Efficient Computation of Optimal Navigation Functions for Nonholonomic Planning

Prashanth Konkimalla   Steven M. LaValle

Department of Computer Science
Iowa State University
Ames, IA 50011 USA
1-515-294-2259   1-515-294-0258 (FAX)
{prasanth,lavalle}@cs.iastate.edu

Abstract

We present a fast, numerical approach to computing optimal feedback motion strategies for a nonholonomic robot in a cluttered environment. Although many techniques exist to compute navigation functions that can incorporate feedback, none of these methods is directly able to determine optimal strategies for general nonholonomic systems. Our approach builds on previous techniques in numerical optimal control, and on our previous efforts in developing algorithms that compute feedback strategies for problems that involve nondeterministic and stochastic uncertainties in prediction. The proposed approach efficiently computes an optimal navigation function for nonholonomic systems by exploiting two ideas: 1) the principle of Dijkstra's algorithm can be generalized to continuous configuration spaces and nonholonomic systems, and 2) a simplicial mesh representation can be used to reduce the complexity of numerical interpolation.

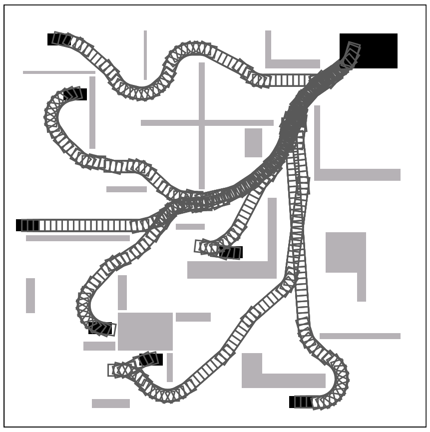


Figure 1: As an example, a navigation function can be used to generate optimal motions for a car-like robot from any initial configuration to a fixed goal region.

1 Introduction

Many applications in robotics require computing a motion strategy that brings a complex geometric body from an initial configuration to a goal region while satisfying complicated, geometric constraints that model the environment. Many of these systems involve nonholonomic constraints due to contacts between the robot and its environment. For example, these constraints can arise from wheeled mobile robots [3, 11, 25], or manipulation of an object without form or force closure [1, 19]. We are particularly interested in computing 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 localization errors, avoidance maneuvers, nonprehensile manipulation, modeling errors, etc.
3. An optimal strategy is requested that brings the robot from an initial configuration to a goal region.
4. Nonholonomic constraints exist for which there are no analytical solutions to the shortest path problem (e.g., there exist solutions in the case of Reeds and Shepp curves [21]).

The first condition is reasonable for many problems, and forms the basis for the vast majority of motion planning research (see, for example, [10]). 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., [6, 22, 24]). The continuous Dijkstra paradigm [4, 20] 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 holonomic planning problems [7]. 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 [12]. Most of these methods do not produce optimal paths (an exception is the case of a car-like robot with reverse), and in general, these methods do not attempt to construct a navigation function or feedback strategy. The method presented in [2] is perhaps most related to ours because it is able to find optimal solutions for very general nonholonomic problems; however, the key distinction is that the method in [2] does not construct an optimal navigation function.

Our goal is to present general, numerical approach to computing an optimal navigation function for a nonholonomic planning problem. Figure 1 shows a computed example; for a given goal, a navigation function can be used to generate a path from any initial configuration. We focus our efforts on kinematic solutions, and make the traditional assumption that a separate controller will be designed to satisfy system dynamics. We also allow the optimal motion strategy to have nonsmooth inputs. Under the restriction to smooth inputs, Brockett's condition would imply that time-varying feedback control is needed. A recent overview of dynamics, control, and tracking issues for car-like robots is given in [18]

Emphasis is placed on developing a technique that can be applied to a broad set of problems, as opposed to requiring major adjustments to be made for a particular problem. In previous efforts we have demonstrated the use of numerical dynamic programming computations with interpolation [15, 16] for computing optimal feedback strategies for problems that involve nondeterministic and stochastic uncertainties in prediction. The current approach can be considered as a significant advancement over the techniques in [15, 16] that is much faster in practice (typically by a couple of orders of magnitude). The current approach applies both to the problems defined in [15, 16] and to the nonholonomic planning problem, which the primary focus of this paper.

