-dependent because our system and constraints are stationary. For clarity in (2), we indicate how to use the cost-to-go to move from stage k to stage $k + 1$.

The principle of optimality Although the cost-to-go will not be stage dependent, it can be computed as the limit of a finite sequence of stage-dependent cost-to-go functions. It will be assumed that for any state from which the goal can be reached, it will be reached after a finite number of stages. The approach is to first consider all zero-stage trajectories, then one-stage trajectories, then two-state trajectories, and so on until the number of considered stages is larger than the number of stages required to reach the goal optimally from any reachable initial state.

In the description of the algorithm, a cost-to-go function will be defined for each stage k. The cost-to-go function $L_k^* : X_{free} \rightarrow [0, \infty]$ represents the cost if the optimal trajectory is executed from stage k until stage $K + 1$, starting at state $x_k \in X_{free}$. Note that $L_{K+1}^* = l_{K+1}$ from (1), which implies that the final stage-dependent cost-to-go function is immediately determined.

The *cost-to-go* function at stage k is given by

$$L_k^*(x_k) = \min_{u_k, \ldots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\}. \tag{3}$$

Equation 3 represents the cost that will be received under the execution of the optimal strategy from stage k to stage $K + 1$.

The cost-to-go can be separated to yield

$$L_k^*(x_k) = \min_{u_k} \min_{u_{k+1}, \ldots, u_K} \left\{ l(x_k, u_k) + \sum_{i=k+1}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\}. \tag{4}$$

The second min does not affect the l term; thus, it can be removed to obtain

$$L_k^*(x_k) = \min_{u_k} \left[l(x_k, u_k) + \min_{u_{k+1}, \ldots, u_K} \left\{ \sum_{i=k+1}^{K} l(x_i, u_i) + l_{K+1}(x_{K+1}) \right\} \right]. \tag{5}$$

The second portion of the min represents the cost-to-go function for stage $k+1$, yielding:

$$L_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + L_{k+1}^*(x_{k+1}) \right\}. \tag{6}$$

This final form represents a powerful constraint on the stage-dependent cost-to-go functions. The optimal input at stage k and state x depends only on the cost-to-go values at stage $k+1$. Furthermore, only the particular cost-to-go values that are reachable from the transition equation $x_{k+1} = f(x_k, u_k)$ need to be considered. The dependencies are local; yet, the globally-optimal strategy is characterized.

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}^*. 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 in a finite 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 motion models are stationary.

4.2 Numerical Computations

The cost-to-go is computed numerically over grid points; however, we can construct smooth trajectories that do not necessarily visit the grid points (as in the case of previous kinodynamic planning algorithms [25, 24, 23, 33]). The cost-to-go at any non-grid point is obtained by interpolation of cost-to-go values at nearby grid points. Note that the resulting trajectories will be continuous because the state transition equation is used to generate each step. See Figure 1 for a one-dimensional illustration. Interpolation is used both for computing cost-to-go functions and during execution.

Let $P \subset X_{free}$ denote a set of *sample points* on which the cost-to-go function L^* will be defined. Ordinarily, the sample points are arranged to form a grid. Let $\tilde{L}^*$ denote the approximate cost-to-go, which is computed by the algorithm. For any state $x \notin P$, let $I(x)$ denote the set of sample points which form the vertices of a cube in the grid that contains x (e.g., in three dimensions, $I(x)$ represents the eight corners of a cube). The value $\tilde{L}^*(x)$ is defined by multilinear interpolation of $\tilde{L}$ over each sample in $I(x)$. Note that in an n-dimensional state space, interpolation is performed between 2^n grid neighbors [51, 52].

An optimal strategy can be computed by successively building approximate representations of the cost-to-go functions. The first step is to construct $\tilde{L}_{K+1}^*$. The final term, $l_{K+1}(x_{K+1})$, of the cost functional is directly used to assign values of $\tilde{L}_{K+1}^*(x_{K+1})$ at the sample points. Note that there is no error: $\tilde{L}_{K+1}^* = L_{K+1}^*$, in which L^* represents the true cost-to-go.

The dynamic programming equation (6) 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 the finite input model, the right side of (6) is an optimization over all inputs $u_k \in U$. The values $\tilde{L}_{k+1}^*(x_{k+1})$ are computed by using interpolation (x_{k+1} is obtained from $x_{k+1} = f(x_k, u_k)$). For the compact input model, a set of sample points are defined in U, and the right side of (6) is an optimization over all inputs

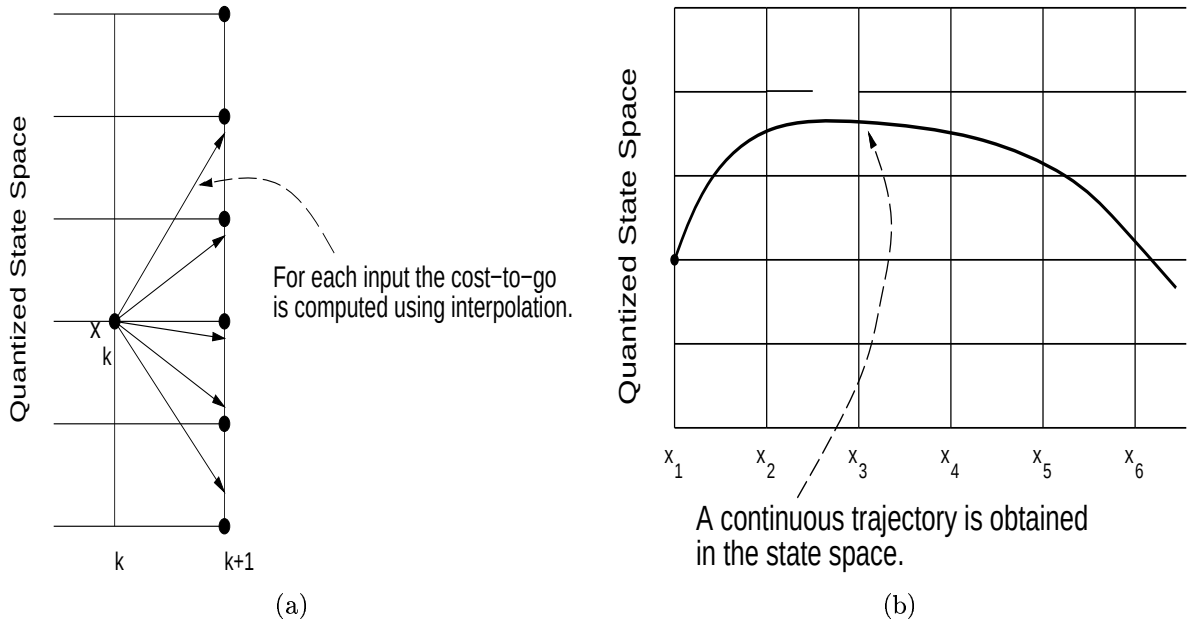


Figure 1: The computations are illustrated with a one-dimensional state space. (a) The cost-to-go is obtained from the next stage by interpolation of the values at the neighboring quantized states. Note that the quantized states do not have to be exactly reached by the state transition equation. (b) During execution, interpolation can also be used to obtain a smooth trajectory.

$u_k \in U$. When the inputs are tried, the constraints can be directly evaluated each time to determine whether each x_{k+1} lies in X_{free}, or a bitmap representation of the state space can be used for quick evaluations. An efficient algorithm for building a bitmap representation of configuration spaces is given in [40].

A numerical technicality exists because of the infinite costs that appear in the cost functional. If $I(x_{k+1})$ contains a sample point, p, such that $\tilde{L}^*_{k+1}(p) = \infty$, then the interpolation would always yield ∞. Numerically, one can test whether $I(x_{k+1})$ contains such a sample point, and declare the resulting interpolation as ∞. Alternatively, a large floating point number can be used to represent numerical infinity, and its value is interpolated.

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 optimal trajectories require only a finite number of stages before terminating in X_{goal}. 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 δ 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 the input, u_k, that minimizes

$$\tilde{L}^*(x) = \min_{u_k} \left\{ l(x, u_k) + \tilde{L}^*(f(x, u_k)) \right\}. \tag{7}$$

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 (7) 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}. Section 4.3 establishes that both $\tilde{L}^*(x)$ and $\hat{L}(x)$ converge to $L^*(x)$ for all $X \in X_{free}$.

4.3 Convergence to the Optimal Solution

This section establishes that as the sampling resolution increases, the error in both the navigation function and in the cost of all resulting trajectories converge to zero. This establishes convergence to the true cost-to-go function over the continuous state space, for a fixed discrete-time representation of a system.

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 dispersions become smaller. The converge result is therefore stated in terms of dispersion:

Proposition 4.1 *For the algorithm in Section 4.2, 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 $d_u = 0$.

The proposition is established by making small modifications to the arguments presented by Bertsekas in [11] based on differences between the two models and algorithms. Rather than repeat several pages of lengthy arguments from that work, we briefly provide an intuitive sketch that indicates the key differences.

The first difference is that the formulation in [11] assumes that $\tilde{L}^*$ is constant over the neighborhood of each sample point, and our formulation uses multilinear interpolation. In general, interpolation provides a closer approximation to L^* than using constant values; thus, convergence still holds.

The second difference is that the original formulation is defined for a time-varying system. Recall our assumption that the optimal trajectory from any reachable state that terminates in X_{goal} in a bounded number of stages. For a given problem, assume that K is chosen large enough so that all optimal trajectories arrive at X_{goal} at or before stage $K + 1$. Let $\tilde{L}_1^*$ represent the resulting cost-to-go after K dynamic programming iterations have been performed. Under our formulation $\tilde{L}^*(x) = \tilde{L}_1^*(x)$ for all $x \in X_{free}$, and it follows from Proposition 2 of [11] that $\tilde{L}^*(x)$ satisfies all of the convergence conditions stated in the proposition above.

The proof in [11], however, requires one additional Lipschitz condition, which in our formulation becomes: there exists a positive constant α_3 such that for all $x, x' \in X_{free}$, and for all $u, u' \in U(x)$,

$$\|l_{K+1}(x,u) - l_{K+1}(x',u)\| \leq \alpha_3 \|x - x'\|,$$

for the finite input model and

$$\|l_{K+1}(x,u) - l_{K+1}(x',u)\| \leq \alpha_3 (\|x - x'\| + \|u - u'\|),$$

for the compact input model. This bounds the slope of the final cost-to-go term; however, in Section 3, l_{K+1} is defined as $l_{K+1}(x) = 0$ if $x \in X_{goal}$ or $l_{K+1} = \infty$ otherwise. This violation of the Lipschitz condition is circumvented by redefining the domain of application for each k (under the time-varying model). Let X_{K+1}, X_K, ..., X_1 denote a sequence of increasing compact subsets of X_{free} in which $X_{K+1} = X_{goal}$. For each k, X_k is defined as the set of all states reachable in a single stage from X_{k-1}. In the algorithm in Section 4.2, inputs were allowed such that $I(x_{k+1})$ contained sample points with infinite value. In the modified formulation, these inputs are no longer permitted because interpolation points are required that are beyond X_k. Once this adjustment has been made, the Lipschitz condition above is satisfied. Using this condition and the Lipschitz conditions stated in Section 3, convergence follows as shown in [11].

