Optimal, Smooth, Nonholonomic Mobile Robot Motion Planning in State Lattices

Mihail Pivtoraiko Ross Knepper Alonzo Kelly

CMU-RI-TR-07-15

May 2007

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

© Carnegie Mellon University

Abstract

We present an approach to the problem of mobile robot motion planning in arbitrary cost fields subject to differential constraints. Given a model of vehicle maneuverability, a trajectory generator solves the two point boundary value problem of connecting two points in state space with a feasible motion. We use this capacity to compute a control set which connects any state to its reachable neighbors in a limited neighborhood. Equivalence classes of paths are used to implement a path sampling policy which preserves expressiveness while eliminating redundancy. The implicit repetition of the resulting minimal control set throughout state space produces a reachability graph that encodes all feasible motions consistent with this sampling policy. The graph encodes only feasible motions by construction and, by appropriate choice of state space dimension, can permit full configuration space collision detection while imposing heading and curvature continuity constraints at nodes. Nonholonomic constraints are satisfied by construction in the trajectory generator. We also use the trajectory generator to compute an ideal admissible heuristic and significantly improve planning efficiency. Comparisons to classical grid search and nonholonomic motion planners show the planner provides better plans or provides them faster or both. Applications to planetary rovers and terrestrial unmanned ground vehicles are illustrated.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Related Work 2
1.3 Problem Statement 4
1.4 Approach 4
1.5 Outline 5

2 Search Space Design **5**
2.1 Conversion to a Sequential Decision Process 5
2.2 Configuration and State Spaces 6
2.3 Sampled Representations and Spatial Equivalence Classes 6
2.4 Regular Lattices and Grids 7
2.5 Feasible Lattices 8

3 The State Lattice **8**
3.1 Modeling Vehicle Maneuverability 9
3.2 Desiderata and Design Approach 10
3.3 Sampling the Heading Dimension 10

4 Primitive Control Sets **11**
4.1 Extended Neighborhoods 11
4.2 Redundant Controls and Path Decomposition 12
4.3 Generating Primitive Control Sets with Path Sampling 12
4.4 Automatic Generation of the Control Set 14

5 Motion Planning using Control Sets **16**
5.1 Search Algorithm 16
5.2 Qualities of Motion Planning in State Lattices 17
5.3 Implementation Considerations 18
5.4 Heuristic Cost Estimate 19
5.5 Heuristic Look-Up Table 20
5.5.1 Constructing the Heuristic Look-Up Table 20
5.5.2 Populating the Heuristic Look-Up Table 21

6 Results **24**
6.1 Experimental Setup 25
6.2 Comparison of the Lattice to Other Control Sets 26
6.2.1 Comparison to a Grid Planner 27
6.2.2 Comparison to Full C Space and Nonholonomic Planners 30
6.3 Heuristics 32

7 Applications **34**
7.1 Off-road Navigation 34
7.2 Planetary Rover Locomotion 35
7.3 Reactive Motion Planning and Path Modification 38

8 Summary **38**

9 Conclusion and Future Work **39**

1 Introduction

Discrete representation of states is a well-established method of reducing the computational complexity of planning at the expense of reducing completeness. However, in motion planning, such discrete representations complicate the satisfaction of differential constraints which reflect the limited maneuverability of many real vehicles. We propose a mechanism to achieve the computational advantages of discretization while satisfying motion constraints.

To this end we introduce a search space, referred to as the state lattice. The lattice is used to formulate a nonholonomic motion planning query as graph search. The state lattice encodes a graph whose nodes are a discretized set of all reachable configurations of the system and whose edges are feasible motions which connect these states exactly. The process of forming edges does not make the standard assumption that adjacent nodes are necessarily connected. Rather, every node is connected by an edge to only those nodes in a small neighborhood for which a feasible connecting motion exists.

Conversely, it is typical to assume that adjacent cells in a 2D grid are connected in a 4-connected or 8-connected arrangement. Such default connectivity fails to capture nonholonomic and continuity constraints. For example, a 90 degree turn at finite speed assumes either that wheels can move sideways or steering angles can change infinitely fast or both. In such search spaces, differential constraints must be considered heuristically in the optimization process during planning, or as an afterthought in plan post-processing. The state lattice, however, has an important property that each connection represents a feasible primitive motion. Such a graph topology - one that intrinsically meets mobility constraints - leads to superior motion planning results because no time is wasted generating, evaluating, or fixing infeasible plans.

It is convenient to sample state space in a regular fashion so that the motions encoded in the edges of the state lattice form a repeating unit that can be copied to every node while preserving the property that each edge will continue to join neighboring nodes exactly. This canonical set of repeating edges is called the control set. The number of edges in the control set is exactly the outdegree of each node in the reachability graph, so it is important to minimize it for efficiency. We will propose path sampling methods which automatically produce a minimal set of primitive paths that, when concatenated, can reproduce any feasible path in the continuum of state space up to some resolution.

1.1 Motivation

This work is motivated by a number of applications which have not been served as well by the planners of the day as they would have been by the planner proposed here. In the context of unmanned ground vehicles, competence in motion planning in dense obstacle fields demands the use of sufficiently high fidelity models of constrained motion, at least in the near field. Our repeated attempts to get by with less have not been particularly successful. A classic case is that of an Ackerman vehicle which drives into a winding corridor only to find it is closed off. Then, the robot must turn around using many reversals of velocity while avoiding the obstacles surrounding it.

Another case of contemporary interest is that of a planetary rover which must visit

a large number of nearby locations when, for every motion, only one of the rocks in view is (roughly) the goal and everything else is an obstacle. Furthermore, the goal is the correct vehicle state for deploying a microscopic imager to a 1 cm by 1cm patch of a particular boulder. Hence the planner must not only target a particular heading, but its ability to do so also depends on eliminating the errors that will be induced by discontinuous curvatures embedded in the path to get there.

A great deal of the motion planning literature still concentrates on the theoretical aspects of the problem. Our approach in this paper departs somewhat from this legacy because we are charged to do our best in the field robotics applications of today. The approach presented here has been conceived through work with off-road mobile vehicles, and its development has been entirely driven by the humbling business of field work. The question of efficiency has received considerable attention. We therefore contribute to the growing trend of describing the efficiency of our results in great detail and we offer comparisons of our method to other popular approaches.

1.2 Related Work

A significant amount of work has been dedicated in recent years to the problem of smooth inverse trajectory generation for nonholonomic vehicles: finding a smooth and feasible path given two end-point configurations. While in general this is a difficult problem, recent progress in this area produced a variety of fast algorithms. The groundbreaking work in analyzing the paths for nonholonomic vehicles was done by Dubins [19] and Reeds and Shepp [49]. Their ideas were further developed in algorithms proposed by Scheuer and Laugier [54], Fraichard and Ahuactzin [21], and Fraichard and Scheuer [22] where smoothness of paths was achieved by introducing segments of clothoids (curves whose curvature is a linear function of their length) along with arcs and straight line segments. Somewhat different approaches by Scheuer and Fraichard [53], and Lamiraux and Laumond [36], among others, have also been shown to solve the generation problem successfully and quite efficiently. On the other hand, Frazzoli, Dahleh and Feron [23] suggest that there are many cases where efficient, obstacle-free paths may be computed analytically, e.g. the systems with linear dynamics and a quadratic cost (double or triple integrators with control amplitude and rate limits). The cases that do not admit closed-form solutions can be approached numerically by solving appropriate optimal control problems (e.g. [20], [4]). A fast nonholonomic trajectory generator by Kelly and Nagy [32], Howard and Kelly [26] generates polynomial spiral trajectories using parametric optimal control. The latest results of this work appear in another article in this issue.

Some of the methods described above also proposed applications to planning among obstacles. Since the beginning of modern motion planning research ([45], [44], [50]), there has been interest in the planning methods that constructed boundary representations of configuration space obstacles ([15], [2], [1] and others). The complexity of motion planning algorithms has also been studied ([16], [47], [3], [28]). With the advent of efficient C-space sampling methods ([6], [25]), there has been interest in algorithms that sample the space in deterministic fashion ([7], [9], others in [37]). Lacaze et al. [35] utilized these ideas to propose a method for planning over rough terrain using generation of motion primitives by integrating the forward model. Cherif [18]

advanced these concepts by basing planning on physical modeling. Note that one of principal differences (and a novelty, to our knowledge) of our approach is leveraging the research in inverse trajectory generators to generate motion primitives under convenient discretization.

Also in the early 1990's, randomized sampling was introduced to motion planning ([8], among others). The Probabilistic Roadmap (PRM) methods were shown to be well-suited for path planning in C-spaces with many degrees of freedom ([29], [30], [27]), and with complex constraints, e.g. nonholonomic, kinodynamic, etc. ([34], [27], [17], [33]). Another type of probabilistic planning was Rapidly-exploring Random Trees (RRT) introduced by LaValle and Kuffner ([39],[40]). RRT's were originally developed for handling differential constraints, although they have also been widely applied to the Piano Mover's problem ([38]). Randomized approaches are understood to be incomplete, strictly speaking, but capable of solving many challenging problems quite efficiently ([14]).

As the randomized planners became increasingly well understood in recent years, it was suggested that their efficiency was not due to randomization itself. LaValle, Branicky and Lindemann [41] suggest an intuition that real random number generators always have a degree of determinism. In fact, Branicky et al. [14] show that quasi-random sampling sequences can accomplish similar or better performance than their random counterparts. The improvements in performance are primarily attributed to the more uniform sampling of quasi-random methods, and hence LaValle, Branicky and Lindemann [41] suggest that a carefully designed low-discrepancy incremental deterministic sequence would be able to do just as well ([42], [43]). For these reasons, Branicky et al. [14] introduced Quasi-PRM and Lattice Roadmap (LRM) algorithms that used low-discrepancy Halton/Hammersley sequences and a regular lattice, respectively, for sampling. Both methods were shown to be resolution-complete, while the LRM appeared especially attractive due to its properties of optimal dispersion and near-optimal discrepancy. In this light, our approach of sampling on a regular lattice can be considered to be one of building on the LRM idea and making it more efficient through a detailed analysis of a fit between the reachability graph of the system and the underlying sampling lattice.

Recent works have also discussed "lazy" variants of the above planning methods that avoid collision checking during the roadmap construction phase (e.g. [13], [12], [14], [51], [52]). In this manner the same roadmap could be used in a variety of settings, at the cost of performing collision checking during the search. An even "lazier" version is suggested, in which "the initial graph is not even explicitly represented" [14]. In this regard, our approach of using an implicit lattice and searching it by means of a pre-computed control set that only captures local connectivity is similar to the Lazy LRM. Our contribution is in exploring the conjecture made in that work and successfully applying it to nonholonomic motion planning.