2 Problem Formulation

Our problem is defined in a bounded 2D or 3D world, $\mathcal{W} \subset \Re^N$, such that $N = 2$ or $N = 3$. Let $\mathcal{O} \subset \mathcal{W}$ denote a static *obstacle region* in the world. An n-dimensional *configuration space*, $\mathcal{C}$, captures position, orientation, and/or other information for robot. Let $x \in \mathcal{C}$ denote a configuration. Let $\mathcal{C}_{free}$ denote the collision-free subset of the configuration space. Let u be an m-dimensional vector of control inputs. A control input could, for example, specify a particular steering angle for a car-like robot. Let $U \subseteq \Re^m$ represent the space of possible control inputs. The task will be to determine control inputs that depend on a current configuration, will "drive" the robot into the goal region, and will optimize a specified criterion.

The nonholonomic kinematic constraints can be represented using control-theoretic notation as $\dot{x} = f(x(t), u(t))$. Let $G \subset \mathcal{C}_{free}$ denote a fixed goal region. In general, G might not be connected. A *loss functional* can be defined that evaluates any configuration trajectory and control function:

$$L = \int_0^{T_f} l(x(t), u(t))dt + Q(x(T_f)). \tag{1}$$

The term $u(t)$ yields the control input applied at time t. With a given feedback motion strategy, γ, this would be chosen as $u(t) = \gamma(x(t))$. The integrand $l(x(t), u(t))$ represents an instantaneous cost, which when integrated, can be imagined as the total amount of energy expended. The term $Q(x(T_f))$ is a final cost that can be used to penalize trajectories that fail to terminate in G.

In most motion planning research the solution takes the form of a path; however, in our case the solution involves configuration feedback. Therefore, we construct a real-valued function on the configuration space that is used for navigation in the sense proposed in [22]. Let the *cost-to-go* function, $L^* : \mathcal{C}_{free} \rightarrow \Re \cup \{\infty\}$ represent the loss according to (1), that would be received by choosing the optimal control input and driving the system until time $t = T_f$. The cost-to-go is also referred to as a value function in [24].

We next define a discrete-time approximation in which actions are performed at each Δt. The expression $\dot{x} = f(x(t), u(t))$ can be approximated in discrete time as a difference equation, $x_{k+1} = f(x_k, u_k)$. For stationary systems the cost-to-go function can be expressed as [9]:

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

in which $l_k(x_k, u_k)$ is the loss (1) that accumulates over time Δt. Our computational approach can be considered as a fast numerical technique that approximately solves this difference equation.

The right side of (2) is used during execution to select the optimal action, u_k, that should be taken at a configuration x_k. Even though L^* will be represented at discrete points, the values of $L^*(x)$ for any $x \in \mathcal{C}_{free}$ can be obtained through an interpolation scheme (which will be discussed in Section 3). This enables the control to be determined from any state (i.e., the system is not required to follow a grid or any other regular pattern; it naturally obeys the nonholonomic constraints).

3 An Efficient Representation of L^*

The main task in generating these optimal motion strategies is to compute the *navigation function*. For this, we will choose a set of points distributed uniformly over the $\mathcal{C}$-space and compute the cost-to-go function at each point, which will be referred to as a *control point*. We will explain the algorithm for computing the cost-to-go function in Section 4. The cost-to-go is only defined on the control points; however, we need to consider smooth trajectories that do not necessarily visit the control points. So, for any point which is not a control point, we compute the cost-to-go at that point by interpolating it with the cost-to-go at the neighboring control points, thus making the cost-to-go function continuous. Now we need to figure out how many neighboring control points need to be considered, which are they, and how to interpolate the cost-to-go at these neighbors.