4.4 Limitations of the method

The general advantages of these computations were noted long ago in [51]: 1) extremely general types of system equations, performance criteria, and constraints can be handled; 2) particular questions of existence and uniqueness are avoided; 3) a true feedback solution is directly generated. The primary drawback is that the application is limited to only a few dimensions. It is important to note, however, that computing optimal feedback solutions for a problem that involves both differential constraints and complicated geometry is very challenging. Even though randomization has been useful for tackling high-dimensional problems in motion planning [4, 6, 32, 35, 41, 49, 64, 74, 95], it is not clear how such ideas could be applied to obtain solutions to the problem studied in this paper.

Given that the curse of dimensionality represents the primary drawback of the method, we are motivated to at least make the dynamic programming computations as efficient as possible. This can increase the applicability of the method by one or two more dimensions in practice. Given the high level of generality with which the method applies, such improvement is expected to be quite valuable in robotics and other applications.

The next two sections propose two substantial improvements to the dynamic programming computations. Linear interpolation is one of the principle bottlenecks in the computations. Section 5 presents an interpolation scheme that reduces the interpolation computations in n-dimensional space from time $O(2^n)$ to $O(n \lg n)$. Section 6 presents algorithms that compute $\tilde{L}^*$ in a single pass over the state space, as opposed to iteratively computing stage-dependent cost-to-go functions. This leads to much greater performance.

5 Interpolation Based on Complete Barycentric Subdivision

This section proposes an interpolation scheme that can be used to reduce the complexity of each evaluation inside the min function in (6) from time $O(2^n)$ to $O(n \lg n)$, in which n is the dimension of X. In addition to

theoretical savings, the method is efficient in practice. Even though we expect the dimension to be small, the dynamic programming computations become several times faster for problems with only a few dimensions.

5.1 Representing the Cost-to-go Function over a Simplicial Complex

Our main goal is to reduce the computational complexity of computing the cost-to-go function at any given point in X. This is achieved by dividing X into a mesh in which each n-dimensional element is a polytope with the minimum number, $n+1$, of vertices. Each element is called a *simplex*, which is a convex polytope that has $n+1$ vertices. A *simplicial complex* $\mathcal{K}$ [84] is a collection of simplexes in a Euclidean space such that: 1) if a simplex $s \in \mathcal{K}$, then every face of s also belongs to $\mathcal{K}$; 2) if $s, t \in \mathcal{K}$ are any two simplexes, then $s \cap t$ is either empty or a common face of s and t.

While approximating the cost-to-go function over a simplicial complex, we also need to ensure that the problems of identifying the simplex that contains a given state and computing the interpolation coefficients are efficient. The process of computing the cost-to-go at a state x can be divided into three steps:

1. A *point location problem*, which involves identifying the simplex that contains x.

2. Computation of the interpolation coefficients. In the case of a simplex, the interpolation coefficients are *barycentric coordinates* [84], which are non-negative coefficients $\beta_1, \beta_2, \ldots, \beta_{n+1}$ such that $\sum_i^{n+1} \beta_i = 1$. Each state, x, in a simplex can be represented as a linear combination $x = \sum_i^{n+1} \beta_i \, p_i$ in which the p_i are the vertices of the simplex.

3. Computation of the cost-to-go using interpolation. In the case of a simplicial complex, $k = n+1$, the number of vertices in an n-dimensional simplex.

Although the time complexity of the last step is reduced from $O(2^n)$ to $O(n)$ by replacing a grid representation with a simplicial complex, the first two steps are considerably more complicated. These steps must also be efficient to make the simplicial complex approach preferable.

The point location problem would be easy if the simplexes are regular. Representation of the mesh over X as a collection of regular simplexes is possible for the 2D case (a tiling of equilateral triangles). In this case, all the three steps can be carried out very efficiently. However, as shown in [22], it is impossible to tile an n-dimensional space with regular simplexes for $n > 2$. Hence, for higher-dimensional state spaces, we will have to handle irregular simplexes, complicating the point location problem. In this case, the complexity has moved from Step 3 to Step 1. Moreover, the simplexes that we choose should also be applicable (and extensible) for higher-dimensional state spaces.

5.2 Complete Barycentric Subdivision

Our approach starts with a tiling of the state space by cubes. The set of sample points will be that same grid points considered in Section 4. However, each cube is subdivided systematically into simplexes using complete barycentric subdivision, which will be explained shortly.

As shown in [66], an nD cube must be divided into at least $2^n(n+1)^{-(n+1)/2}n!$ simplexes. Thus, a square can be divided into no less than two triangles. Similarly, a 3D cube can be divided into no less than five tetrahedra. It might seem that the point location problem would be straightforward if the number of

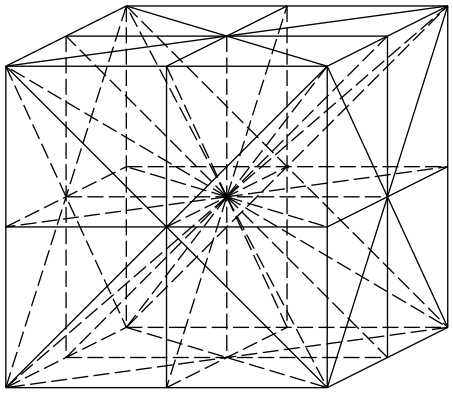


Figure 2: A 3D supercube (which contains 8 cubes) is divided into 48 tetrahedra.

simplexes per nD cube is minimized. However, attempts to reduce the number of simplexes often lead to a more difficult point location problem. This difficulty leads to the proposed approach, which does not produce the optimal number of simplexes, but is very efficient for point location and interpolation.

A tiling of supercubes Suppose that the sample points are arranged in a grid configuration. Let a *supercube* denote the nD cube formed by taking three consecutive sample points along each axis. Each supercube contains 2^n smaller cubes, whose vertices are sample points. Suppose that there are $2m + 1$ sample points per axis. The state space can be partitioned into m^n non-overlapping supercubes in which the vertices of each supercube are sample points. Intuitively, the supercubes are formed by cutting the grid resolution in half along each axis.

Subdividing a supercube The *complete barycentric subdivision* is shown in Figure 2 for a 3D supercube, in which 48 tetrahedra are obtained. The construction places a new vertex at the center of each "feature," in which a feature is either an edge, a face, or the supercube itself. For each edge, the new vertex is connected to the endpoints of the edge. For each face, the new vertex is connected to the corners and the new vertices that came from splitting all of the edges. For the supercube, the new vertex is connected to all existing vertices. Note that the vertex in the center of the cube is common to all tetrahedra. In n dimensions, the subdivisions are extended in a straightforward manner. An nD supercube is divided into $2^n n!$ simplexes. Each nD half-sized cube is divided into $n!$ simplexes, and there are 2^n half-sized cubes within the original cube. It is important to note that we do *not* explicitly store simplexes; only the sample points are stored. The point location problem will simply be a matter of choosing the sample points whose convex hull contains the state.

Voronoi structure Complete barycentric subdivision yields a useful Voronoi structure that greatly facilitates the point location problem. The *Voronoi region* of a set of *features* of a polytope, P, is a set of points interior to P which are closer to that feature than to any feature. If a point, x, lies in the Voronoi region of the feature f, then f is the closest feature to x. In the present context, the set of features for an nD supercube will be the collection of its faces, which are $(n-1)$D supercubes.

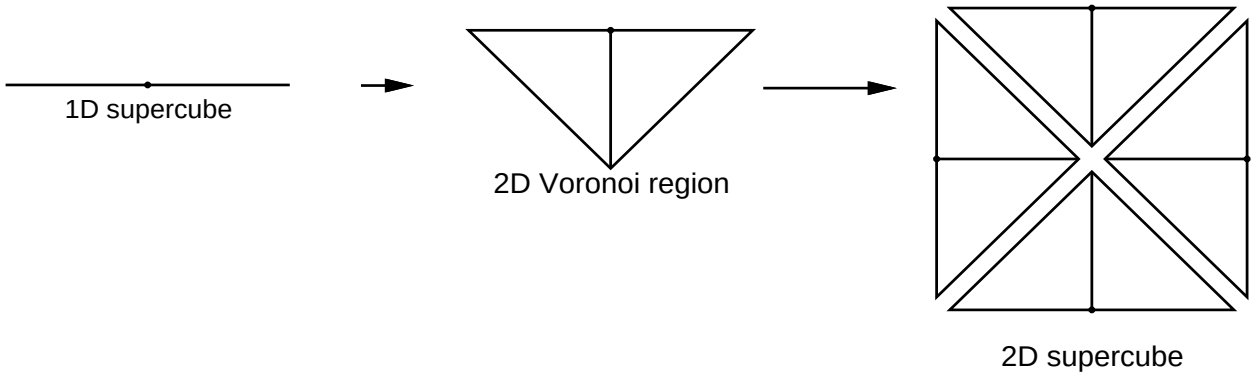


Figure 3: Obtaining a 2D supercube from four 1D supercubes.