The question of lattice sampling has also been raised in the context of motion planning utilizing control quantization. Bicchi, Marigo and Piccoli [11] as well as Pancanti, Pallottino and Bicchi [48] showed that, for systems that can be expressed as that are not underactuated, through careful discretization in control space, it is possible to force the resulting reachability graph of system to be a lattice. It was also shown that this technique can be applied to a large class of nonholonomic systems. That approach presents

a way to generate a path given its terminal points and shows how under suitable conditions a regular lattice of reachable points can be achieved. However, this is usually difficult to achieve, and under most quantizations the vertices of the reachability graph are unfortunately dense in the reachable set [38]. By contrast, our method uses an inverse path generator capable of generating essentially any feasible motion, so we can choose a convenient discretization of state space based on the problem demands and generate a custom set of controls to suit this discretization.

In this work we also build on a considerable amount of research in analysis of system reachability. One line of research uses planning methods themselves to analyze reachability of complex systems [10]. Since we focus on applications of motion planning of wheeled vehicles, we have developed a technique based on exhaustive search that has worked quite well for exploration of the reachability graph. Our method involves pruning this search early to analyze reachability efficiently.

In the development of Rapidly-exploring Dense Trees [38] for motion planning with differential constraints, the importance of designing off-line a family of motion primitives that captures the specifics of the system under consideration is noted [38]. In this light, our proposed control set is precisely that set of primitives that reflects symmetries of wheeled vehicles and encodes nonholonomic constraints via off-line reachability analysis. Thus, this work is aligned with the latest developments of differentially-constrained motion planning research, while continuing the latest trends of deterministic sampling methods and the efficiencies of ”lazy” exploration of state space.

Initial concepts of this work were validated in a successful field implementation of a nonholonomic motion planner built using an earlier version of the state lattice of limited size represented explicitly [31]. In this article, we propose significant improvements in efficiency and generality while recasting the approach to generate the reachability graph on-line as it is searched.

1.3 Problem Statement

The basic problem we address here is that of finding a feasible path between two given postures in the presence of arbitrary obstacles. If no path exists between the postures, the solution should indicate that, and otherwise it must find a solution. This rule is enforced as the sampling resolution approaches zero. The previous statement implies that the objective is a resolution-complete path planner. We also desire the solution to be optimal with respect to an arbitrary but well behaved notion of a path's cost (where cost could be assigned to path length).

1.4 Approach

The approach presented in this work builds upon a long heritage and many strengths of earlier motion planning techniques in order to achieve a solution of a highly general form and real-time performance.

Our aim is to leverage the previous effort, and combine these leading ideas in a novel and unique manner. In particular, our method builds upon the previous introduction of an implicit representation of the search space [14], latest research in inverse

trajectory generation ([54], [23], [26] among others), incremental deterministic sampling ([41]), and an off-line analysis of the reachability set of the system for greatly improved on-line performance.

In general terms, the crux of our proposal is the generation of a search space (a graph) that satisfies differential constraints by construction. The problem of motion planning is thereby reduced to unconstrained heuristic search and any generated plan must be feasible. We show that a planner can be built using this search space with node outdegree comparable to that of a grid. Hence, similar efficiencies are obtained in the generation of plans of equal numbers of states. Moreover, through a careful design of a search heuristic, we can preserve this efficiency even with a node outdegree that is considerably higher.

1.5 Outline

The paper is organized into 9 sections. In Section 2 we embark from our problem definition and carefully construct the design of a special search space that our technique is based upon; we discuss its features and requirements. We develop these design decisions in Section 3 into a concrete specification of a search space that satisfies these requirements, a state lattice. In the following Section 4, an implicit representation of the state lattice as a theoretical construct is derived with the goal of practical and efficient motion planning. Section 5 is dedicated to the details of exploiting the properties of this space for building an efficient heuristic search algorithm. In Section 6 we analyze the performance of the presented motion planner using some standard benchmarks and also in comparison to basic grid search. Section 7 is dedicated to a discussion of concrete robotics applications where this work can be quite beneficial. Section 8 summarizes the paper. We conclude in Section 9 with closing remarks, and future plans for this research.

2 Search Space Design

This section presents a progression of design principles that, when followed, result in the creation of a search space which is well-suited to motion planning problems subject to differential constraints. The search space that results satisfies such constraints by construction, enabling path planning to proceed without considering them at all. The resulting solution path is therefore always guaranteed to be feasible.

2.1 Conversion to a Sequential Decision Process

Perhaps the most basic and powerful step in the practical formulation of the motion planning problem is its conversion to a sequential decision process. This conversion occurs by instantiating both a process which samples space and a process which selects controls that generate the desired motions between these samples. The search space then consists of a set of states and a set of controls which connect them.

Such a representation defines an embedded graph where the states are the nodes and the controls are the edges. The *indegree* and *outdegree* of a node are defined

respectively as the number of edges entering and leaving the node. In this graph, motion planning becomes a process of making decisions (of which control should be executed next) at the nodes, as those nodes are encountered during the search for a solution. It is often possible to elaborate the search space in a lazy fashion, as the search proceeds, leading to memory requirements which only grow directly with the number of nodes encountered.

2.2 Configuration and State Spaces

One of the basic operations of motion planning is that of testing a candidate path for collision, or more generally, for its utility or cost score. For this operation to be well defined, the paths to be tested for collision must be expressed in configuration space (C space) so that the position of every part of the vehicle is fully determined everywhere on a path. It is almost universal to formulate motion planning problems in configuration space, consisting of perhaps position and heading (x, y, θ) in the plane or in the local terrain tangent plane.

However, C space is often not of sufficiently high dimension to represent all of the differential constraints which a feasible path must satisfy. For the purpose of encoding such constraints, state space, which might add dimensions of curvature, κ, or velocity, v, to configuration space, is needed.

2.3 Sampled Representations and Spatial Equivalence Classes

In problems where analytic representations are not convenient, a second useful step in problem formulation is to establish spatial equivalence classes in order to avoid attempting to search the entire continuum. Under this approach, an arbitrary state is identified with some canonical state which represents all of the space in the local region around it. This discretization can be accomplished either by an explicit association or by a choice of controls (edges) which guarantees that only the canonical states will be encountered as decision points during the search. It is natural to extend this concept to apply to paths as well. If regions surrounding the endpoints of a path are considered equivalent, the same rule could apply to all states forming the path. This leads to an equivalence class of paths that includes all that can be contained within a contiguous region in configuration space (Figure 1). Consider two paths τ_1 and τ_2 with identical endpoints. Path τ_1 will be considered to be equivalent to path τ_2 if τ_1 is entirely contained within a region Q around τ_2. The region Q is defined as the set of configurations within a distance δ determined by some metric ρ:

$$\forall q \in \tau_1, \forall q' \in \tau_2, Q = q' : \rho(q, q') < \delta \tag{1}$$

One of the most basic trade-offs in sampling-based motion planning is that between the computational complexity and the sampling resolution. In return for the reduced complexity of lower resolutions, we must accept the possibility that solutions requiring close approaches to obstacles cannot be resolved as resolution decreases.

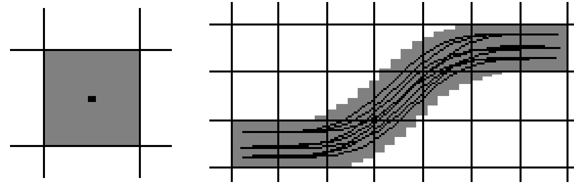

Figure 1: Spatial Equivalence Classes for Points and Paths. All of space in a rectangloid in C space is placed in an equivalence class represented by the point at the center. Similarly, all paths contained within the swath swept by a cell following a path can be placed in an equivalence class of paths.

2.4 Regular Lattices and Grids

Although there are alternatives of a probabilistic nature, this work considers deterministic sampling mechanisms which produce regular lattices of states. A principal advantage of regular sampling is the fact that any control which joins two given states will also join many other pairs of identically arranged states, because such pairs occur throughout the search space. Through an application of the same argument to all paths joining any node to its neighbors, it becomes clear that the same set of controls emanating in a repeating unit from a given state can be applied at every other instance of the repeating unit. Therefore, in this case of a regular lattice, the information encoding the connectivity of the search space (ignoring obstacles) can be precomputed, and it can be stored compactly in terms of a canonical set of repeated primitive motions, henceforth referred to as a *control set*.

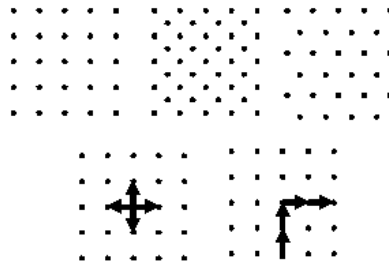

Figure 2: Regular Lattices. Top: rectangular, diamond, and triangular (hexagonal) lattices. Bottom Left: Controls for a 4-connected lattice. Bottom Right: Discontinuous heading change.

The simplest case of a regular lattice is a rectangular grid (Figure 2). In this case, it is common to use a control set consisting of north, south, east, and west motions to create a ”4-connected” graph. While this is straightforward, the transitions between controls that occur at the nodes can be problematic in practice. Consider, for example, a path which enters a node from the south (i.e. heading north) and then leaves toward

the east. When heading changes across a node in a generated path, its execution at any nonzero linear velocity implies an instantaneous (and therefore impossible) heading change. Similarly, even a transition between two different curvatures at the same heading is infeasible at nonzero velocity. The elimination of such infeasible plans requires a mechanism to enforce continuity at the nodes.

2.5 Feasible Lattices

If the connections between nodes in a graph are to represent feasible motions, paths leaving a node toward the north (implied by entering from the south) will not be able to connect via the same paths to exactly the same states as paths leaving toward the east. If two states at the same position but of two different headings are connected differently to their neighbors, they are distinct states from a sequential planning perspective. If they are distinct states, entry and exit headings will agree by construction. Pure rotations, if possible, would be represented explicitly in the graph by appropriate edges.

By this argument, enforcing heading continuity requires that heading be included among the dimensions of the search space. Similarly, if velocity or curvature continuity were desired, these dimensions would also have to be added, as was hinted in Section 2.2. Note that inclusion in the dimensions means only that (x_1, y_1, θ_1) and (x_1, y_1, θ_2) will be considered distinct states - not that either will necessarily ever be explicitly stored in memory.

Once a node and its neighbors are considered to be points in configuration space, the node at (x_i, y_i) becomes a line of nodes arranged along the new heading axis. If the embedded graph is to encode any optional paths, the number of edges in the graph must also grow so that each of the new nodes has an outdegree greater than unity. Since the start and the end of an edge may now have different headings, the controls involved no longer result in straight lines. A simple example of such a search space is the reachability graph of a special version of the Reeds-Shepp car (Figure 3).

3 The State Lattice

The search space will be called a *state lattice* and it will be constructed according to the principles outlined above. The concept is similar to a grid but:

- 3D, 4D or even higher dimensional spaces will be used in order to enforce continuity constraints at nodes (i.e. to avoid 90 degree turns). In this way, transitions between controls will remain feasible.
- States will be connected to some of their immediate neighbors using only feasible motions. In this way, transitions between states will remain feasible.