3.1 A Simple Interpolation Scheme

Linear interpolation schemes have been used extensively for numerical optimal control computations (e.g., [8, 9]). We can define an interpolation scheme as follows. Let us consider a 3D $\mathcal{C}$-space, although the ideas apply to higher dimensions. Let $\mathcal{C}$ be divided into cubes with control points as vertices. In this representation, any point, x, in the space will have eight neighbors which are the vertices of the cube that x is contained in. The cost-to-go at each neighbor contributes to the cost-to-go at the point. Now, we will compute the weights for each neighbor. A neighboring vertex has more weight if it is closer to x. If $\{p_1, \ldots, p_8\}$ are the neighbors, and $\{\beta_1, \ldots, \beta_8\}$ are positive weights based on linear interpolation, then the cost-to-go at x is given by

$$L^*(x) \approx \sum_{i=1}^{8} \beta_i L^*(p_i). \tag{3}$$

So, given any point in space, we need to follow three steps to compute the cost-to-go at that point. 1) Find the cube that contains the point. This step will be referred to as a *point location problem*. 2) Compute the interpolation coefficient associated with each neighbor (vertices of the cube) of the point and 3) Find the cost-to-go for that point using the above equation. If the above representation is used, steps 1 and 2 can be carried out very efficiently, but the time complexity of step 3 is exponential in dimension. This is because the number of neighbors associated with any point is 2^n(where n is the dimension). So, an alternative representation which can carry out all the three steps more efficiently would be preferred. The main problem in using the above representation is the large number of neighbors associated with any point. If we can divide the cube further into tetrahedra, then step 3 becomes simpler. This is because the number of neighbors of any point would reduce to 4 (or $n+1$ in general) from 8 (or 2^n), thus reducing the time complexity of step 3 from exponential to linear in dimension. At the same time, we need to make sure that this decomposition into tetrahedra does not complicate steps 1 and 2. Also, the representation has to be general enough to work in any dimension. Section 3.2 presents such a method.

3.2 Complete Barycentric Subdivision

To compute the cost-to-go function at any point efficiently, we need to represent the $\mathcal{C}$-space in such a way that steps 1,2, and 3 in Section 3.1 can be carried out efficiently. The main reason for dividing a cube further into tetrahedra is to represent the space as union of polytopes with $n+1$ vertices (which are called *simplexes*) instead of polytopes with 2^n vertices. Dividing the space into cubes can make the point location problem easy, but computing the cost-to-go (step 3) becomes

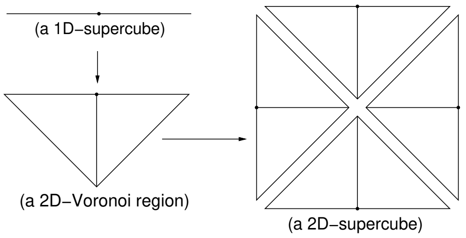


Figure 2: An extension from 1D to 2D.

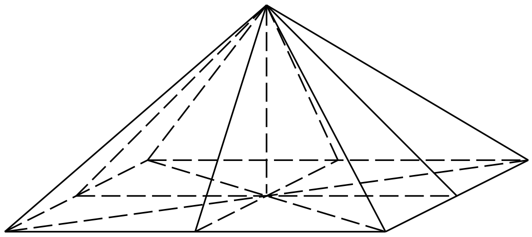


Figure 3: An extension from 2D to 3D.

complicated. To overcome this problem, we further divide the cube into tetrahedra (in general, we divide hypercubes into simplexes).

It has been shown in [17] that an nD-cube must be divided into at least $2^n(n+1)^{-(n+1)/2}n!$ simplexes. Thus, a 2D-cube (a square) can be divided into no less than two triangles (2D-simplex). The minimum number of tetrahedra that a 3D-cube can be divided into is found to be 5. But, these kind of triangulations are not feasible for our problem as it makes the point location problem difficult. So, a special way of representation is chosen that is most suitable for our problem. In our representation, we will first divide the space into cubes, and then divide each set of eight cubes(which will be referred as $3D$-*supercube*) into 48 tetrahedra as shown in figure 4. Effectively, each cube is divided into 6 tetrahedra. In general, an n dimensional $\mathcal{C}$-space is divided in to nD-cubes, and each set of 2^n nD-cubes(nD-*supercube*) is further divided into $2^n n!$ simplexes. Figures 2 and 3 illustrate that this representation is general and is ap-

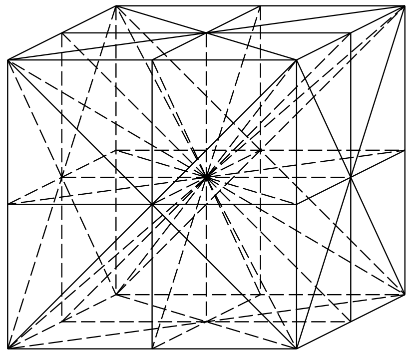


Figure 4: A $3D$-*supercube*(set of 8 cubes) divided into 48 tetrahedra.

plicable to higher dimensional $\mathcal{C}$-spaces. It is important to note that we do *not* explicitly store simplexes; only the control points are stored.