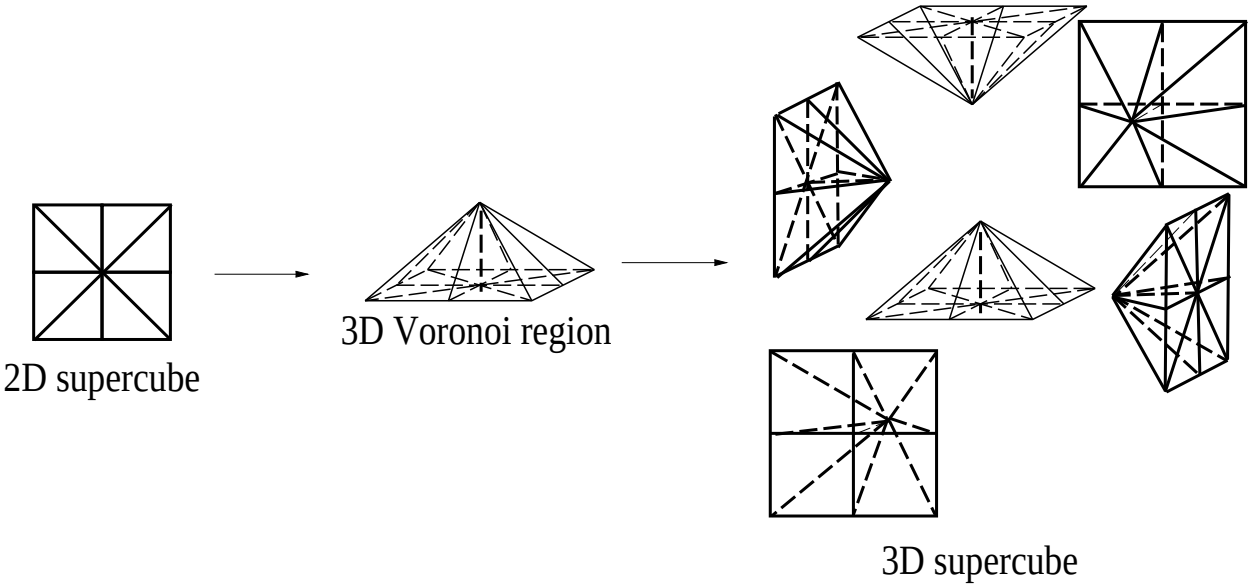


Figure 4: Obtaining a 3D supercube from six 2D supercubes.

Consider an extension from four 1D supercubes to a 2D supercube, shown in Figure 3. The center vertex of the 1D supercube forms a Voronoi boundary that identifies the closest feature (the features are simply the 0D endpoints). Take the center of each 1D supercube, and "lift" it to the center of the 2D supercube. The resulting triangular region is the Voronoi region for the 1D supercube. The dividing segment within the triangular region indicates which vertex of the 1D half-sized cubes is closer. The 2D cube is constructed by "gluing" together four lifted 1D supercubes.

The extension from 2D to 3D is shown in Figure 4, in which six 2D supercubes are lifted and glued to obtain a 3D supercube. By similar constructions, the Voronoi structure for an nD supercube can be obtained by lifting and gluing $2n$ $(n-1)$D supercubes.

Point location problem Suppose a state, x, is given. The first task is to locate the supercube in which x is contained, which can be performed in constant time. The next step is to use the Voronoi structure to quickly determine the simplex that contains x. This involves selecting $n+1$ sample points of the nD

supercube.

Consider the 2D case, and without loss of generality, assume that the 2D supercube that contains x has its four corners at $(-1,-1)$, $(1,-1)$, $(1,1)$, and $(-1,1)$. The task is to determine the three vertices that define the triangle that contains x. The center vertex of the supercube is a vertex of this triangle because it is common to all triangles in the supercube. Let (x_1, x_2) denote the coordinates of x. Consider the four distance quantities $\{|1-x_1|, |1+x_1|, |1-x_2|, |1+x_2|\}$. Each represents the distance from x to an edge of the 2D supercube. Thus, by finding the minimum of these quantities, the 1D supercube nearest to x can be quickly found. The center of this supercube provides the second vertex of the triangle that contains x. The second smallest of the four distance quantities above indicates which endpoint of the 1D supercube is closest to x; this endpoint is the third vertex of the triangle that contains x.

In general, for an n-dimensional supercube with vertex coordinates of the form $(\pm 1, \pm 1)$, there are $2n$ distance quantities:

$$\{|1-x_1|, |1+x_1|, |1-x_2|, |1+x_2|, \ldots, |1-x_n|, |1+x_n|\}.$$

The smallest n of these are used to determine the simplex that contains x. The smallest identifies the closest $(n-1)$D supercube. The second smallest identifies the closest $(n-2)$D supercube within the closest $(n-1)$D supercube. This iteration continues until the closest 0D supercube is obtained, and all $n+1$ vertices of the simplex that contains x are computed. This procedure requires sorting of the $2n$ distance quantities, which can be performed in time $O(n \lg n)$. Thus, it takes time $O(n \lg n)$ to determine the simplex that contains x.

Computation of the barycentric coordinates Once we find the simplex that contains the point, we need to find the barycentric coordinates of the point in its simplex to perform the linear interpolation. Intuitively, each coordinate reflects how close x is to the vertex of the simplex. It turns out that the distance quantities can be used once again.

Assume once again that the nD supercube that contains x has vertices with coordinates of the form $(\pm 1, \pm 1)$. Let p_1, p_2, p_3, ..., p_{n+1} be the vertices of the simplex that contains x. Let β_1, β_2, β_3, and β_4 be the respective barycentric coordinates. Let d_1, d_2, ..., d_n be the n smallest distance quantities, sorted in increasing order, as discussed for the point location problem. The point x must be expressed using barycentric coordinates as $x = \sum_{i=1}^{n+1} \beta_i p_i$. Simple algebraic manipulations reveal that $\beta_1 = 1 - d_1$, $\beta_2 = d_1 - d_2$, ..., $\beta_n = d_{n-1} - d_n$, $\beta_{n+1} = d_n$. Thus, once the distance quantities are sorted, the barycentric coordinate computation requires time $O(n)$. Ordinarily, computing the barycentric coordinates takes time $O(n^2)$ because a linear system is solved; however, our choice of sample points simplifies the problem to $O(n)$.

Computation of cost-to-go Once point location and barycentric coordinate computation have been performed, the cost-to-go, is computed in time $O(n)$. Thus, the following proposition is obtained:

Proposition 5.1 *The method of complete barycentric subdivision solves the point location and interpolation problems in time $O(n \lg n)$, in which n is the dimension of X_{free}.*

Hence, the computation of cost-to-go at any point in space takes $O(n \lg n)$ time, as opposed to $O(2^n)$ time using the grid approach. The improvement is also significant in practice. Even though n is not too large, the scaling constant in the analysis is small.

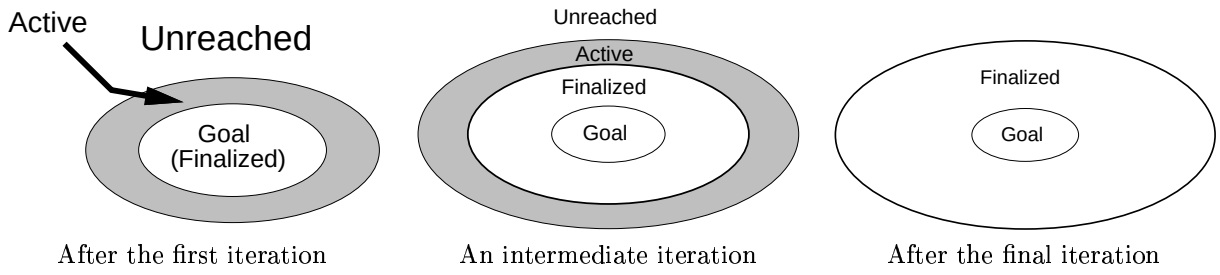


Figure 5: The classical algorithm lowers the cost-to-go values in the active set until the finalized set spans all of the reachable portion of the state space.

6 Algorithms for Computing Optimal Navigation Functions

In this section, improved algorithms for computing the optimal cost-to-go function are given. The concepts are independent of those in Section 5. The algorithms can utilize the original interpolation scheme from Section 4.2 or the improved scheme from Section 5. For this reason, we describe the algorithms in terms of a set, P, of sample points, without regard to particular interpolation issues. Let p denote a state that is a sample point.

To motivate the concepts that follow, recall the classical algorithm from Section 4.2. For a sample point, 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 (recall that the dynamic programming travels backwards through time). Several observations can be made. First, 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,* 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 (6) 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 (6) 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. These sets in the classical algorithm are illustrated in Figure 5.

Sections 6.1 to 6.3 present a series of algorithms that exploit these observations. Each algorithm makes stronger assumptions than the one before, but is able to be more efficient due to the assumptions. The main idea is to focus the computation around P_a.

*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

6.1 An Algorithm Based on Active Sets

In this section, it is assumed that P_a is small relative to P in each iteration. This arises in applications in which the state transition equation causes small changes in state. In other words, $\|x_{k+1} - x_k\| < \epsilon$ for some small $\epsilon > 0$ over all states, $x_k \in X$ and possible inputs $u_k \in U$. For typical robotics problems, ϵ is small relative to the dimensions of X, which leads to a small active set in the classical dynamic programming computations. If P_a is sufficiently small for every dynamic programming iteration, then it is worthwhile to avoid scanning the entire grid each time. Instead the computation can be focused on the active sets.

Using a single cost-to-go function To focus the computation, it would be helpful if the dynamic programming algorithm maintains a single copy of the cost-to-go function. The algorithm in Section 4 defines a cost-to-go function for each k. In practice, however, the cost-to-go $\tilde{L}^*_{k+1}$ can be discarded after iteration k is completed because the dynamic programming computation, (6), is local in time. This implies that only two copies of the cost-to-go are needed at any iteration. Suppose, however, that only a single copy of the cost-to-go function is used. Denote this copy by $\tilde{L}^*$ (without a k subscript). At iteration k, the computation for sample point p is

$$\tilde{L}^*(p) = \min_{u_k} \left\{ l(p, u_k) + \tilde{L}^*(f(p, u_k)) \right\}. \tag{8}$$

Preimage concepts The next concept is used to avoid most of the useless computations from the original algorithm. 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. Using the classical interpolation scheme in an n-dimensional space, there are 2^n sample points in $I(x)$. Using the method from Section 5, there are only $n + 1$ points. For a set of sample points, P_1, let $R(P_1) \subseteq X_{free}$ denote the set of all states, x, such that $I(x) \subseteq P_1$. In other words, R identifies a region over which a cost-to-go could be computed through interpolation of sample points in 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 $x_{k+1} = f(x_k, u_k)$ and $x_k \in C$. In other words, $Pre(C)$ gives the set of states from which the set C is reachable in a single stage. A planning framework based on preimages was introduced in [70], and was also applied in [27, 54, 63].

Algorithm details Figure 6 shows an algorithm based on preimages that avoids most of the wasted computations of the classical algorithm. There is some computational overhead involved in maintaining P_a; thus, the algorithm will become more preferable as the size of P_a is smaller. Step 1 initializes P_a and P_f. The finalized set, P_f, can be computed by performing a scan conversion of the goal region. Steps 2 to 4 perform a cost-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, computed using interpolation and (8); this value is referred to as $lub(p)$, which indicates that it represents lowest upper bound on the cost-to-go. Over time, the value is repeatedly updated until $lub(p) = \tilde{L}^*(p)$, the optimal cost-to-go. Step 4 inserts these sample points into the active set, P_a.