On the basis of the above, the search space will encode only feasible motions. States are chosen to exhibit the regular structure of a lattice. For example, for the 4D state space consisting of position, heading, and curvature, this formally means that if the motion:

$$[x_0, y_0, \theta_0, \kappa_0] \rightarrow [x_f, y_f, \theta_f, \kappa_f]$$

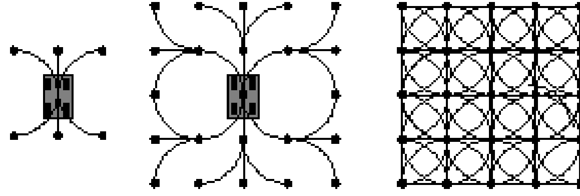

Figure 3: A 3D Search Space. The Reeds-Shepp car can move forward and backward and it can drive straight or turn left or right at a fixed curvature. This control set is derived from these basic controls by carefully choosing their length. Left: The carefully chosen control set precisely hits nodes in a rectangular grid. Center: The reachability tree to depth 2. Right: The reachability tree obtained by copying the control set at every node in a C space with 4 headings. Each dot represents 4 distinct nodes. While this search space will not generate a turn exceeding the maximum curvature, and while heading is continuous across nodes, the instantaneous transitions among curvatures at the nodes do not respect steering rate limitations.

is feasible, then so is the motion:

$$[x_0 + i\Delta x, y_0 + i\Delta y, \theta_0, \kappa_0] \rightarrow [x_f + i\Delta x, y_f + i\Delta y, \theta_f, \kappa_f]$$

for any integers (i, j) and steps $(\Delta x, \Delta y)$ corresponding to the size of the repeating unit. Rotational symmetries will be similarly exploited.

3.1 Modeling Vehicle Maneuverability

The generation of a control set requires a capacity to solve the two point boundary value problem of *trajectory generation* which can be formulated as follows. Let x denote the state vector of the system and let u denote the control input. The vehicle system satisfies some nonlinear dynamics which are generally of the form:

$$\dot{x} = f(x, y, t) \tag{2}$$

This differential equation embodies the *vehicle model* - how the vehicle moves based on its control inputs.

The trajectory generation problem is that of solving for a control which causes the vehicle to move from a start state x_0 to some terminal state x_f. The balance of the paper will assume the existence of such an algorithm. The solution used in the implementation discussed below is based on [26].

While it is not necessarily the case that only one distinct feasible motion exists between two well separated states, the trajectory generator will be configured to produce exactly one, or report failure (meaning that no such motion exists). This limitation may be relaxed by computing a family of solutions and retaining only the sufficiently distinct members. For example, in the case of a 4D configuration space for an Ackerman steer vehicle, a unique solution is generated by finding cubic curvature polynomials,

e.g. equation 3. Such path parametrization will be used in the examples in this report, unless otherwise noted.

$$\kappa(s) = a + bs + cs^2 + ds^3 \tag{3}$$

Edges in the graph can be annotated with the parameter vector of the polynomial in order to describe the motion between the involved states. Many other alternative representations are possible with this trajectory generation algorithm. Using the trajectory generators mentioned above, trajectories are computed in sub-millisecond times. Thus, thousands of trajectories can be processed in a reasonable time period to produce a control set off-line for a specified vehicle model and choice of configuration parameters.

3.2 Desiderata and Design Approach

Numerous design freedoms related to the layout of the states and the controls remain even after the above guidelines are followed. It therefore becomes natural to ask: which design is best in some overall sense? Two distinct designs could be compared on the basis of:

1. Directivity: How short are shortest encoded paths (in the absence of obstacles) to those that would be generated if resolution were infinite?
2. Penetrability: How likely is a given search space to be able to penetrate a field of obstacles?
3. Complexity: How much computation is required to solve a particular planning query?

Typically trade-offs exist between all of the criteria in any design problem. The optimal design problem is that of finding the best design given some, potentially application dependent, weighted measures of them all. For the sake of generality, no optimal design will be attempted here, because the basis for such optimality would require assuming a particular application. However, the design will at least be somewhat principled.

A rectangular grid is chosen in part to enable comparisons to other planners later in the article. Then, on the basis of item 1., it seems that non-uniform heading sampling is the best approach with no significant negative consequences, as described in the following section, so it will be adopted. In an attempt to address both 2. and 3., the paths will be organized into equivalence classes based on their mutual separation. The elimination of all but one path in each class will achieve similar penetrability to the entire class but with much reduced computation. The resulting control set will then be minimal with regard to the definition of equivalence used.

3.3 Sampling the Heading Dimension

Once heading is a bona fide dimension, the question of how to sample that dimension arises. The search space in Figure 3 is incapable of generating a path, or a portion of

a path, which is oriented diagonally. Clearly this can lead to inefficient plans when the start and goal configurations lie on or near a diagonal. This consideration is also a reason to prefer an 8-connected grid to a 4-connected one.

However, given the advantage of more heading samples, it becomes natural to ask: why stop at 8? Indeed for any sampling, there always remains a direction which is reproduced with least efficiency in the sampled representation. Each heading sample creates new options for connecting neighboring states more directly. However, this advantage comes at the expense of increased planning computation due to the increased outdegree of each node in the graph (the extra headings make more nodes available to be reached from a given node).

In a rectangular grid, regular sampling in heading leads to incapacity to generate straight lines in any direction other than principal and diagonal ones. An irregular sampling of heading is better from this perspective. Generally a line at orientation θ will intersect a node if it satisfies $\theta = \arctan(i, j)$ for any two integers i and j. Hence, an irregular sampling of heading is preferred for a lattice, where the nodes are arranged in a rectangular grid.

4 Primitive Control Sets

4.1 Extended Neighborhoods

The search space in Figure 3 is useful, because a simple set of controls can be used to search the entire, unlimited state lattice, while it is not even explicitly represented. Furthermore, the precise relationship between controls and states makes it possible to redirect back-pointers in heuristic search while maintaining continuity of the solution. Despite this, this control set is not practical in a dense obstacle field. The sampling density ideally should not be driven by the controls, but by the obstacle density (the narrowest corridor that must be traversed). If the sampling density is increased, the nodes move closer together. Then, it becomes impossible to connect a state to any of its immediate neighbors other than the one to which it is pointing, due to an upper bound on curvature - which might arise for mechanical or safety reasons.

With an outdegree of unity, the search space then collapses into a set of independent disconnected straight pathways. In order to maintain a discrete search space, it becomes necessary to permit nodes to connect to other nearby nodes which are not immediately adjacent. Once such extended neighborhoods are permitted, the graph generated by applying the resulting control set at every node makes it possible to generate complicated maneuvers such as those used in parallel parking.

With this intuition we will now build upon the developed concept of a control set for a Reeds-Shepp car and extend it to the general case. Once the extended neighborhoods are permitted in control sets, a few issues arise. It remains to be determined:

- how far the neighborhood should be extended.
- whether a central node should connect to all or only some of the neighbors in the extended neighborhood.

We will address these issues in the following sections.

4.2 Redundant Controls and Path Decomposition

When all local neighbors are connected to a central node, redundancy and near redundancy of controls becomes more apparent and more prevalent as the size of the neighborhood increases. In Figure 4, for example, under a policy of retaining only one path in an equivalence class of paths (Figure 1), the long control in the left figure would be declared redundant and eliminated. The motivation for doing so is that significant computation can be saved while not losing the capacity to generate any paths which are sufficiently distinct from the ones encoded in the reduced control set.

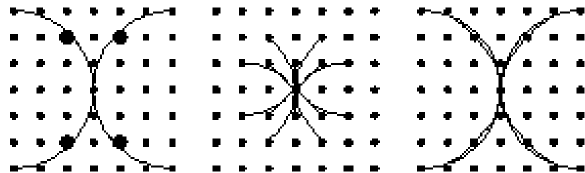

Figure 4: Redundant Controls. Left: The enlarged nodes are almost on the path of a control which is bound for a more distant node. Center: If all of the local neighbors are connected to the center, these particular controls will also be present. Right: A composition of two of the controls in the center figure is a very good approximation to each control in the left figure.

Guided by this motivation, we define a process of *path decomposition*, whereby if an edge between two nodes is sufficiently close to a third node (in position, heading, and any other state variables) somewhere along the edge, the edge is considered to be decomposable into two component edges. The first component edge joins the start to the intermediate node and the second joins the intermediate node to the end node (see Figure 5). Strictly speaking, two trajectories with nearly coincident endpoints are not guaranteed to be nearly coincident everywhere along their length unless the trajectory generator or the representation has this property. If this is not the case, the test for decomposition can include an explicit computation of the distance between the old and the new paths.

We also introduce a *decomposition threshold*, which denotes a certain distance using a metric in effect in the given state space. If a path comes within this distance to a lattice node, it is declared decomposable at that node. The process of path decomposition is strongly related to path discretization treated in Section 2.3. In fact, the path is decomposed only if the concatenation of two sub-paths belongs to the same equivalence class. The sub-paths can themselves also be decomposed recursively.

4.3 Generating Primitive Control Sets with Path Sampling

The discussion thus far has treated the state lattice as an operational data structure used to represent a state space as a graph. In the *generative* formulation used so far, the lattice is constructed from the control set by copying the control set to all regularly arranged nodes. This section takes an opposite *reductionist* view. In this view the

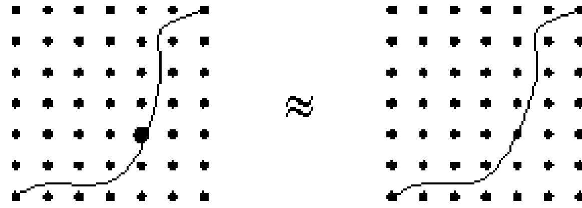

Figure 5: Path Decomposition. Left: The enlarged node is almost on the path. Right: The composition of two controls which pass precisely through the node is in the same equivalence class as the original path so it is used to replace the original. Since the entire path is displaced at the point of decomposition, a closeness threshold smaller than a cell applies when comparing paths instead of points.

control set is constructed by systematically eliminating all redundant motions from a conceptual ideal state lattice. This process of *path sampling*, produces a discretization of all of the paths in the ideal lattice. Suppose that this ideal lattice somehow originally contains all feasible motions connecting every state to every other reachable state. Path decomposition can then be used to develop a primitive control set for a state lattice.

Since the structure is regular, the generation of a primitive control set is accomplished conceptually by choosing an origin, since all motions emanating from it, if feasible, must emanate from every other state. In this case an origin is the set of all nodes (of differing headings, curvatures etc) at the same position. Next we choose a decomposition threshold, and sequentially decompose all feasible motions emanating from that origin. Any generated sub-paths that cannot be further decomposed are stored in a unique set of primitives (i.e. repetitions are discarded). When no more motions are decomposable, the unique set of remaining primitives is the control set which encodes all feasible motions up to a resolution consistent with the decomposition threshold used for path sampling.