In each simplex, the interpolation weights are selected as the *barycentric coordinates* [23]. This is the set of positive coefficients $\beta_1, \beta_2, \ldots, \beta_{n+1}$ such that $\sum_i^{n+1} \beta_i = 1$ and each point, x, in the simplex can be represented as a linear combination $x = \sum_i^{n+1} \beta_i x_i$ in which the x_i are the vertices of the simplex.

Now, our problem is, given any point, find the simplex that contains the point and find its barycentric coordinates in that simplex. For this, we find the supercube that contains the point. Then, the coordinates of the point in the supercube, with the center of the supercube as origin, is computed. This requires a linear translation of the coordinate frame which takes linear time. At first sight, it may seem that, the point location problem takes exponential time because the number of simplexes the point can possibly be in is $2^n n!$ (number of simplexes in the supercube). We will show that, because of a special representation used, the point location problem can be solved in just $\mathrm{O}(n \lg n)$ time instead of exponential time.

For simplicity, let supercube refer to a 3D-supercube. First, we make the following observations about our representation: 1) The center of the supercube is common to all of the tetrahedra in the supercube. Hence, this is a vertex of the tetrahedron that contains the point. 2) Each supercube has six faces. Consider the 3D-Voronoi region associated with each face. We will refer an n dimensional Voronoi region as *nD-Voronoi region*. In general, an nD-supercube has $2n$ faces and hence $2n$ nD-Voronoi regions. The 3D-Voronoi region that contains the point can be found by determining the coordinate with highest magnitude. In other words, the coordinate that is closest to a face is found. Once this is done, the same observations as above are made, but at dimension $(n-1)$. For the 3D case: 1) The center of the corresponding face (which is a 2D-supercube) in the 3D-Voronoi region is common to all the tetrahedra in that 3D-Voronoi region and, in other words, common to all the triangles on the face. Hence this too is a vertex of the tetrahedron that contains the point. 2) The 2D-supercube (square) has 4 2D-Voronoi regions. See Figures 2, 3 and 4. So, the problem reduces to finding the 2D-Voronoi region that contains the point by finding the coordinate with next highest magnitude. Once this is done, the same thing as above is done at one more dimension less (at 1D). The work involved in finding the tetrahedron that contains the point, is the work involved in arranging the coordinates of the point in decreasing order of their magnitudes, which takes $\mathrm{O}(n \lg n)$ time. Hence the point location problem takes only $\mathrm{O}(n \lg n)$ time.

The next task is to find the barycentric coordinates of the point in its tetrahedron. Suppose without loss of generality that the corresponding supercube has been scaled to contain unit cubes, centered at $(0,0,0)$, and the point x lies in the first octant (all coordinates are between 0 and 1). If p_1, p_2, p_3, and p_4 are the vertices of the tetrahedron found at each step, let β_1, β_2, β_3 and β_4 are the respective barycentric coordinates, and x_1, x_2, and x_3 are the magnitudes of coordinates of the point (in decreasing order), then $\beta_1 = 1 - x_1$, $\beta_2 = x_1 - x_2$, $\beta_3 = x_2 - x_3$, $\beta_4 = x_3 - 0$. In general, in an n dimensional $\mathcal{C}$-space, for $1 \leq \mathrm{i} \leq \mathrm{n}$, if p_i is the i^{th} vertex of the simplex, and β_i is the barycentric coordinate of p_i, x_i is the i^{th} coordinate of the point x, then $\beta_i = x_{i-1} - x_i$, where $x_0 = 1$, $x_{n+1} = 0$. It can be seen from above that, computation of barycentric coordinates takes $\mathrm{O}(n)$ time, since the arrangement of coordinates of the point in decreasing order of magnitude has already been done in the previous step.