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

```
1    P_a ← {}; P_f ← P ∩ X_goal
2    for each p ∈ Pre(R(P_f)) \ P_f
3        Compute lub(p)
4        INSERT(p, P_a)
5    while P_a ≠ ∅ do
6        for each p ∈ 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)) \ P_f
```

Figure 6: This algorithm computes the optimal navigation function while avoiding most of the wasted computations of the classical algorithm.

deleted. If p is truly in the preimage, then the second case will not be possible; however, in practice it can occur often. Exact preimage computations can be rather complicated, but generating an overapproximation is usually simpler. For example, a spherical region containing the true preimage of each sample point could be used. While the computations can be expedited, the tradeoff is that some sample points might not actually be reachable. Step 9 will also handle this case by deleting them from P_a. Steps 10 and 11 add the sample point to P_f because the optimal cost-to-go has been computed. If the distance traveled by the state transition equation is small relative to the sample point spacing, the $lub(p)$ value might reduce in each iteration, but not converge in a finite number of iterations. In this case, one can set a small numerical threshold, such as 10^{-10}, and declare $lub(p)$ to be unchanged when the change is less than the threshold. In Step 12, new sample points are added to P_a as long as they do not belong to P_f. The algorithm terminates when P_a is empty. This should occur when all reachable sample points have been finalized and added to P_f.

The algorithm only stores $\tilde{L}^*$ for all sample points in P_f. The cost-to-go at any state in $R(P_f)$ can be obtained by linear interpolation, which results in a navigation function that can be used to drive the state into the goal.

Proposition 6.1 *For the algorithm in Figure 6, both $\tilde{L}^*(x)$ and $\hat{L}^*(x)$ converge to $L^*(x)$ for all $x \in X_{free}$ as d_x and d_u approach zero.*

Proof: The first step is to establish that a single cost-to-go function, $\tilde{L}^*$, that can be used for the dynamic programming computations, as opposed to maintaining a copy for each k. The cost-to-go obtained using (8) is never greater than the cost-to-go from the algorithm in Section 4. Some sample points in P have cost-to-go values that reflect iteration k, while others will reflect iteration $k+1$. In the original algorithm, all sample points have cost-to-go values that reflect iteration k. Thus, the value of $\tilde{L}^*$ at each sample point is less than or equal to the value of $\tilde{L}^*_{k+1}$.

Furthermore, the value $\tilde{L}^*(p)$ obtained in the current algorithm is never less than than $\mathrm{L}^*(p)$ as computed by the original algorithm. Since $\tilde{L}^*(x) \leq \tilde{L}^*_k(x)$, the cost-to-go at each sample point, p, is bounded from above by $\tilde{L}^*_{k+1}(p)$ and from below by $\tilde{L}^*(p)$. Proposition 4.1 in combination with the upper and lower bounds imply that the dynamic programming computations converge if a single cost-to-go function is maintained.

1 $Q \leftarrow \{\}$; $P_f \leftarrow P \cap X_{goal}$
2 **for each** $p \in Pre(R(P_f)) \setminus P_f$
3 Compute $lub(p)$
4 INSERT(p, Q)
5 **while** $Q \neq \emptyset$ **do**
6 $p_{min} \leftarrow$POP(Q)
7 **for each** p in $Q \cap (Pre(R(P_f \cup p_{min}) \setminus R(P_f)))$
8 Recompute $lub(p)$
9 INSERT(p_{min}, P_f)
10 **for each** $p \in Pre(R(P_f)) \setminus Q$
11 Compute $lub(p)$
12 INSERT(p, Q)

Figure 7: This algorithm computes the optimal navigation function in a single pass over the state space.

Next, consider the restriction of the dynamic programming computations to the active sets. The first preimage includes all sample points that could possibly lower their value. Sample points in the goal will keep their zero value, and all other sample points will retain their infinite values. The dynamic programming computations over P_a will therefore yield the same result as iterating over all of P. In the next iteration, the preimage again includes all sample points that could possibly lower their value. Thus, in each iteration, the algorithm uses preimages to ensure that the same result will be obtained if all of P were considered instead of focusing the computation on P_a. The convergence as stated in Proposition 4.1 is maintained because the restriction of the dynamic programming computations from P to P_a does not change the computed cost-to-go values. ∎

6.2 A Continuous Dijkstra-like Algorithm with Interpolation

In many cases, the algorithm in Figure 6 can be improved. In addition to the assumption that P_a is small, suppose that that distance from one state, x_k to another x_{k+1} obtained by the state transition equation is more than the grid spacing. For example, all inputs might produce motions that place the new state in a different simplex or grid cell than that of the original state. This is a reasonable assumption for many problems. A similar condition was used in the nonholonomic planning algorithm presented in [7]. In this case, the algorithm in Figure 6 can be improved to obtain an approach that is similar to Dijkstra's algorithm for finding optimal paths from a single vertex in a graph.

The modified algorithm is shown in Figure 7. The first five steps are the same as in Figure 6, except that P_a is replaced by a priority queue, Q. The priority function for each sample point $p \in Q$ is $lub(p)$, and the POP operation removes the point from Q with the smallest value. In this algorithm, the point, $p_{min} \in Q$ with the smallest value can enter the finalized region in a single stage. In this case, p_{min} is immediately finalized.

Steps 5 to 11 are iterated until Q is empty. After each iteration, $\tilde{L}^*$ becomes known for a new sample point. In Step 6, p_{min} is removed from Q (the sample point for which $lub(p)$ is the smallest). It is known that $lub(p_{min}) = \tilde{L}^*(p_{min})$ because a single-stage trajectory exists that brings p_{min} into $R(P_f)$ with less cost than from any other sample point in Q. Once $\tilde{L}^*$ is known for a new sample point, P_f must be appropriately

expanded, which adds new sample points from which $\tilde{L}^*$ can be obtained through interpolation.

To maintain optimality in the next iteration, it is required that the sample point in Q with the smallest cost correctly takes into account all previously finalized sample points. Thus, improved upper bounds are computed in Step 8 for sample points in Q that can reach a region in a single stage that uses p_{min} in the interpolation.

After the costs of these sample points are updated, the next part is to add new elements to Q, which is performed in Steps 9-11. The priority queue, Q, is empty when there are no new sample points that can reach $R(P_f)$ in a single stage.

Proposition 6.2 *For the algorithm in Figure 7, both $\tilde{L}^*(x)$ and $\hat{L}^*(x)$ converge to $L^*(x)$ for all $x \in X_{free}$ as d_x and d_u approach zero.*

Proof: We first argue that in each iteration, p_{min}, is correctly added to P_f. Consider the cost-to-go values at $x_{k+1} \in f(p_{min}, u)$ for each $u \in U(x_{k+1})$ for the finite input model or each sample point in U for the compact input model. For some of these inputs, $x_{k+1} \in R(R_f)$. Let u^* denote the input that minimizes $\tilde{L}^*(x_{k+1})$ over the inputs such that $x_{k+1} \in R(P_f)$. At least one such input exists because p_{min} belongs to the preimage. The cost-to-go values obtained by applying inputs that result in $x_{k+1} \notin R(P_f)$ must be greater than or equal to the cost-to-go obtained by applying u^* because all sample points in P_a have higher cost-to-go than those in P_f. This is similar to the condition that allows Dijkstra's algorithm to obtain shortest paths by selecting the vertex with the smallest cost-to-come at each iteration. Thus, the cost-to-go value computed for p_{min} by using the input u^* must be optimal. This implies that p_{min} can be added to P_f because its value cannot be decreased in future iterations. The addition of p_{min} to P_f extends $R(P_f)$, and consequently $Pre(R(P_f))$ is extended. Thus, new control points are added to Q if they are able to reach $R(P_f)$. This ensures that the correct preimages are maintained.

After each iteration of the algorithm, it must be ensured that the sample point in Q with the lowest cost-to-go has a true $\tilde{L}^*$ value that is less than or equal to the other other sample points in Q. Otherwise, the argument above will not hold, and it might not be possible to add p_{min} to P_f because there are other sample points in Q that could eventually have a lower cost-to-go value than p_{min}. Steps 7 and 8 prevent this problem by recomputing the cost-to-go estimate for all sample points from which an input can take them to a state, x_{k+1}, in which $I(x_{k+1})$ contains p_{min}. This is similar to a step in Dijkstra's algorithm in which the costs or neighboring node are updated after the optimal cost is known for a node.

At each iteration, the correct preimages are maintained and the cost-to-go computed for p_{min} is the cost-to-go $\tilde{L}^*(p_{min})$ which would have been computed by the algorithm in Figure 6. Therefore, convergence of the algorithm in Figure 7 follows from the convergence established in Proposition 6.1. ∎

6.3 A Wavefront Algorithm for Time Optimal Solutions

For the special case in which time-optimal solutions are sought, a more efficient variation of the algorithm in Figure 7 can be obtained. In the case of time optimality, $l(x_k, u_k) = \Delta t$ if $x_k \notin X_{goal}$ and $l(x_k, u_k) = 0$ otherwise. This measures the number of stages that transpire until the trajectory arrives at the goal. Suppose this cost functional is applied in the algorithm in Figure 7. In this case, it is possible to finalize many sample

```
1  P_f ← P ∩ X_goal
2  W ← Pre(P_f) \ P_f
3  while W ≠ ∅ do
4      for each p ∈ W do
5          Compute lub(p)
6          if lub(p) is finite
7              INSERT(p, P_f)
8      W ← Pre(P_f) \ P_f
```

Figure 8: This algorithm computes a time-optimal navigation function in successive waves.

points in the same iteration. This motivates the construction of a "wavefront" of sample points that are added to the finalized region simultaneously, instead of removing each individually from the priority queue. The resulting algorithm is shown in Figure 8.

Step 1 initializes the finalized set, P_f, to the sample points in the goal region. The set W represents the wavefront, which is initialized in Step 2 to the set of all states that can reach P_f in a single stage. Steps 3 to 8 repeat until there are no wavefronts remaining, and P_f spans the reachable portion of the state space. Steps 4-7 process the wavefront. For each $p \in W$, the optimal cost-to-go can be computed immediately. As stated in Section 6.2, the preimage might be an overapproximation in practice for efficiency reasons. For this reason, Step 6 checks to determine whether $lub(p)$ is finite. If it is finite, then $lub(p) = \tilde{L}^*(p)$, which is the optimal cost-to-go, and p is added to P_f. Otherwise, if $lub(p) = \infty$, then p was not in the true preimage. After all of the sample points W have been processed, a new wavefront is generated in Step 8 based on the new finalized set P_f.