Clearly, this process will have discretized and approximated the ideal state lattice, but the hope is that the fundamental expressiveness will have been retained while drastically reducing the complexity of the search space. To a constructionist, the real test is to now regenerate the lattice from the control set and compare it to the original. The path discretization encoded in the decomposition threshold controls the fidelity of the reconstruction. A smaller threshold leads to more controls and a better reconstruction. Conversely a larger threshold leads to less controls and a poorer reconstruction. The selection of a good decomposition threshold depends on the application. At one extreme, the decomposition threshold should be somewhat less than the separation of the lattice nodes. A value larger than this will produce effects similar to those mentioned in Section 4.1: the search space will collapse into a set of independent disconnected straight pathways.

In addition to its consistency with state sampling, path sampling can be justified by the behavior of real robots: path sampling thresholds can be related to the non-zero path following error of physical vehicles. Path following error must be accepted at execution time anyway, so it is wasteful to spend computing resources computing a

motion plan that is overly precise.

As was mentioned, the control set will be used in the node expansion of the search process. The number of controls is the node outdegree. Therefore, there is a clear trade-off between the quality of the control set and the efficiency of search. The decomposition threshold is a parameter that a designer can control in favoring either the quality of representation or the speed of search. This speed-resolution trade-off is pervasive in discrete motion planning. A reasonable initial guess for the decomposition threshold that we have verified experimentally is an order of magnitude less than the distance between lattice nodes.

At this point we have described a conceptual process to decompose an ideal state lattice into something practical. However, it is not clear how this ideal lattice can be produced before it is pruned, and it is not clear that a finite control set necessarily results from the above decomposition process. In the following section we present the algorithm to obtain a practical set of primitive controls based on two additional assumptions. First, we reduce the size of the ideal lattice to that of a local neighborhood of a canonical node, but that neighborhood is shown to be enough to approximate the entire space. Second, we decompose only the unique motion to each neighbor which is produced by our trajectory generator. This second assumption can be removed with additional effort to elaborate the space of all distinct motions with fixed end states.

4.4 Automatic Generation of the Control Set

To recap, given a suitable trajectory generator, it becomes possible to connect any state in a lattice to all of its immediate neighbors by a feasible motion, if such a motion exists. In this section we present a practical formulation of the algorithm for generating a control set.

A key concept of this formulation is *radiation* from the central node. Suppose the control set edges are generated in order of increasing radius from the origin. Suppose further that each edge is tested immediately after generation for passing sufficiently close to an intermediate node. If it does, and if the component motions are feasible, then these component motions must already be in the control set because they must be shorter and all shorter feasible motions have already been generated or themselves decomposed. Radiation also solves the question of how large the neighborhood should be because, eventually, and barring a few pathological cases, all edges are decomposable at some radius and the process terminates.

The process starts at the central node set. This set includes all of the nodes at the center position with different headings. First, every node in the central set is tested against every immediately adjacent node (those of Manhattan distance of unity) to see if a feasible motion connecting the two exists. There are 4 immediately adjacent positions in a rectangular grid, and each position will have a node for each heading sample. Next, the central nodes are tested against the nodes which are one more unit away. When a feasible motion exists, it is tested to see if it passes sufficiently close to an intermediate node to decompose the path. There is no guarantee that the two components are feasible, but if they are, and if their composition is sufficiently close to the original, the original is discarded. The essential algorithm is as follows:

Algorithm 1

```
01 ControlSet = Empty
02 Qi = Lattice Origin
03 R = 0 // radius away from Qi
04 do{
05     R += 1;
06     FoundAtLeastOnePath = false
07     for each node Qf s.t. Manhattan(Qi,Qf)==R
08         OrigPath = FindInLattice(Q1,Qf)
09         if(OrigPath was found)
10             qs = Qi
11             do {
12                 qs=GetNextSampleAlongPath(OrigPath)
13                 Nnear = GetNearestLatticeNode(qs)
14                 SubP1 = FindInLattice(Qi,Nnear)
15                 SubP1 = FindInLattice(Nnear,Qf)
16                 if((SubP1 found OR SubP2 found) &&
17                   OrigPath == (SubP1+SubP2))
18                     break // goto 07: New OrigPath
19             } while (qs != Qf)
20             Add OrigPath to ControlSet //keep
21             FoundAtLeastOnePath = true
22         end if
23     end for
24 } while(FoundAtLeastOnePath)
```

On line 08, the path between the origin and a chosen node is verified to be feasible. This could be done either by looking for the corresponding connection in an existing lattice or by using a trajectory generator. The function `GetNextSampleAlongPath` on line 12 steps along the path in small increments. These candidate points of decomposition are checked on lines 16-17. If the subpaths are feasible and their concatenation is equivalent to the original path, then the decomposition is successful, and the algorithm moves to the next node. The variable `FoundAtLeastOnePrimPath` is a flag that indicates whether any primitives were added at this value of Manhattan radius R. If not, then all paths at this radius have been decomposed, and the algorithm terminates.

A few subtleties are not apparent in this code fragment:

- For vehicles of limited turn radius, it can be expedient to avoid generating edges whose headings differ from the start node by more than, say, 90 degrees. Such edges will not be a sufficiently direct connection of two nodes to qualify as a primitive motion. Most will be decomposable anyway.
- Again for vehicles of limited turn radius, the radius from the origin is not guaranteed to be monotone with arc length. In these cases it may be advisable to sort the nodes by arc length rather than Manhattan distance in order to satisfy the assumptions underlying the basic induction. Since arc length is not known until trajectories are generated, this approach requires that all be generated, stored, and sorted before the algorithm starts the elimination pass.

- The test that the decomposition matches the original is intended to mitigate two effects. First, it may not be the case that two trajectories with similar endpoints are sufficiently close together everywhere to be deemed equivalent. It depends on the properties of the trajectory generator. Second, the equivalence test is not necessarily transitive. If one trajectory is decomposed into two others and one of the two is itself decomposed, the path differences accumulate. The test `FindInLattice` must be designed to return true even when the tested path was decomposed.

An example control set generated with this algorithm is provided in Figure 6.

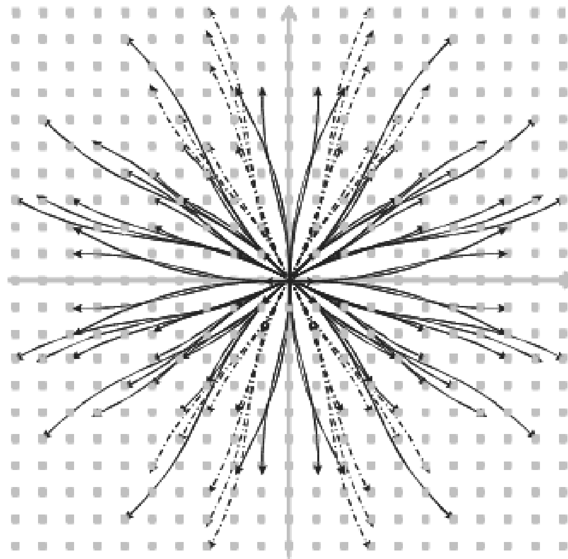

Figure 6: Ackerman Steer Control Set. There are sixteen distinct nodes at the origin corresponding to four nodes at multiples of $0°$, four at reflections of $26.6°$, four at multiples of $45°$, and four at reflections of $63.4°$.

5 Motion Planning using Control Sets

This section is devoted to a detailed discussion of constrained motion planning using control sets. As was mentioned earlier, the essential principle of the present approach is that, with the definition of the search space we have constructed, motion planning is reduced to unconstrained search in this space.

5.1 Search Algorithm

Since the control set is an implicit representation of the search space, the search graph can be incrementally constructed so that only the states that are explored by the search

are stored in memory. Any systematic graph search algorithm is appropriate for finding a path in the lattice. Many types of search, such as greedy algorithms and dynamic programming, are acceptable. It is typically desired that the planner return optimal paths with respect to the desired cost criterion (e.g. time, energy or path length) and it be efficient. The A* heuristic search algorithm was used in this work because it satisfies these requirements.

A brief review of A* is useful for later reference. A* maintains two state lists, CLOSED and OPEN. Under an appropriate set of assumptions, the CLOSED list consists of states to which the least costly path is known. The OPEN list is the set of states to which paths have been found that are not known to be the least expensive; it is sorted by the estimated total traversal cost from start to goal via each state in the list. During each step of the search, the state with the shortest path estimate is removed from the OPEN list, expanded, and moved to CLOSED. The expansion process involves following each outgoing edge to its terminal state, which is added to the OPEN list in turn. Since the graph is dynamically constructed, it is not known beforehand which edges in the control set collide with obstacles. During each expansion, the cost of each edge is computed by *convolution*, a process of placing the vehicle frame (i.e. its geometric model) along the path in short successive intervals and sampling the cost of each pose by integrating the cost over the area covered by the vehicle. Convolution is performed in the workspace rather than by performing a line integral in C space because in the general case, computing the appropriate C space expansion would be more costly.

Next we analyze the features of the motion planner based on such search, and in later sections we look at portions of it in more detail. Considerable attention is devoted to estimation of the heuristic in Section 5.5. A heuristic which is as accurate as possible dramatically improves search efficiency.

5.2 Qualities of Motion Planning in State Lattices

In previous sections we have discussed in detail that discretization of states, controls and resulting paths plays an important role in our approach. It allows simplifying the motion planning problem on many levels: from transforming the continuum of paths into a discretized state lattice, then into finite set of primitive controls. In this progression, we observe that the quality of this discretization is dictated by the chosen resolution. If, hypothetically, the resolution were allowed to increase without limit, it is not hard to see that the state lattice and the corresponding control set, if recomputed at each resolution level, would in turn approach the continuum. Thus, we conclude that the presented method of motion planning, based on a systematic search, is resolution-complete.

Another important quality to be considered is optimality (with respect to path length, since a kinematics vehicle mobility model has been considered so far). This property is also closely related to the notion of discretization, and the corresponding resolution, that permeates this work. We have seen in Section 4.3 that the choice of decomposition threshold affects the quality with which a control set represents the ideal state lattice, and so it is with the resolution of the corresponding discretization. If the limit of this resolution were set at infinity, we would obtain a perfect representation of the ideal state lattice at infinite resolution - the continuum. Therefore, assuming that

an admissible heuristic is used in the search, we conclude that the present approach exhibits *resolution-optimality*. It is possible, in principle, to plan paths that approach optimal paths arbitrarily closely.

In practical implementations, these considerations translate into a capability to efficiently generate motion plans that are optimal up to the chosen discretization. The user has the power to control the closeness to optimality through the decomposition threshold parameter. This is another benefit of the approach, and a novelty, to our knowledge. In an extended experimental study described in Section 6, we have observed that the algorithm typically returns paths that are practically indiscernible from optimal.

5.3 Implementation Considerations

In this section we describe several techniques that are helpful for improving the efficiency of the implemented motion planner.