Finally, the cost-to-go at x, is found by using (3). Because there are only $n+1$ interpolation neighbors, this step now takes only $\mathrm{O}(n)$ time. Hence, the computation of cost-to-go at any point in space takes $\mathrm{O}(n \lg n)$ time, as opposed to $O(2^n)$ time using the simple approach. The improvement is also significant in practice (i.e., the scaling constant in the analysis is small).

4 The Algorithm

This algorithm is an adaptation of the method in [14], and it performs a kind of wavefront propagation by iteratively constructing nonholonomic preimages. The n-dimensional $\mathcal{C}$-space is divided into simplexes as described above. Let p be a control point, and let P be the set of all control points. Let *C-set* be the set of all control points whose optimal cost-to-go value has already been correctly computed. If all of the vertices of a simplex belong to C-set, then the simplex is considered a *computed simplex* (it can be used for interpolation), and the union of all the computed simplexes is called the *computed region*. The algorithm iteratively grows the computed region outward from the goal. Let the *preimage*, *P-set*, be a set of all control points in $\mathcal{C}_{free} \setminus G$, from which the robot can get into the computed region in a single time step, Δt.

Our objective is to compute the cost-to-go at each p. The first step in the algorithm is to initialize P. Let $P_{goal} = \{p \mid p \in G\}$. For each $p \in P_{goal}$, assign $L^*(p) = 0$, and for each $p \notin P_{goal}$, assign $L^*(p) = \infty$. Then, at Step 2, we initialize C-set to P_{goal}. Steps 4 to 10 are contained in the main iteration of the algorithm. Each iteration in this **repeat** loop forms a stage. Initially, at stage 1, C-set $\leftarrow P_{goal}$. In Step 4, P-set is determined. Let *T-set* be a temporary set of control points. In Step 5, T-set is set to $\emptyset$. In Steps 6 to 9, we remove a control point, p, from P-set until it is empty. Each time we compute the cost-to-go at that control point using (2) and add it to T-set. The point p is initially added to T-set instead of C-set in each iteration because it may effect the computation of cost-to-go at the control points removed in subsequent iterations. In other words, if p was instead added to C-set instead, the computed region may expand in each iteration, thus effecting the computation of cost-to-go at rest of the control points in P-set. To avoid this, we add p to T-set instead. Thus, the computed region is kept constant between Steps 6 and 9. Hence, at each stage, the computed region expands only once. This happens at Step 10, when all of the control points in T-set are added to C-set.

The most challenging and time-consuming part of the algorithm is the computation of $L^*(p)$ at Step 8. As explained in Section 3, the cost-to-go at any point in the

```
OPTIMAL_NAVIGATION_FUNCTION()
Initialize P
C-set ← P_goal
repeat
    P-set ← preimage(C-set)
    T-set ← ∅
    while (P-set != ∅) do
        p ← get_element(P-set)
        Compute L*(p)
        T-set ← T-set ∪ {p}
    C-set ← C-set ∪ T-set
until (T-set ! = ∅)
Return L*
```

Figure 5: This algorithm computes the optimal feedback motion strategy in a single pass over the configuration space.

finalized region is found by first locating the simplex that contains the point, finding its barycentric coordinates in that simplex, and then interpolating them with the cost-to-go at the respective vertices.

5 Results

This section presents some results from an implementation of the algorithm in Figure 5 using GNU C++ and LEDA under Linux on a Pentium Pro PC. We have implemented the algorithm and computed navigation functions for two different nonholonomic problems: the planning for a car-like robot, and manipulation planning by pushing. The configuration space for each problem has three degrees of freedom.

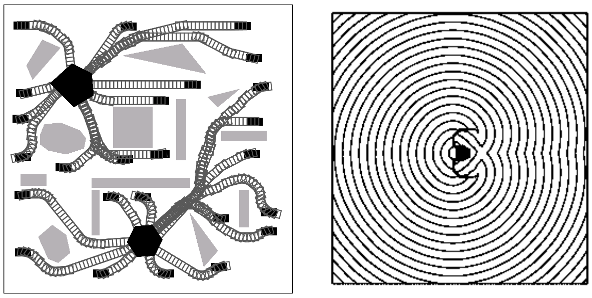
5.1 Car-like Robot

In this problem, a car-like robot with nonholonomic constraints and a minimum turning radius is driven to its goal, optimizing the distance traveled. Alternatively, the number of reversals can be minimized by adding extra cost to backward motions. Figures 1 and 7.a show the trajectories of the robot from several chosen configurations for two different problems in which the car is allowed to move both forward and backward. Figure 6 shows the computation times for the problem in Figure 1. The precomputation time is shown separately, which is a simple bitmap computation of the configuration space (it can be accomplished by much fast techniques than what is shown here; see, for example, [5]). Figure 7.a shows a problem in which G has multiple connected components. The navigation function naturally steers the robot to the appropriate component of G.