Proposition 6.3 *For the algorithm in Figure 8, both $\tilde{L}^*(x)$ and $\hat{L}^*(x)$ converge to $L^*(x)$ for all $x \in X_{free}$ as d_x and d_u approach zero.*

Proof: The arguments for convergence are similar to those of Proposition 6.2. Due to the time-optimality criterion, multiple sample points may be finalized simultaneously. For each $p \in W$, the finalized region $R(P_f)$ can be reached in a single stage. The optimal input must yield the final cost-to-go $\tilde{L}^*(p)$ because if the trajectory first visits $R(W \cup P_u)$, more time will be required in terms of stages before the finalized region is reached. This would fail to optimize the criterion. Thus, for each wavefront computation, the values $\tilde{L}^*(p)$ are correctly computed for each $p \in W$. This implies that the same results are obtained as for the algorithm in Figure 6, and the method algorithm in Figure 8 converges to the optimal solution by application of Proposition 6.1. ∎

7 Implementation and Experiments

To illustrate the concepts in this paper, we apply them to the computation of time-optimal paths for a variety of problems. Our implementation uses GNU C++ and LEDA under Red Hat Linux on a Pentium III 500 Mhz PC (note that current PCs are roughly 3 to 4 times faster). To obtain the results in this section, we implemented the algorithm in Figure 8 using the interpolation scheme introduced in Section 5. Some results obtained from the algorithm in Figure 7 are presented [61]. More experimental results appear in [46].

Table 1: Computational performance from a GNU C++ implementation on a 500Mhz Pentium III PC running Linux. The columns denote: P = problem number, Res = resolutions, Precomp = precomputation time, DP = dynamic programming computation time.

P	Res	Precomp (sec)	DP (sec)
1	$70 \times 70 \times 30$	16.37	9.12
1	$100 \times 100 \times 30$	33.40	15.77
1	$200 \times 200 \times 30$	122.91	72.15
2	$70 \times 70 \times 30$	13.51	9.63
2	$100 \times 100 \times 30$	27.11	14.35
2	$200 \times 200 \times 30$	109.18	56.77
3	$200 \times 200 \times 30$	120.66	68.97
4	$200 \times 200 \times 30$	115.69	53.08
5	$200 \times 200 \times 30$	112.46	110.06

Once the optimal cost-to-go, $\tilde{L}^*$, is computed for the finalized set of sample points, it is used to produce a trajectory from an initial state. For the dynamic programming computations, preimages were computed by selecting all sample points that could be reached in one stage by applying a simple velocity bound to the state. This represents a simple, overapproximation to the true preimage.

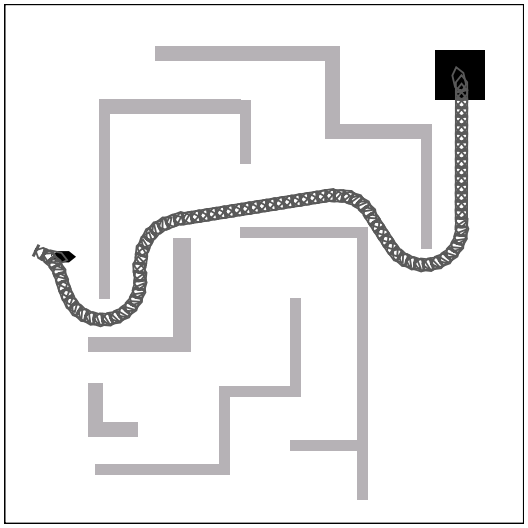
Our approach has been implemented and evaluated on five different types of problems: 1) planning for an ideal car-like robot, 2) planning for a box-pushing robot, 3) planning for a car-like robot with prediction uncertainties, 4) planning for a car-like robot with sensing errors, and 5) a kinodynamic planning problem for a car-like robot. Some of the results are presented in the remainder of this section; more results appear in [46, 61].

Car-like robots For Problems 1 to 5, a polygonal car-like robot is defined in a 2D world that contains polygonal obstacles. The state space is simply the 3D configuration space, $X = \mathbb{R}^2 \times S^1$. Nonholonomic constraints arise due to the rolling contact of the wheels with the ground and also from a limited steering angle.

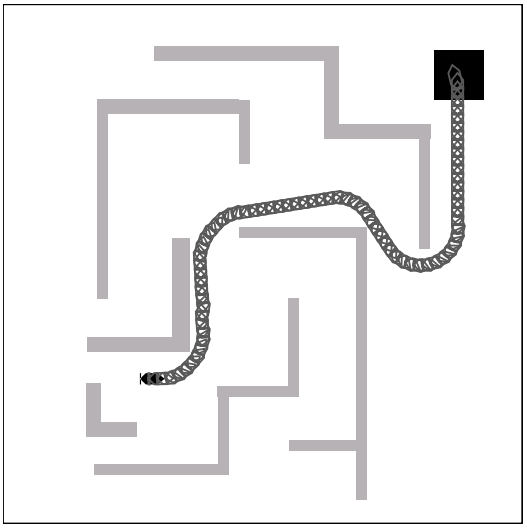
The algorithm is implemented for the Reeds-Shepp car [81] (forward and backward) for three different workspaces. We have also implemented the algorithm for the Dubins car [26] (forward only). A maximum turning angle of 30 degrees is used for all examples. Navigation functions for four problems that involve the Reeds-Shepp car were computed. The results are shown in Figure 9.a-9.e. Figure 9.f shows the trajectories of the optimal navigation function for the Dubins car (forward only).

The computation times are shown for various resolutions in Figure 1. The column labeled "Precomp" refers to the time taken for initialization of a bitmap that is used for rapid collision detection; there exist much faster methods for this computation [39]. These results and the results presented in [46, 61] indicate that the new algorithm is an order of magnitude or two faster in practice than the classical dynamic programming approach applied to similar problems [60, 52].

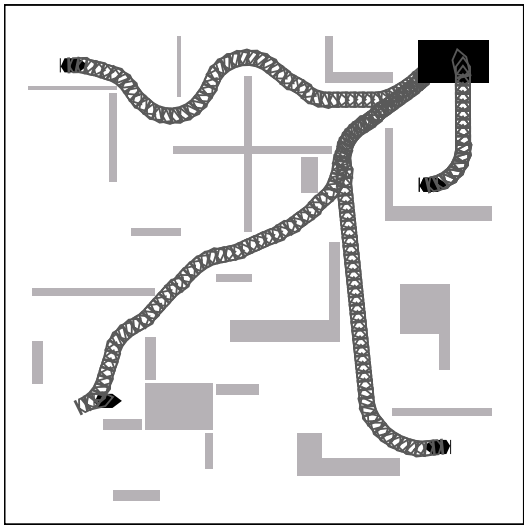
Figure 10 shows the level set contours of L^* for a Dubins car in an environment without obstacles. The orientation is 0, and x and y are variables. This result illustrates that the computed cost-to-go can be considered as a numerical approximation to the *nonholonomic metric* determined in [59, 88].



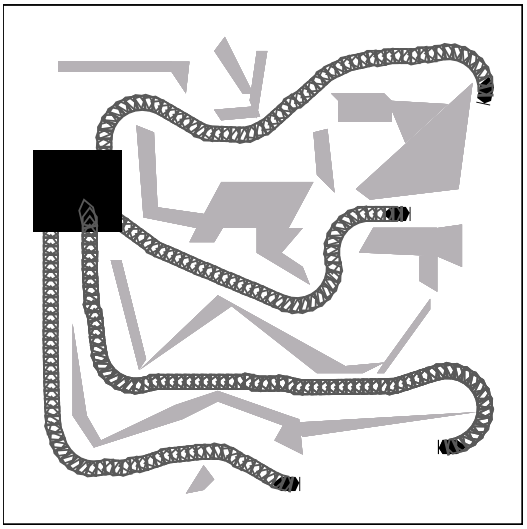
(a) Problem 1



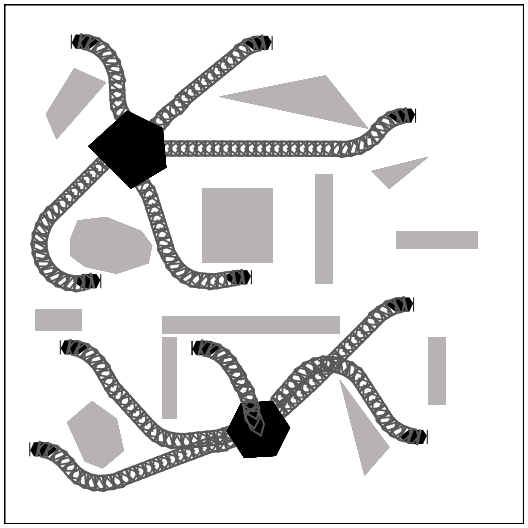
(b) Problem 1



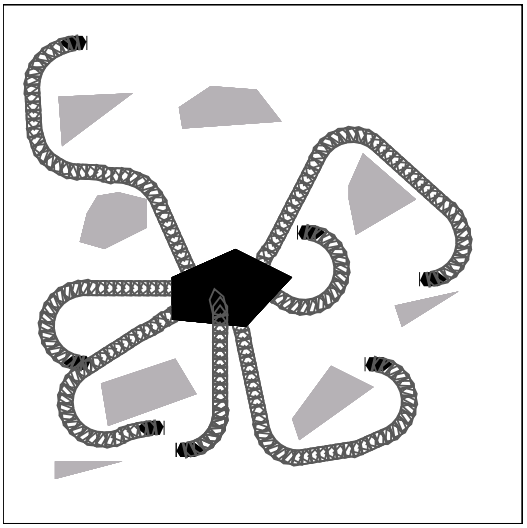
(c) Problem 2



(d) Problem 3



(e) Problem 4



(f) Problem 5

Figure 9: Several computed examples for car-like robots. (a) to (e) involve Reeds-Shepp cars. (a) and (b) show two different paths for the same navigation function. (c) and (d) show several different paths obtained by applying the computed navigation function to different initial states. (e) shows an example in which the goal region is disjoint. (f) involves a Dubins car.