If the terrain shape is known sufficiently well at planning time, the terrain-aware version of the trajectory generator [26] can be used to generate unique versions of each edge encountered during the search. This is, of course, more computationally expensive than assuming flat terrain and using the precomputed, terrain independent control set, but it is important to note that the planner incorporates an integral trajectory generator which allows it to function correctly in rough terrain. In most realistic situations, the environment is not known perfectly at planning time. In the case of terrain shape, it is practical to regenerate each of the edges of the solution path as the terrain involved enters the field of view of the terrain sensors. It is possible in the worst case for an edge that was created for flat terrain to become infeasible on rough terrain. However, this case is no different than the appearance of an unknown obstacle, and a replanning strategy would presumably already be in place to handle such events.

When velocity is added to the state space, the planner can also plan high speed motions using a control set that is aware of dynamic effects. Turns would become progressively less ambitious as velocities increase but a different topology would emerge naturally in the control set at the higher speeds. Of course, the approach is not immune to the curse of dimensionality but such high dimensional spaces could be employed at least for local obstacle avoidance planning.

The performance of our implementation of A* was improved by a modification to the expansion step. Convolution takes a substantial portion of the algorithm's runtime, as is the case with other sampling based planners, including probabilistic roadmaps [51]. Convolution was delayed in our implementation by substituting the curve's minimum possible cost for its score, until its terminal state is selected for expansion from the OPEN list. At this point, the edge has its true cost computed via convolution. If it does not run into an obstacle, the score is computed so that it can be compared with the next best state on the OPEN list and reinserted if necessary. This procedure still guarantees optimality since an optimistic estimate of the curve's true cost is used, but it avoids expending unnecessary effort due to the fact that only one edge out of many from each expansion is actually needed for the solution. In environments where the cost density of the terrain is nearly constant, this technique causes little overhead since the true path cost is nearly proportional to path length. This optimization is particularly valuable for the state lattice, where the outdegree may be high and edges can be rela-

tively long. While in general, it is necessary to evaluate full cost of all members of the OPEN list, we found that in the binary obstacle environments, this technique resulted in appreciable speed-ups.

Lastly, to improve the efficiency of our implementation of A*, we found it useful to use a dual heap-table data structure for OPEN and CLOSED. The heap was used for all typical operations, while the table was set to contain a copy of the heap's data. Each cell of the table was one float for the cost of the corresponding lattice node. In such a way, queries to the CLOSED list as to whether a node was previously visited could be computed in constant time. The overhead cost to maintain this data structure were shown to be much less than its gain.

In order to compute the overall cost of a node in the OPEN list, A* also requires a heuristic function to estimate the remaining cost-to-goal. For optimality, A* requires that the heuristic must return an underestimate on every possible query. For efficiency, the estimate should be as close to the true cost as possible. In the case of the state lattice, this turns out to be a difficult problem, though a surmountable one, as outlined below.

5.4 Heuristic Cost Estimate

The heuristic function required by A* estimates the cost-to-goal from any state encountered during search. In estimating this cost, most heuristics cannot take terrain variation into account. In the absence of such terrain-specific cost information, the best that can be done is compute a minimum cost that is directly proportional (by minimum cost per unit distance) to distance traversed. Therefore, distance will be used interchangeably with cost in the rest of this discussion.

Among the simplest options for a heuristic estimate in the state lattice is the Euclidean distance metric. This function is computationally efficient and it satisfies the admissibility requirement of A*. However, in many cases, it vastly underestimates the true path length in the lattice, resulting in inefficient search.

A heuristic for a vehicle with limited turning radius moving in the plane could be derived from methods of Reeds and Shepp [49] using only three controls: maximum left, maximum right and straight. The authors offer a list of 48 formulas which enumerate the shortest path for particular configurations of start and goal states. Cherif [18] also used a heuristic consisting of CSC paths (where C is a circular arc of minimum admissible radius and S is a straight line).

However, even the Reeds-Shepp heuristic is too much of an underestimate for some lattice control sets, and hence the resulting search is not as efficient as possible. The effects of discretization on path optimality, considered in Section 5.2, contribute to the fact that paths in the lattice may be slightly longer than what a vehicle moving continuously in the plane could actually accomplish. Additionally, the Reeds-Shepp paths are discontinuous in curvature (i.e. infeasible to execute), resulting in further underestimates. Therefore, to generate the perfect heuristic distance function for the state lattice, it is necessary to incorporate information about the structure of its control set.

5.5 Heuristic Look-Up Table

Given ample computational resources, a straight-forward and, more importantly, an effective way to predict path costs is to pre-compute and store the actual costs that planner will need, using the planner itself. Such a Heuristic Look-Up Table (HLUT) can be imagined as a large multi-dimensional array of real-valued query costs. Assuming that such a table could be generated, the invocation of the heuristic function becomes a simple table dereference.

Like most other heuristic functions, the HLUT cannot take into account obstacles or other terrain variation when precomputing queries, so an obstacle-free policy must be assumed. The HLUT discussed here is designed for an Ackerman steered vehicle, but the techniques are applicable to any control set for any type of vehicle with any kind of differential constraints. The notion of an HLUT suggests some issues to be considered, namely the utility of a table of reasonable size, the amount of time required for generating it, as well as the policy for heuristic queries which are not in the table.

5.5.1 Constructing the Heuristic Look-Up Table

The memory issue can be addressed merely by exploiting symmetries in the lattice control set to reduce duplicate information. Those symmetries include translation, rotation and reflection (Figure 7). Because the discretization of the lattice is regular in position, all queries can be treated as if they originate from the origin. Only a subset of possible initial headings needs to be generated. For example, if the control set exhibits 8-axis symmetry, it is sufficient to precompute only 1/8th + 1 discrete initial headings. Finally, many goal states are redundant. For instance, with an initial heading of zero, there is symmetry about the x-axis. With all of these massive space savings, a useable size HLUT can be stored in 200,000 entries, consuming perhaps 2.5MB.

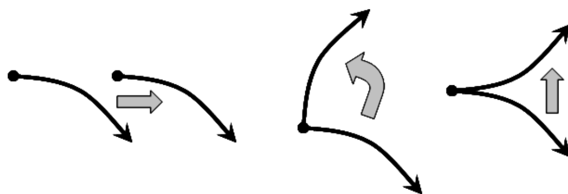

Figure 7: Symmetries of the State Lattice. Many different lattice edges and combinations of edges are equivalent under a small set of transformations. Exploitation of these properties dramatically reduces the total size of the HLUT. Symmetries shown from left to right: translation, rotation, and reflection.

To generate such a table requires solving many A* queries. If an average A* lattice planning query with the Euclidean heuristic takes about 0.1 second, then generating 200,000 entries by the most naive method would require more than five hours. Fortunately, several properties of A* allow the table to be populated much more efficiently. When a search is performed, much more is learned about the graph than just the final

path length. Every time a state is expanded and put on the CLOSED list, the lowest cost path to that state is known, so another optimal query cost can be inserted into the HLUT. Secondarily, it is wasteful to delete the A* lists and start from scratch between queries, as these lists contain valuable state. The CLOSED list can be reused as-is, but the OPEN list is sorted according to distance estimates to the old goal state. Depending on the size of the OPEN list, it may be cheaper to delete it and begin anew or to recompute each estimate according to the new goal location. For a particular implementation, this cut-off was found to occur at approximately 150,000 OPEN states, so that several queries can be processed in a row before the lists need to be expunged. Finally, the order in which queries are populated in the HLUT has a significant effect, because the exact solutions of earlier queries can be used as a heuristic in later queries. The issue of population of the table is considered below.

The look-up table cannot be infinite in size. Therefore, some queries will occur for which no entry exists in the table. An alternative *backup heuristic* such as Euclidean distance must, of course, already be in use for purposes of generating the HLUT. It can also be used to satisfy queries missing from the table during planning. Doing so is easily justified by the fact that Euclidean heuristic gives better approximations on more distant queries.

When generating the table, there must be some termination condition. Such a condition should be automatic, principled, and tunable according to the needs of various applications. This issue really breaks down into two separate questions. First, which queries should be included or excluded? And second, how many entries should be included?

To answer the first question, it is helpful to define the *trim ratio* as the ratio of the backup heuristic's estimate for a particular query to the true path cost which the HLUT would include for that query. There are two considerations for including a particular entry in the HLUT. Queries with low trim ratios should be included because the backup heuristic does a poor job of estimating them (hence, they should be "trimmed" last). Secondly, frequently needed queries should be included for the sake of efficiency, since failing over to the backup heuristic results in loss of efficiency. Figure 8 shows trim ratios in a slice of the HLUT where initial and final headings are zero for each query. Note that paths into dark regions (of low trim ratio) would correspond to compositions of maneuvers which result in ultimate sideways motion. These are the queries whose costs are most valuable to have stored in the HLUT.

When performing A* searches to populate the look-up table, an important consideration is the order in which queries are performed. Ordering affects the ultimate selection of queries due to the inclusion of all states on the CLOSED list. Also, the ability to reuse previously computed state varies considerably with ordering of queries. Three different approaches to HLUT population are considered below.

5.5.2 Populating the Heuristic Look-Up Table

The simplest method of look-up table population is the naive approach, which iterates through each entry in the table in a raster-scan fashion. The result of A* when run on each query is inserted in order. This is the slowest of the methods which were tested. It does not share significant state between sequential queries, and the most difficult

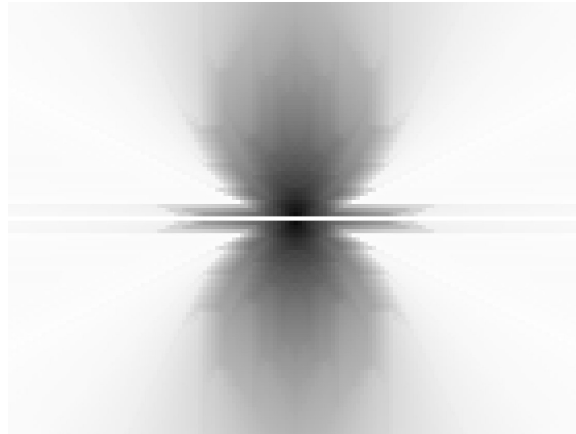

Figure 8: Visualization of HLUT. This cross-section of an example 4D HLUT shows a slice of paths from the origin to various (x, y) positions in which $\theta_0 = \theta_f = 0$. Brightness represents trim ratio, the ratio of Euclidean distance / nonholonomic path length. Dark regions correspond to low ratios.

queries are presented to the planner first, when no HLUT has yet been built up to improve performance.

The table can be populated much faster by the use of Dijkstra's algorithm. In this approach, the search is run in such a way that the next node expanded in the graph is always the unexpanded node closest in cost to the origin. This algorithm very quickly populates many HLUT entries, but those states are the easiest ones to reach, meaning that they generally have higher trim levels (Figure 9). The Dijkstra's search can be terminated at an arbitrary time.