Figure 7.b shows the level set contours of L^* for a point car-like robot that can only go forward.

Res	Precmp	DP
$\lvert I\rvert \times \lvert J\rvert \times \lvert K\rvert$	(sec)	(sec)
$60 \times 60 \times 30$	8.25	27.22
$100 \times 100 \times 30$	22.62	34.55

Figure 6: Computational performance from a GNU C++ implementation on a Pentium Pro PC running Linux. The columns denote: P = problem number, Res = resolutions, Precmp = time to compute C-space bitmap, DP = computation time for generating the optimal navigation function.



a. b.

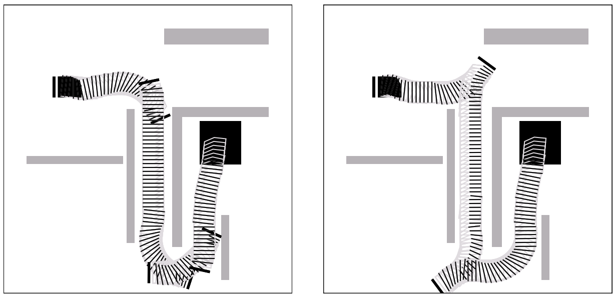
Figure 7: a) A car-like robot problem with multiple goals. Given any initial configuration, the car goes to the nearest goal, thus demonstrating the optimality of the solution. b) The level sets of the cost-to-go function for a particular orientation of a car that can go only forward.

5.2 Push-Planning

In the case of the push planning problem [1, 19], the robot's task is to push a box to the goal while avoiding obstacles. The box can be pushed from along one of two edges (which are highlighted in the figure). It is assumed that the robot makes a line contact with the robot. The dimensions of the robot are not considered and it is assumed that the robot can switch from one edge to other with out colliding with any of the obstacles. We can optimize paths either by the distance traveled, the number of reversals, or some combination. Figure 8.a shows the trajectory of the box if no cost is added for switching of edges. In this case, it takes 121 time steps, switching edges 6 times to reach the goal. Figure 8.b shows the trajectory of the box if a finite cost is added for switching of edges. In this case, it takes 125 time steps, switching edges only 2 times to reach the goal.

6 Conclusions

An approach has been presented for computing optimal feedback motion strategies for nonholonomic planning problems. Instead of precomputing a path, the



a. b.

Figure 8: Trajectory of the box: a) when no cost is added for switching edges; b) when a finite cost is added for switching edges.

solution takes the form of a navigation function, as considered in [22]. Two improvements are made over our previous approach to the problem of numerically computing optimal cost-to-go functions [13]: 1) The solution can be obtained in a single iteration over the configuration space; 2) the complexity of the interpolation has been reduced from $O(2^n)$ to $O(n \lg n)$. The technique developed is general in nature and can be extended to higher degree-of-freedom problems. It is important to note that once a navigation function is computed, it can be quickly utilized during execution. We hope to apply this algorithm to more difficult nonholonomic, and even kinodynamic planning problems, of up to six degrees of freedom. The computational savings from using our new interpolation scheme make extensions to the harder problems more feasible.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

[11] J.-P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray. A motion planner for nonholonomic mobile robots. *IEEE Trans. Robot. & Autom.*, 10(5):577–593, October 1994.

[12] 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.

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

[14] 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. To appear.

[15] 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.

[16] 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.

[17] Carl W. Lee. *Subdivisions and Triangulations of Polytopes.* CRC Press, 1997.

[18] 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.

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

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

[21] 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.

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

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

[24] 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.

[25] P. Svestka and M. H. Overmars. Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. In *IEEE Int. Conf. Robot. & Autom.*, pages 1631–1636, 1995.