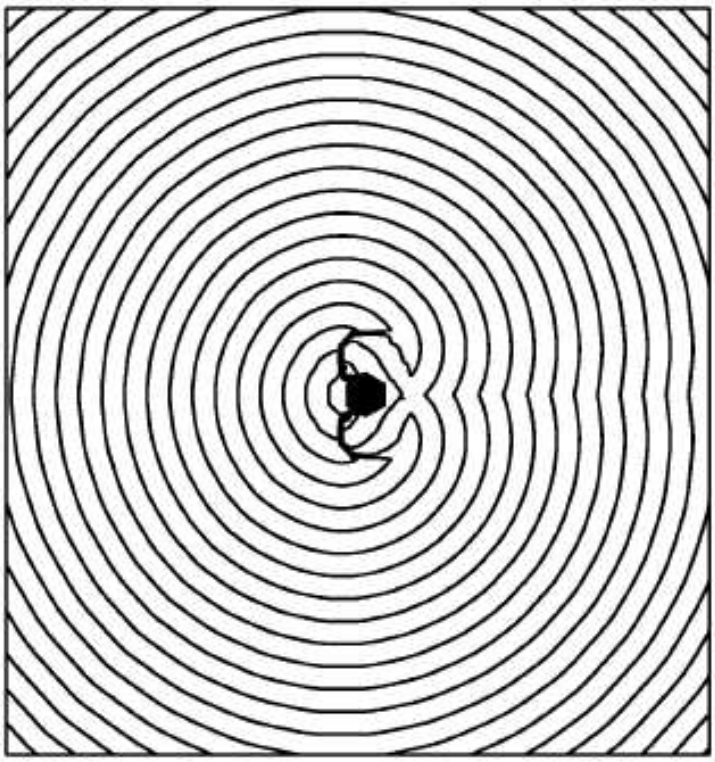
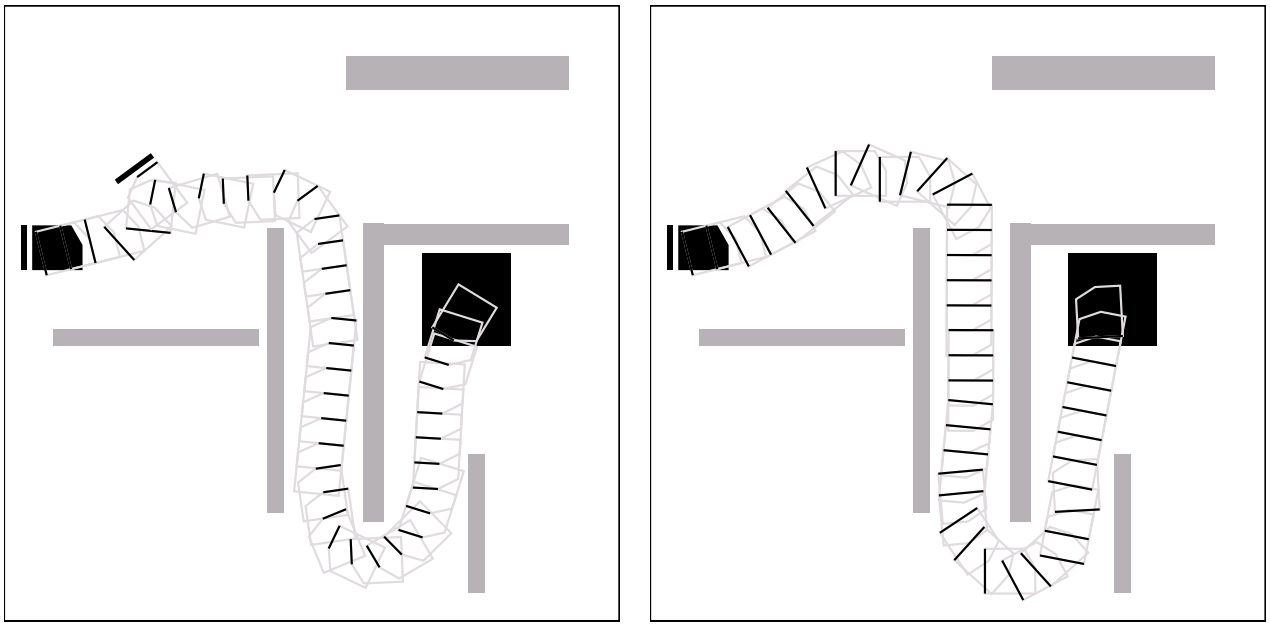


Figure 10: The level sets of the cost-to-go function for a particular orientation of a Dubins car.

Push planning Problems 6 and 7 are based on the push-planning problem introduced in [73]; a similar problem was considered in [2]. A robot must manipulate a polygonal object by pushing it along a flat surface. It is assumed that the robot and the object make contact at a single edge. Instead of grasping the object, the robot must rely on friction forces to prevent the object from sliding against the contact edge between the robot and the object. For a given orientation of the object, a friction model, and a particular edge from which to push, the robot velocities are constrained because the object must not slip. We assume a maximum pushing angle and choose one of two object edges for pushing. We also assume that the robot can switch the pushing edge without hitting the obstacles. The state space is $\mathbb{R}^2 \times S^1$, which is the configuration space of the object.

Computed results are shown in Figures 11 and 12. As defined in Section 3, the cost functional (1) can depend on the input u_k. In the push planning problem, the number of times the robot switches the pushing edge can be optimized (minimized) by associating more cost to inputs that require a change of the pushing edge. Figure 11.a shows the trajectory of the object when time taken by the robot to push the object to the goal is minimized. Figure 11.b shows the trajectory of the object when the number of times the robot switches the pushing edge is minimized. In Figure 11.a, the robot completes the task in 111 time stages, switching edges once. In Figure 11.b the robot takes 120 stages, but does not change the contact edge. Figure 11.c shows the multiple trajectories of the object along with the pushing edge from different initial states for another workspace. Figure 2 shows the computation times for Problems 6 and 7.



(a) Minimal time (b) Minimal reversals

Figure 11: Problem 6 involves a push planning problem under two different criteria.

Table 2: Computational performance from a GNU C++ implementation on a 500Mhz Pentium III PC running Linux. The columns denote: P = problem number, Res = resolutions, Precmp = precomputation time, DP = dynamic programming computation time.

P	Res	Precmp (sec)	DP (sec)
6	$200 \times 200 \times 30$	23.028	34.12
7	$200 \times 200 \times 30$	22.36	37.71
8	$50 \times 50 \times 30 \times 24$	4.95	883.51

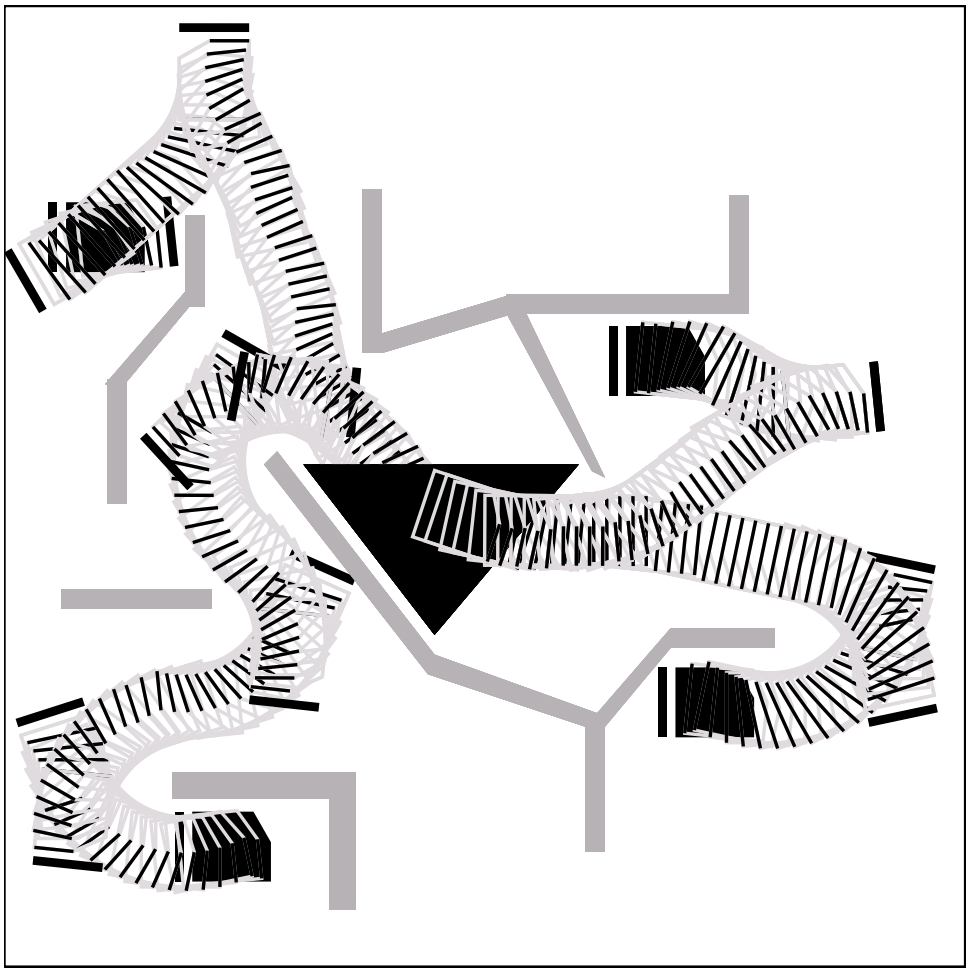
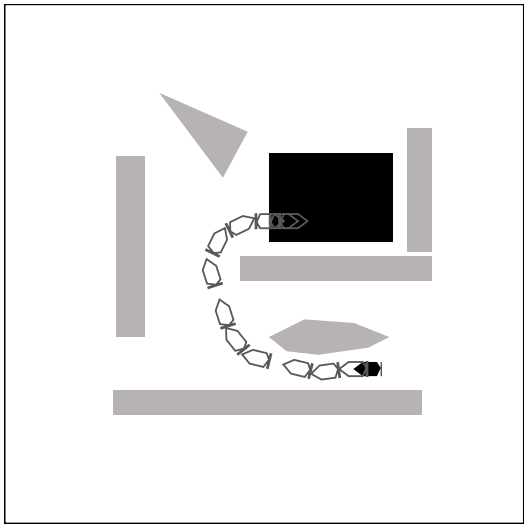


Figure 12: Problem 7 involves a time-optimal push planning problem in which the computed navigation function is applied from different initial states. This results in different trajectories that reach the triangular goal region in the center.

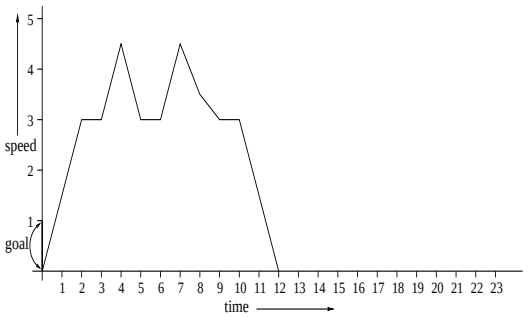
Kinodynamic planning Problem 8 illustrates an optimal navigation function that produces an optimal trajectory with the consideration of a simple dynamical model. This can be considered as a feedback variation of the kinodynamic planning problem as presented in [18, 19, 20, 25, 24, 23, 31, 82]. We assume a maximum speed, a maximum turning speed, and also a maximum speed while backing, for the car. With speed as the fourth state variable, the state space has four dimensions.