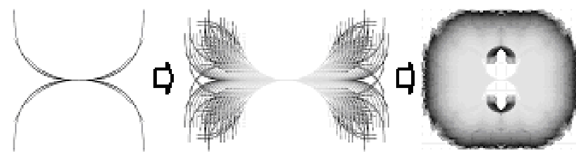

Figure 9: Populating the HLUT Using Dijkstra's Algorithm. Expansion of the lowest cost unexpanded state in the tree is an efficient way of finding the shortest distance to many states at once.

The final method, *brushfire*, is more principled in that it selects first those queries with the lowest trim levels. This algorithm maintains a HORIZON list of queries sorted by trim level, similar to the way in which A*'s OPEN list is sorted by cost. The HORIZON list is initially populated with queries in which both states are at the origin in every combination of initial and final heading. Each is assigned a trim ratio of 0.0,

since that is the Euclidean distance between overlapping points in a plane. During each iteration, the lowest-valued query is popped from the HORIZON list and given to A*. Each neighboring state in the (x, y)-plane (with the same orientation) is pushed onto the HORIZON list and sorted according to its parent's trim ratio. The ratio of the parent state is used since the exact cost of the query is required and it isn't yet known for the child. The brushfire algorithm terminates when a desired trim level is reached. The process is depicted graphically in Figure 10; note that symmetry can be considered to speed up the algorithm. In order for this method to find all states below a given trim level, it must be possible to draw a path outward from the origin through an arbitrary state such that the trim ratio increases monotonically along the path (Figure 11). Brushfire is a faster algorithm than the naive approach because it makes better use of precomputed state, but it is slower than Dijkstra's algorithm because it requires A* list resets.

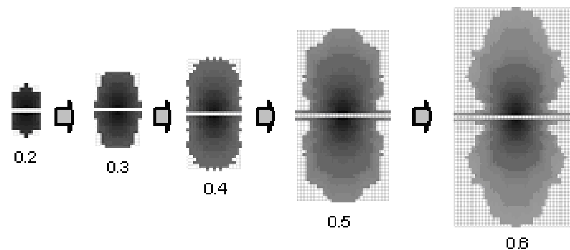

Figure 10: Growth of HLUT in Brushfire. As the HORIZON grows outward, queries of higher trim ratio are incorporated into the HLUT.

The Dijkstra's search algorithms may leave gaps in the HLUT, as shown in Figure 9, while the two slower techniques produce dense results. A gap occurs in the table when an entry is absent but is surrounded by neighboring entries which are included. Gaps are undesirable because they result in less predictable search time among potential A* queries using the heuristic. Furthermore, a major underestimate resulting from falling back to a backup heuristic due to a gap causes a false lead for A*, which would then expend a lot of unnecessary search time.

In order to fully populate the HLUT as quickly as possible while retaining desired properties, a combination of methods can be used. First, Dijkstra's algorithm fills in the majority of the HLUT. Then the gaps are filled in using the brushfire method. In this case, brushfire skips over those queries which were previously filled in to avoid duplication of effort.

Taken together, these techniques are remarkably efficient. Table 1 shows computation time required to generate several different sizes of HLUT on an ordinary desktop computer. Each size look-up table was produced by performing Dijkstra's algorithm out to some size (ranging from 70 to 90 cells depending on the desired trim ratio), and then running the brushfire algorithm to fill in the gaps.

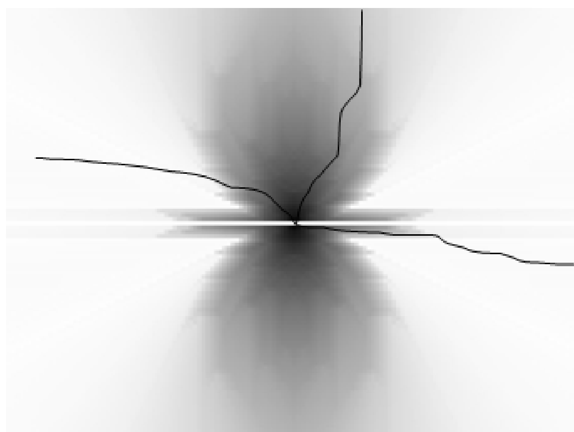

Figure 11: Monotonicity of Trim Level. Several example paths are shown in the (x, y)-plane which are monotonic in trim level. Every point in the plane must have at least one such path which passes through it in order for the brushfire algorithm to populate the HLUT with all queries below a certain trim level.

Trim ratio	HLUT entries	Generation time (mm:ss)
0.6	202,338	1:15
0.7	365,345	3:28
0.8	648,877	13:33

Table 1: Generating the HLUT. Through a combination of techniques, sizable heuristic look-up tables can be generated efficiently. The runs shown here were performed on a 3 GHz Pentium 4.

6 Results

A large set of experiments was conducted on the planner in order to assess its performance. These experiments invoke the planner with a number of queries in a variety of configurations. Here we compare our planner with other algorithms: the basic grid-based planner (on 4-, 8-, and 16-connected grids) and the well-known Barraquand and Latombe (BL) nonholonomic planner [9]. In doing so, we attempted to level the playing field for fair comparison. Therefore, the same implementation of A* was used for all algorithms. The primary differences were in the node expansions, and the cost and heuristic estimates. For the former, we have adopted a notion of a generalized control set: for the grid search it is simply a grid-based node expansion (straight paths), whereas for the BL planner it is the node expansion as described in that work. The first set of tests looks at how the sample lattice control set performs in comparison to other types of generalized control sets. Next the impact of the heuristic look-up table on state lattice performance is examined.

6.1 Experimental Setup

A representative lattice control set was used in all tests. Its state space consisted of the 2D translational coordinates, heading and curvature (x, y, θ, κ). For the sake of simplicity, curvature was constrained to be zero at each discrete state. This control set is depicted in Figure 12.

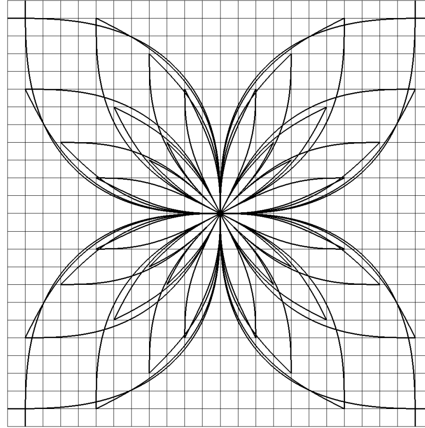

Figure 12: Lattice Control Set Used in Experiments. The state lattice control set selected for testing has 16 discrete headings, a maximum curvature of 1/8 of cell, and an average outdegree of 12, for a total of 192 curves. The straight edges cannot be seen because they are obscured by the longer curved edges.

For these tests, a list of 10,000 random queries was generated, consisting of an initial and final state, also expressed as position, heading, and curvature. The set of queries was generated with the intent to require the planner to produce paths ranging from simple (nearly straight paths) to complex (e.g. parallel parking or n-point turn maneuvers) among obstacles. Each query was tested with a wide variety of configurations, including different obstacle fields, alternative control sets such as a grid or Barraquand-Latombe, and A* heuristic functions.

Start and goal states were produced with a random number generator that provides evenly distributed real values in a requested range. Initial positions were selected at random from the free space in order to produce the maximum possible variety of queries. The goal position was then specified by a randomly selected radius and angle specified in polar coordinates with respect to the start, repeating this step as necessary to ensure that the goal is also in the free space. Initial and final headings were randomly selected, and curvature at end-points was constrained to be zero. Goal states were constrained to be no more than ten minimum turning radii (80 cells) from their corresponding start states when projected onto x, y space.

Each query was tested in two worlds. In both worlds, cost to traverse free space was held constant at 1 unit per cell of free space. Hence, path cost was equal to the distance traveled. The baseline case was an obstacle-free world, meaning that the HLUT gave

exact solutions for cost. Results were also obtained using a world with randomly placed point obstacles, shown in Figure 13 with paths generated by two different planners. These obstacles are the size of one map cell, which in this control set corresponds to 1/8th of the minimum turning radius. Points were generated with uniform distribution and 5% density in the plane.

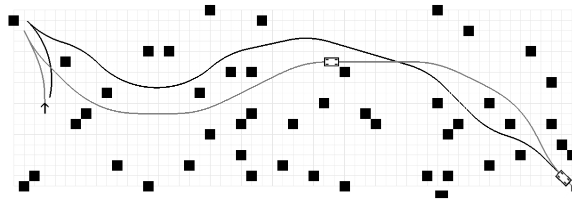

Figure 13: World with Obstacles. A portion of the world with point obstacles used in the experiments is shown here. Two plans are shown which solve the same query. The black line shows the plan generated by the state lattice, while the grey line traces the path returned from the BL planner.

Metrics for performance included time, memory consumption, and the quality of the resulting plan. In analyzing the performance of the planner, it is necessary to distinguish among the spectrum of queries ranging from simple to complex, as was suggested above. In the case of a grid, Euclidean distance would be an appropriate measure of difficulty. When planning in full state space, however, the length of the path followed by a nonholonomic vehicle can vary widely even in cases where Euclidean distance between the end-points is held constant.

In order to quantify the complexity of a particular query, the distinction is made between absolute difficulty, which is proportional to path length, and relative difficulty, which reflects how much the resulting path deviates from a straight line. Figure 14 illustrates the two concepts. Both factors contribute to the overall resource requirements for a particular planning problem. Absolute difficulty is measured here by the path length. In the case of relative difficulty, the ratio of Euclidean distance / nonholonomic distance was used. In this scale, values near one mean that the resulting path is nearly a straight line, while those values nearest to zero indicate that much maneuvering is necessary to reach the final pose. In the analyzes below, results are shown for queries with absolute difficulty of 40 cells. This number was chosen arbitrarily for clarity of presentation. Any other absolute difficulty would have conveyed similar results.

6.2 Comparison of the Lattice to Other Control Sets

In this section, a variety of alternative control sets are considered. In each set of cases, the same framework is used, including an A* planner, heuristics, and world representation. The only distinction is the method by which neighboring states are generated during the expansion step. A standard lattice control set, depicted in Figure 12, was used across all tests. An overview of all the control sets which are considered here is presented in Table 2. Conceptually, a control set with longer edges requires fewer

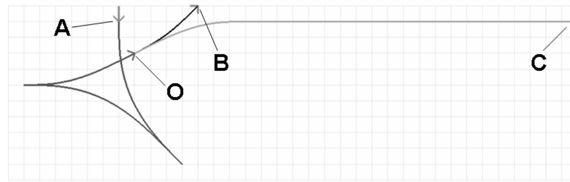

Figure 14: Absolute and Relative Query Difficulty. The difficulty of a query can be quantified in two dimensions. Each path, A, B, and C, starts at O. Query A is high in absolute difficulty as well as relative difficulty because it is long and has multiple cusps. B is simple in both measures. Query C has the same absolute difficulty as A (same length), but the same relative difficulty as B (nearly a straight line).

expansion steps to reach the goal but a greater total length to integrate, proportional to the outdegree. Correspondingly, a higher outdegree increases memory requirements for the planner and computation in the expansion step, but also improves the chance of finding a more direct route in fewer plan steps. Below, the best performing of several combinations is empirically determined.