We consider a simple four degree-of-freedom model. This model is very simplistic, and is chosen only for illustration purposes. Let f_{max} be the maximum centrifugal force that the car can withstand before laterally slipping or toppling. Let ϕ_{max} be the maximum steering angle (based on mechanical limits), and let s_{max} be the maximum speed the car can go with steering angle ϕ_{max}. Let m be the mass of the car and ℓ, the distance between the front and rear axles of the car. Thus,

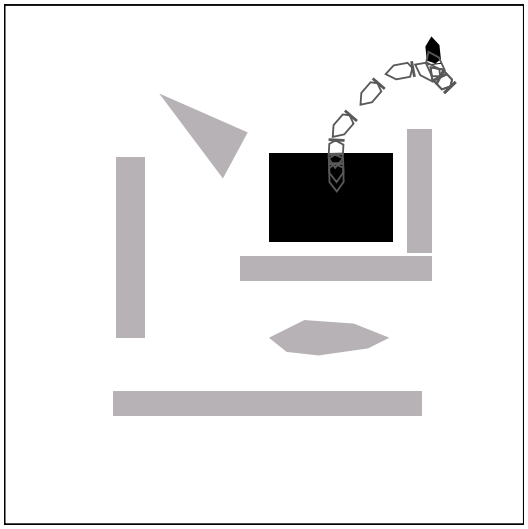
$$f_{max} = \frac{m s_{max}^2 \tan \phi_{max}}{\ell}. \tag{9}$$



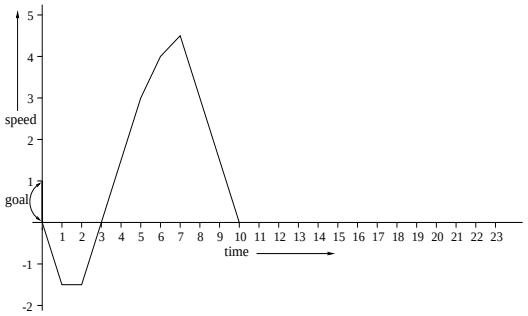
(a)



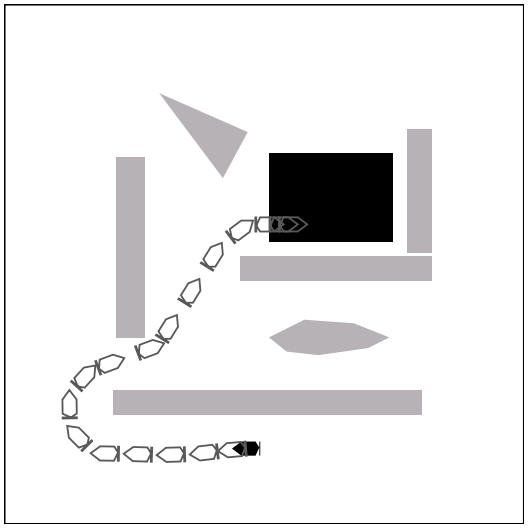
(b)



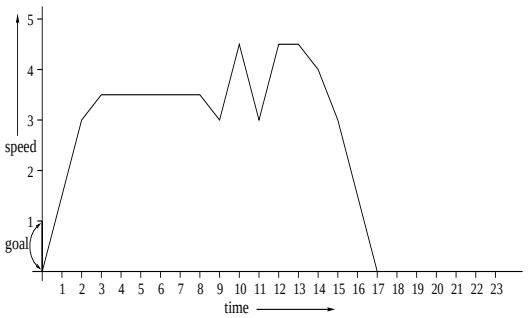
(c)



(d)



(e)



(f)

Figure 13: Problem 8 involves designing time-optimal trajectories for a car with bounded acceleration and velocity-dependent steering limits.

If the car is going at a speed s, then the limit on the steering angle, ϕ, is

$$ms^2\ell \tan\phi \leq f_{max}. \tag{10}$$

Using (9),

$$\phi \leq \arctan(s_{max}^2 \tan\phi_{max}/s^2). \tag{11}$$

Figure 13 shows a computed example. Figures 13 (a), (c), and (e) show the trajectories of the car from different initial states. Figures 13 (b), (d), and (f) show the graphs of the speed of the car as a function of time for the respective trajectories. For the above results, a maximum speed of 4.5 units/sec, a maximum turning speed (s_{max}) of 3.5 units/sec, a maximum speed while backing of 3.0 units/sec, a maximum steering angle (ϕ_{max}) of 45 degrees, and an acceleration (or deceleration) of 1.5 units/sec^2 have been used. Table 2 shows the computation times.

Given the computation time, we believe that the algorithm could also solve five-dimensional kinodynamic problems if one can tolerate a few hours of computation. In this case, the realistic 5-DOF model from [10] could be used, for example, to yield very accurate dynamical simulations of the vehicle. Although several hours might seem costly, there presently exist no general-purpose algorithms which can compute such solutions in the presence of obstacles (other than the classical algorithm from Section 4).

8 Conclusions

We have presented several algorithmic ideas that improve dynamic programming computations in the construction of optimal feedback motion strategies. The two key ideas were to: 1) devise a simplicial complex representation that reduces the complexity of interpolation from $O(2^n)$ to $O(n \lg n)$ for an n-dimensional state space, and 2) focus the dynamic programming computations on the active sample points in the state space. Our approach is for general system equations and environments; however, we expect that their application in practice will be limited to no more than five or six dimensions. Note that it is, however, unfair to compare our approach with recent randomized approaches to path planning that can handle high-dimensional problems. In contrast to the classical path planning problem, in this paper we seek solutions that include: 1) optimality, 2) differential constraints, and 3) feedback. Presently, there exist no general planning algorithms that can meet all of these requirements for high-dimensional state spaces. The algorithmic concepts presented in this paper can at least increase the maximum state space dimension handled in practice by one or two over the maximum state space dimension for the classical dynamic programming approach in Section 4. We successfully demonstrated the techniques through extensive implementation and experimentation. More experiments are presented in [46, 61]. The approach was observed to be much faster than existing dynamic programming techniques that determine an optimal cost-to-go function.

Handling uncertainties The ideas of Sections 5 and 6 can be extended to problems that involve probabilistic uncertainty models. In [46], extensions to the algorithms in this paper are presented for problems that involve either uncertainty in predictability, uncertainty in sensing, or both. For the problems of uncertainty in predictability, the model in [65] was used to derive feedback strategies that are optimal in the expected sense. For problems of both uncertainty in sensing and predictability, the model in [63] was used

to derive feedback strategies that are based on information states. In each of these contexts, we have observed dramatic performance improvement over the approaches in [63, 65]. The convergence proofs of [11] were originally developed for problems that involve probabilistic uncertainty in prediction. Our convergence proofs are derived from this work, and they can be extended to the case of computing the optimal expected cost-to-go in the case of probabilistic uncertainty.

Topics for future work One point that should be given more careful attention in future work is the computation of the preimages. In the present implementation, the computed preimage typically contains many more sample points than necessary. Since the set of all of the sample points in the actual preimage is a subset of the computed preimage, the correctness of the solution is not affected. However, it certainly increases the computation time as some points are considered several times until the stage when they belong to the true preimage arrives. Hence, more accurate preimage computation for particular nonholonomic systems could improve the computation times.

Acknowledgments

We are grateful for the funding provided in part by NSF CAREER Award IRI-9875304 (LaValle). The majority of this work was conducted while both authors were at Iowa State University. We thank the anonymous reviewers for their helpful suggestions in earlier versions of this work.

References

[1] M. D. Adams, H. Hu, and P. J. Roberts. Towards a real-time architecture for obstacle avoidance and path planning in mobile robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 584–589, May 1990.

[2] P. K. Agarwal, J.-C. Latombe, R. Motwani, and P. Raghavan. Nonholonomic path planning for pushing a disk among obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 1997.

[3] P. K. Agarwal, P. Raghavan, and H.Tamaki. Motion planning for a steering constrained robot through moderate obstacles. In *Proc. ACM Symposium on Computational Geometry*, 1995.

[4] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 113–120, 1996.

[5] D. Balkcom and M. Mason. Geometric construction of time optimal trajectories for differential drive robots. In *Preprints of the 4th Workshop on Algorithmic Foundations of Robotics*, Dartmouth College, Hanover, NH, 2000.

[6] J. Barraquand and J.-C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *IEEE Int. Conf. Robot. & Autom.*, pages 1712–1717, 1990.

[7] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.

[8] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[9] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, Princeton, NJ, 1962.

[10] J. Bernard, J. Shannan, and M. Vanderploeg. Vehicle rollover on smooth surfaces. In *Proc. SAE Passenger Car Meeting and Exposition*, Dearborn, Michigan, 1989.

[11] D. P. Bertsekas. Convergence in discretization procedures in dynamic programming. *IEEE Trans. Autom. Control*, 20(3):415–419, June 1975.

[12] J. T. Betts. Survey of numerical methods for trajectory optimization. *J. of Guidance, Control, and Dynamics*, 21(2):193–207, March-April 1998.

[13] J. D. Boissonnat and S. Lazard. A polynomial-time algorithm for computing a shortest path of bounded curvature amidst moderate obstacles. In *Proc. ACM Symposium on Computational Geometry*, pages 242–251, 1996.

[14] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Trans. Syst., Man, Cybern.*, 19(5):1179–1187, 1989.

[15] F. Bullo. Series expansions for the evolution of mechanical control systems. *SIAM J. Control and Optimization*, 40(1):166–190, 2001.

[16] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *Int. J. Robot. Res.*, 18(6):534–555, 1999.

[17] L. G. Bushnell, D. M. Tilbury, and S. S. Sastry. Steering three-input nonholonomic systems: the fire truck example. *Int. J. Robot. Res.*, 14(4):366–381, 1995.

[18] J. Canny, A. Rege, and J. Reif. An exact algorithm for kinodynamic planning in the plane. *Discrete and Computational Geometry*, 6:461–484, 1991.

[19] M. Cherif. Kinodynamic motion planning for all-terrain wheeled vehicles. In *IEEE Int. Conf. Robot. & Autom.*, 1999.

[20] 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.

[21] C. I. Connolly, J. B. Burns, and R. Weiss. Path planning using laplace's equation. In *IEEE Int. Conf. Robot. & Autom.*, pages 2102–2106, May 1990.