Control Set	Total Edges	Ave. Edge Length	Ave. Outdegree
original control set	192	8.72	12
ctrl set w/ turn-in-place	224	7.47	14
BL	96	4.00	6
grid-4	4	1.00	4
grid-8	8	1.21	8
grid-16	16	1.72	16

Table 2: A Quantitative Look at the Control Sets. Parameters of a control set have a strong influence on how the planner will perform while using them.

6.2.1 Comparison to a Grid Planner

The basic grid search method of planning does not consider heading, which leads to several drawbacks. First, it is not clear what the heading state is unless two adjacent states on the path are compared and even in this case, the heading can only have 4 values. Second, it is impossible to plan feasible paths for vehicles incapable of a point turn. Third, heading continuity constraints cannot be enforced. A grid has nonetheless been widely used due to its simplicity and efficiency.

But is this approach inherently faster than searching in a full-dimensional state space? For this test, the planner was run with four different control sets, in each case using a heuristic function which always returned the exact distance to the goal. The basic state lattice was matched with a large HLUT. Three grid control sets were tested, in which each state is connected to its 4, 8, and 16 nearest neighbors (Figure 15), using

a perfect heuristic for each level of connectivity.

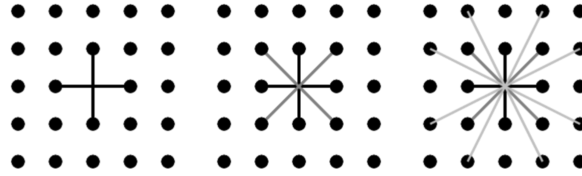

Figure 15: Grid Control Sets. Three different grid control sets were tested. A point is connected to the 4, 8, or 16 nearest neighbors that have unique headings.

Performance of the control sets in the absence of obstacles is shown in Figure 16. In order to normalize the results, only queries of absolute difficulty equal to 40 cells were used. The data are presented across a range of relative difficulty. In the absence of obstacles, very interestingly, the lattice performs on par with basic grid search. Only on most difficult queries, the lattice consumes more CPU time than a grid control set because the nonholonomic path solution diverges more dramatically from a straight line solution. Of course, for more difficult problems, the answer returned by the grid is increasingly infeasible to execute on a real vehicle - and the path is also unlikely to arrive at the correct heading.

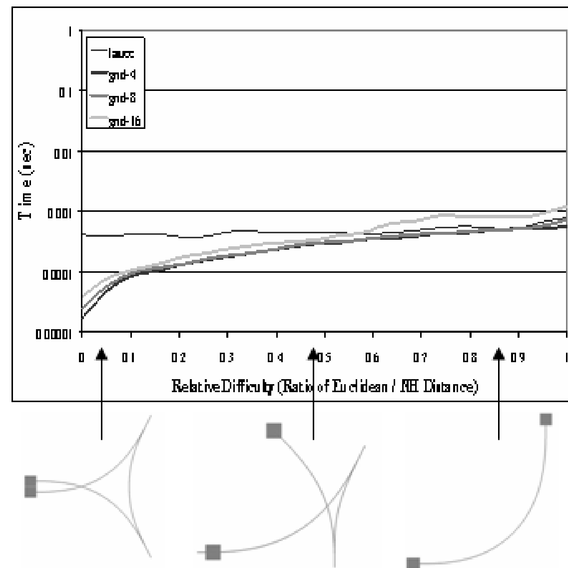

Figure 16: Lattice vs. Grid without Obstacles. Queries with an absolute difficulty of 40 cells are shown. The lattice performs similarly to the grid planners overall. Only for the greatest relative difficulty (nearest zero) does the lattice require more CPU cycles. Of course, the benefit of this extra computation is a plan which is feasible.

Results in the presence of obstacles are shown in Figure 17. It is intuitive that the grid should outperform the lattice in this obstacle field for any class of query. If an obstacle appears in the path of a grid plan, that plan is often displaced by only a few cells in order to circumvent the obstacle. In the case of the lattice under test, however, only smooth continuous paths with a maximum curvature of 1/8 = 0.125 are considered. These requirements substantially limit the planner's options, making it more difficult to find a satisfactory path through an obstacle field as shown in Figure 18. So while the grid path deviates slightly in the cluttered environment, paths generated by the state lattice are often much more sophisticated in order to plan around obstacles. Thus, the difficulty experienced by the lattice planner reflects the true mobility limits of the vehicle. In plain terms, a grid planner produces faster answers, but they are usually wrong. Despite the increased search requirements, the lattice remains consistently only one order of magnitude slower than the grid planner, returning on average in less than 0.1 second for all classes of query.

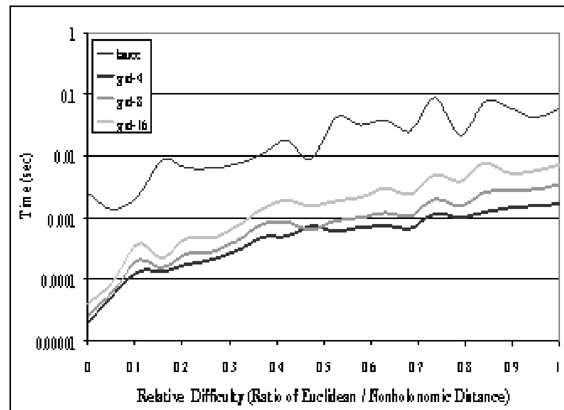

Figure 17: Lattice vs. Grid with 5% Obstacle Density. Queries of length 40 cells are shown. The grid outperforms the lattice, but it usually returns infeasible solutions in a dense obstacle field. Lattice planner runtime is still acceptably fast.

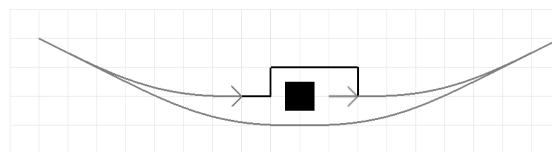

Figure 18: Obstacle Avoidance with Two Control Sets. Grid planners can easily avoid small isolated obstacles, as shown by the black path. By contrast, the grey path has limited curvature and so can be feasibly traversed by a constrained vehicle.

6.2.2 Comparison to Full C Space and Nonholonomic Planners

The comparison between the state lattice and grid-based planner is not entirely fair, since the two planners search different spaces, and the grid results cannot be traversed by nonholonomically constrained vehicles without post-processing. The Barraquand and Latombe (BL) planner is well known and has been a popular nonholonomic planner for over a decade. It has also been used in a variety of real mobile robot applications [5], [46]. In this section, several different implementations of that planner are examined. In all cases, the identical A* algorithm was applied, but the heuristic cost estimate was altered to produce a fair comparison. In the first case, the algorithm was run as documented in [9], with a zero heuristic applied. In the second case, Euclidean distance was used as heuristic. Finally, an attempt was made to produce a perfect heuristic by generating an HLUT for the BL planner, for a fair comparison with an HLUT-enabled lattice planner.

In order to make the BL control set work in a fashion compatible with the state lattice test framework, an adjustment was necessary. Barraquand and Latombe [9] describe the edge lengths as being the L^1 diameter of a cell and assert that their planner requires the discretization of the search parameters to be fine enough. The state lattice's discretization settings were experimentally adjusted to allow proper operation of the BL planner: the edge lengths were set to four cell widths. The resulting reachability tree is shown in Figure 19 along with the standard state lattice tree. Reverse edges were omitted from both trees for clarity.

The performance of BL with three different heuristics is shown in Figure 20 along with the state lattice using two different heuristics. The heuristic which enforces optimality for the BL planner is zero, as proposed by authors, but the computational time required makes this impractical in most situations. In a fair match-up using the Euclidean distance heuristic, the two planners perform comparably, but only the lattice retains optimality (Figure 21). However, note that the Euclidean heuristic was used for the lattice planner only for the fairness of the comparison. The lattice planner using the HLUT significantly outperforms our implementation of BL, both with and without its own HLUT, as shown below.

The HLUT generation algorithm was run using a BL control set in an attempt to produce a perfect BL heuristic. However, in A* runs which used this heuristic, many false leads were still explored. The HLUT generation algorithms discussed above make a fundamental assumption that the cost between two points in the graph depends only on their relative states because the lattice is regular in translational discretization. The BL algorithm, however, does not operate on a regular search space. As edges grow outward, other edges are excluded, such that the path between two points depends on the order in which states were expanded to produce each edge between the two points. So while using the HLUT resulted in the best BL performance, this heuristic cannot be guaranteed to be admissible for that algorithm. Note, however, that with any non-zero heuristics the planner does not guarantee optimality anyway.

An alternative lattice was also tested. This control set has the same controls as the basic lattice, except that extra zero-length edges were added so that the vehicle is permitted to turn in place. The cost of these additional edges was computed based on the aggregate distance of travel of each of the wheels in the vehicle. The effect of this

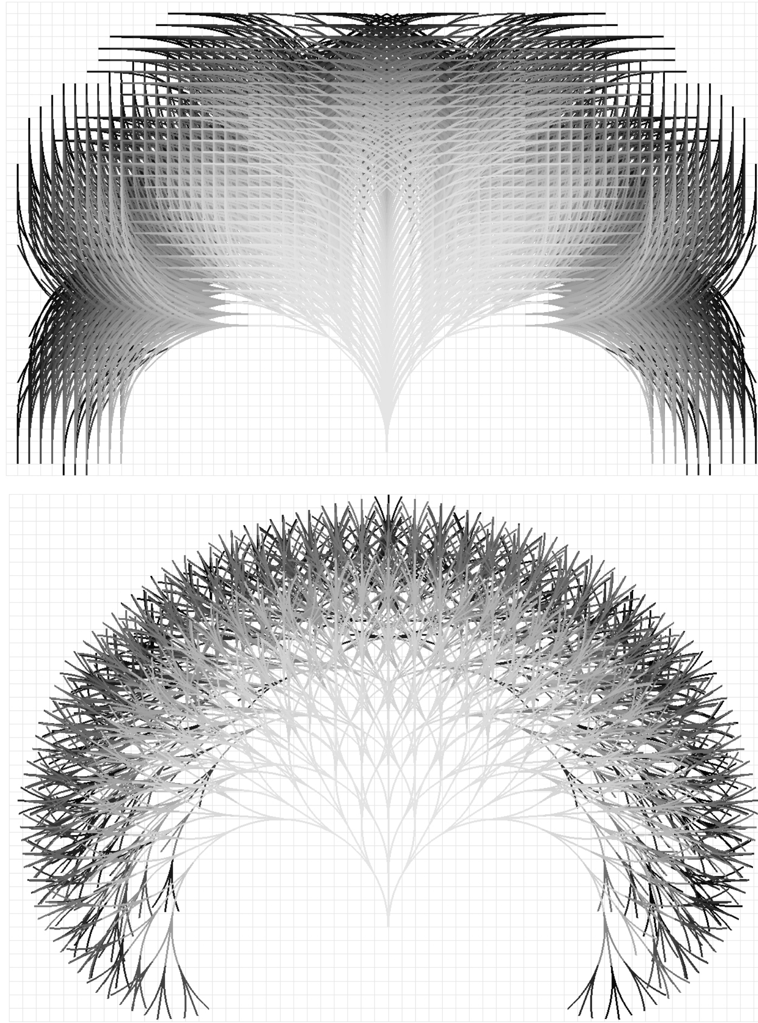

Figure 19: Reachability Trees for the Lattice and BL. At top, the first 1400 expansions of the basic state lattice control set in best-first order. At bottom, the first 1400 expansions of the BL control set were generated using the same algorithm. Lighter edges are older in both images.