[22] H. S. M. Coxeter. *Regular Polytopes*. Dover Publications, New York, NY, 1973.

[23] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. *Algorithmica*, 14(6):480–530, 1995.

[24] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning: Robots with decoupled dynamics bounds. *Algorithmica*, 14(6):443–479, 1995.

[25] B. R. Donald, P. G. Xavier, J. Canny, and J. Reif. Kinodynamic planning. *Journal of the ACM*, 40:1048–66, November 1993.

[26] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.

[27] M. A. Erdmann. On motion planning with uncertainty. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1984.

[28] P. Ferbach. A method of progressive constraints for nonholonomic motion planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 2949–2955, 1996.

[29] M. Fliess, J. Levine, P. Martin, and P. Rouchon. Flatness and defect of nonlinear systems. *International Journal of Control*, 61(6):1327–1361, 1993.

[30] S. Fortune and G. Wilfong. Planning constrained motion. In *STOCS*, pages 445–459, 1988.

[31] T. Fraichard. Dynamic trajectory planning with dynamic constraints: A 'state-time space' approach. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1393–1400, 1993.

[32] E. Frazzoli, M. A. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicles motion planning. Technical Report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1999.

[33] G. Heinzinger, P. Jacobs, J. Canny, and B. Paden. Time-optimal trajectories for a robotic manipulator: A provably good approximation algorithm. In *IEEE Int. Conf. Robot. & Autom.*, pages 150–155, Cincinnati, OH, 1990.

[34] J. Hershberger and S. Suri. Efficient computation of Euclidean shortest paths in the plane. In *Proc. 34th Annual IEEE Sympos. Found. Comput. Sci.*, pages 508–517, 1995.

[35] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. Comput. Geom. & Appl.*, 4:495–512, 1999.

[36] A. Isidori. *Nonlinear Control Systems.* Springer-Verlag, Berlin, 1989.

[37] P. Jacobs and J. Canny. Planning smooth paths for mobile robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 2–7, 1989.

[38] P. Jacobs, J. P. Laumond, and M. Taix. Efficient motion planners for nonholonomic mobile robots. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1229–1235, 1991.

[39] L. E. Kavraki. *Random Networks in Configuration Space for Fast Path Planning.* PhD thesis, Stanford University, 1994.

[40] L. E. Kavraki. Computation of configuration-space obstacles using the Fast Fourier Transform. *IEEE Trans. Robot. & Autom.*, 11(3):408–413, 1995.

[41] 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.

[42] O. Khatib. *Commande dynamique dans l'espace opérational des robots manipulateurs en présence d'obstacles.* PhD thesis, Ecole Nationale de la Statistique et de l'Administration Economique, France, 1980.

[43] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Robot. Res.*, 5(1):90–98, 1986.

[44] R. Kimmel, N. Kiryati, and A. M. Bruckstein. Multivalued distance maps for motion planning on surfaces with moving obstacles. *IEEE Trans. Robot. & Autom.*, 14(3):427–435, June 1998.

[45] R. Kindel, D. Hsu, J.-C. Latombe, and S. Rock. Kinodynamic motion planning amidst moving obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 2000.

[46] P. Konkimalla. Efficient computation of optimal navigation functions for nonholonomic planning. Master's thesis, Iowa State University, Ames, IA, 1999.

[47] P. Konkimalla and S. M. LaValle. Efficient computation of optimal navigation functions for nonholonomic planning. In *Proc. First IEEE Int'l Workshop on Robot Motion and Control*, pages 187–192, 1999.

[48] B. H. Krogh. A generalized potential field approach to obstacle avoidance control. In *Proceedings of SME Conference on Robotics Research*, August 1984.

[49] 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.

[50] G. Laffierriere and H. J. Sussman. Motion planning for controllable systems without drift. In *IEEE Int. Conf. Robot. & Autom.*, 1991.

[51] R. E. Larson. A survey of dynamic programming computational procedures. *IEEE Trans. Autom. Control*, 12(6):767–774, December 1967.

[52] R. E. Larson and J. L. Casti. *Principles of Dynamic Programming, Part II.* Dekker, New York, NY, 1982.

[53] J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, MA, 1991.

[54] J.-C. Latombe, A. Lazanas, and S. Shekhar. Robot motion planning with uncertainty in control and sensing. *Artif. Intell.*, 52:1–47, 1991.

[55] J. P. Laumond. Trajectories for mobile robots with kinematic and environment constraints. In *Proc. of International Conference on Intelligent Autonomous Systems*, pages 346–354, 1986.

[56] 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.

[57] J. P. Laumond and T. Siméon. Motion planning for a two degrees of freedom mobile robot with towing. Technical Report 89-148, Laboratoire d'Analyse et d'Architecture des Systemes / Centre National de la Recherche Scientifique, Toulouse, France, 1989.

[58] J. P. Laumond, T. Siméon, R. Chatila, and G. Giralt. Trajectory planning and motion control of mobile robots. In *Proceedings of IUTAM/IFAC Symposium*, pages 351–366, 1988.

[59] J. P. Laumond and P. Souéres. Metric induced by the shortest paths for a car-like robot. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1299–1303, 1993.

[60] S. M. LaValle. *A Game-Theoretic Framework for Robot Motion Planning.* PhD thesis, University of Illinois, Urbana, IL, July 1995.

[61] S. M. LaValle. Numerical computation of optimal navigation functions on a simplicial complex. In P. Agarwal, L. Kavraki, and M. Mason, editors, *Robotics: The Algorithmic Perspective.* A K Peters, Wellesley, MA, 1998.

[62] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.

[63] S. M. LaValle and S. A. Hutchinson. An objective-based framework for motion planning under sensing and control uncertainties. *International Journal of Robotics Research*, 17(1):19–42, January 1998.

[64] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 473–479, 1999.

[65] S. M. LaValle and R. Sharma. On motion planning in changing, partially-predictable environments. *International Journal of Robotics Research*, 16(6):775–805, December 1997.

[66] C. W. Lee. *Subdivisions and Triangulations of Polytopes.* CRC Press, 1997.

[67] Z. Li and J. F. Canny. Robot motion planning with nonholonomic constraints. Technical report, Electronics Research Laboratory, University of California, February 1989.

[68] Z. Li and J. F. Canny. *Nonholonomic Motion Planning.* Kluwer Academic Publishers, Boston, MA, 1993.

[69] Z. Li, J. F. Canny, and S. S. Sastry. On motion planning for dextrous manipulation, part i: The problem formulation. In *IEEE Int. Conf. Robot. & Autom.*, pages 775–780, 1989.

[70] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic systhesis of fine-motion strategies for robots. *Int. J. Robot. Res.*, 3(1):3–24, 1984.

[71] A. De Luca, G. Oriolo, and C. Samson. Feedback control of a nonholonomic car-like robot. In J.-P. Laumond, editor, *Robot Motion Plannning and Control*, pages 171–253. Springer-Verlag, Berlin, 1998.

[72] K. M. Lynch. Controllability of a planar body with unilateral thrusters. *IEEE Trans. on Automatic Control*, 44(6):1206–1211, 1999.

[73] K. M. Lynch and M. T. Mason. Stable pushing: Mechanics, controllability, and planning. *Int. J. Robot. Res.*, 15(6):533–556, 1996.

[74] E. Mazer, J. M. Ahuactzin, and P. Bessière. The Ariadne's clew algorithm. *J. Artificial Intell. Res.*, 9:295–316, November 1998.

[75] J. S. B. Mitchell. *Planning Shortest Paths.* PhD thesis, Stanford University, 1986.

[76] R. M. Murray, M. Rathinam, and W. M. Sluis. Differential flatness of mechanical control systems. In *Proc. ASME International Congress and Exposition*, 1995.

[77] R. M. Murray and S. Sastry. Nonholonomic motion planning: Steering using sinusoids. *Trans. Automatic Control*, 38(5):700–716, 1993.

[78] Y. Nakamura and R. Mukherjee. Nonholonomic path planning of space robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 1050–1055, 1989.

[79] W. S. Newman and N. Hogan. High speed robot control and obstacle avoidance using dynamic potential functions. In *IEEE Int. Conf. Robot. & Autom.*, pages 14–24, 1987.

[80] C. O'Dunlaing. Motion planning with inertial constraints. *Algorithmica*, 2(4):431–475, 1987.

[81] J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific J. Math.*, 145(2):367–393, 1990.

[82] J. Reif and H. Wang. Non-uniform discretization approximations for kinodynamic motion planning. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 97–112. A K Peters, Wellesley, MA, 1997.

[83] E. Rimon and D. E. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Trans. Robot. & Autom.*, 8(5):501–518, October 1992.

[84] J. J. Rotman. *Introduction to Algebraic Topology.* Springer-Verlag, Berlin, 1988.

[85] G. Sahar and J. M. Hollerbach. Planning minimum-time trajectories for robot arms. *Int. J. Robot. Res.*, 5(3):97–140, 1986.

[86] J. A. Sethian. *Level set methods : Evolving interfaces in geometry, fluid mechanics, computer vision, and materials science.* Cambridge University Press, 1996.

[87] A. M. Shkel and V. J. Lumelsky. Incorporating body dynamics into sensor-based motion planning: The maximum turn strategy. *IEEE Trans. Robot. & Autom.*, 13(6):873–880, December 1997.

[88] P. Souéres and J. P. Laumond. Shortest paths synthesis for a car-like robot. In *IEEE Transactions on Automatic Control*, pages 672–688, 1996.

[89] H. K. Struemper. *Motion Control for Nonholonomic Systems on Matrix Lie Groups.* PhD thesis, University of Maryland, College Park, MD, 1997.

[90] 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.

[91] H. Sussman and G. Tang. Shortest paths for the Reeds-Shepp car: A worked out example of the use of geometric techniques in nonlinear optimal control. Technical Report SYNCON 91-10, Dept. of Mathematics, Rutgers University, 1991.

[92] M. Taix. *Planification de Mouvement pour Robot Mobile Non-Holonome.* PhD thesis, Laboratoire d'Analyse et d'Architecture des Systemes, Toulouse, France, January 1991.

[93] R. B. Tilove. Local obstacle avoidance for mobile robots based on the method of artificial potentials. In *IEEE Int. Conf. Robot. & Autom.*, pages 566–571, May 1990.

[94] L. Yang and S. M. LaValle. A framework for planning feedback motion strategies based on a random neighborhood graph. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 544–549, 2000.

[95] Y. Yu and K. Gupta. On sensor-based roadmap: A framework for motion planning for a manipulator arm in unknown environments. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1919–1924, 1998.