capability on resulting plans is depicted in Figure 22, where the path is substantially shortened by avoiding circuitous maneuvering. These additional controls have no significant effect on overall planner performance (Figure 23), but they provide valuable added flexibility in negotiating dense obstacle fields. Moreover, for practical applications, the lattice planner algorithm allows tuning the robot's preference for performing point turns versus smooth maneuvers by assigning appropriate costs to the zero-length

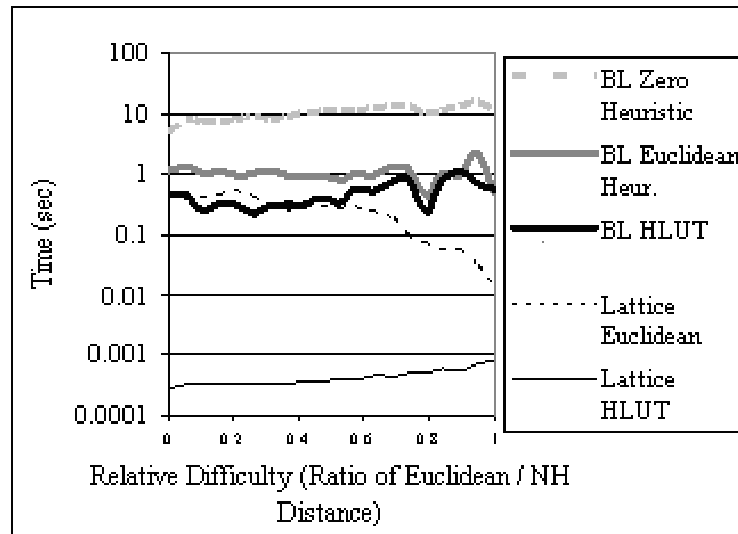

Figure 20: Lattice vs. BL without Obstacles. Queries with an absolute difficulty of 40 cells are shown. Various heuristics are considered for each planner. Only when the slow Zero heuristic is used does BL return optimal paths.

edges.

6.3 Heuristics

The lattice has been demonstrated to have some benefits over other control sets, but those experiments relied on an HLUT with approximately 2.3 million entries consuming over 28MB. This table was generated by first using best-first search to a distance of 100 cells, followed by the brushfire method with a trim ratio of 0.9. Even though this amount of memory is not significant for today's systems, it would be desirable to devote fewer resources to the heuristic function if possible. Therefore, the impact on the planner of using smaller HLUTs was examined.

There is a basic trade-off involving HLUT size, which is defined according to its backup heuristic (Euclidean distance). A trim ratio of 0.0 is tantamount to a simple Euclidean distance heuristic function, which we have seen to perform poorly on many lattice queries due to the increased number of A* expansion steps. Conversely, if the HLUT is very large, the minimal number of expansions is performed at the expense of increased demands for memory to store the HLUT itself. The ideal HLUT is large enough to efficiently solve queries without being so large that gains in memory saved in exploration are consumed by the HLUT itself.

The effect of trim level on average processor requirement can be observed by aggregating all queries for a given trim level and examining the effect of varying the HLUT size on CPU consumption. In Figure 24, the effect of trim level on computation

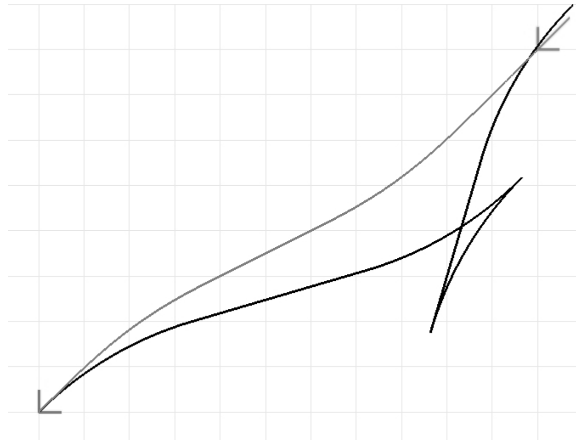

Figure 21: BL sub-optimality example. In black, the BL planner with Euclidean heuristic is used. In grey, the standard lattice control set with Euclidean heuristic returns an optimal path. BL does not because it does not permit revisiting of states.

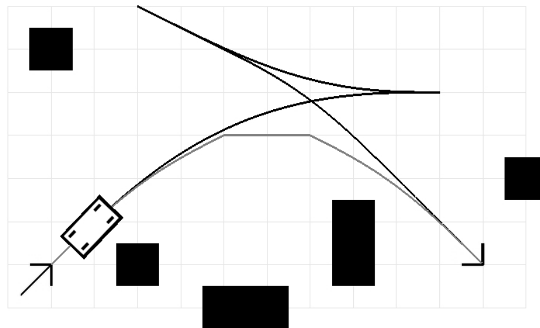

Figure 22: Lattice Control Set with and without the Ability to Turn in Place. In black, a control set with a minimum turning radius of 8 cells is used. In grey, the same control set is augmented by two additional controls which allow the vehicle to turn in place at a cost equal to traversing 5 cells. This plan performs the turn-in-place maneuver twice in order to execute a sharp turn.

time is shown. There is a clear minimum in the curves both with and without obstacles, which occurs at trim level 0.8, corresponding to an HLUT of approximately 2.5MB.

In Figure 25, the same data are examined from the perspective of memory usage. Here it is apparent that there is a trade-off between the amounts of memory consumed by the HLUT and as a result of the graph search itself. Once again, the optimal size is a trim level of 0.8.

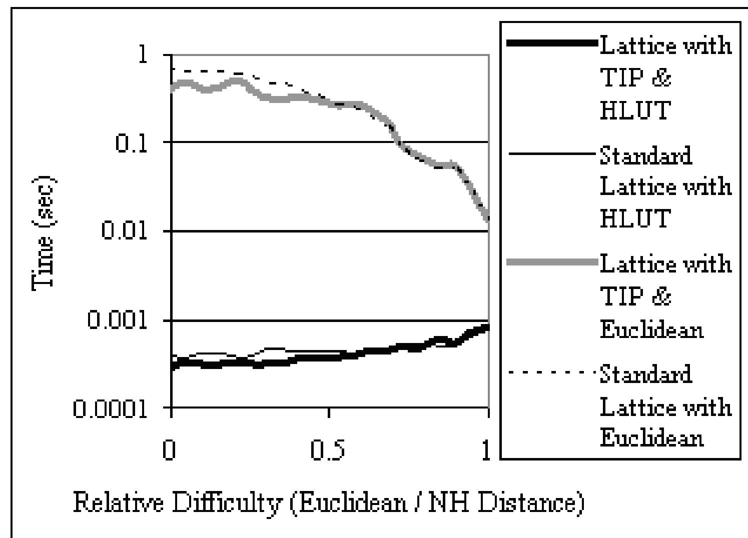

Figure 23: Lattice vs. Enhanced Lattice without Obstacles. Queries with an absolute difficulty of 40 cells are shown. Performance for the two lattices is comparable.

When using this new look-up table as a heuristic rather than the original one (that was ten times larger), the reduction in performance of the planner was shown to be quite insignificant, since only large trim ratio queries were removed. But the size of the HLUT can be easily tuned as desired for any application, simply by selecting the desired trim ratio.

7 Applications

Here we discuss several important mobile robotics applications that could benefit from the presented approach to constrained motion planning.

7.1 Off-road Navigation

The initial motivation for this work came from experiences with car-like mobile robots operating in cluttered natural terrain. Unstructured environments often pose significant challenges to mobile robots. The perception problem is known to be difficult; however, even if detailed, timely, and accurate perception information is available, it is still a difficult problem to fully utilize it in motion planning, especially under real-time constraints. Today's navigation algorithms often make mistakes that result in the vehicle being trapped in a set of dense obstacles with no way to proceed in the forward direction.

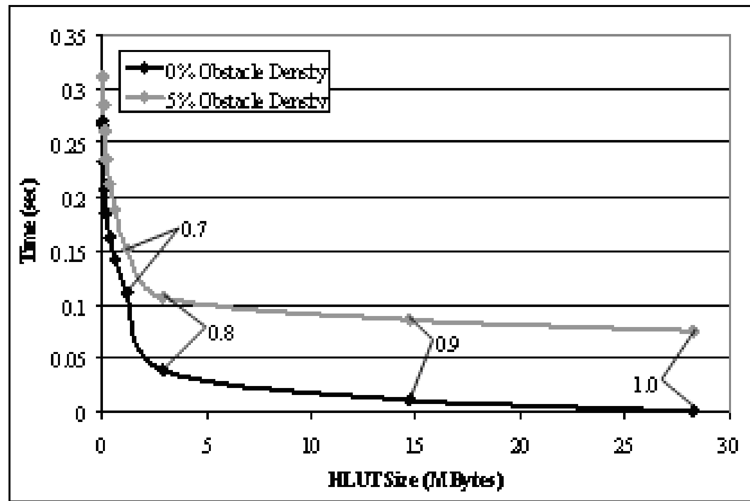

Figure 24: Time Comparison of HLUT Sizes. Upper trim levels are annotated. A* search time varies inversely with HLUT size, but there is a clear knee to the curve, with limited gains at trim levels above 0.8.

During the DARPA Perception for Off-Road Robotics (PerceptOR) project, our vehicle faced such problems several times an hour in challenging terrains. The initial ideas of the approach presented herein were investigated first on this program. An implementation was developed using most of the central concepts of this method: an explicit state lattice based on the mobility model of the PerceptOR vehicles (Figure 26) was precomputed, and the A* algorithm was utilized to search it [31]. The algorithm was invoked as an aggressive limited scope behavior when simpler alternatives failed to produce a solution.

Note that the final formulation of the method presented here achieves about two orders of magnitude speed up relative to the one utilized on PerceptOR. Custom, obstacle avoiding n-point turn maneuvers can now be generated in reactive fashion. Moreover, the new implicit representation of the lattice reduces memory requirements significantly. The extended horizon of the planner would enable the generation of a backup maneuver all the way out of a discovered cul-de-sac, if this maneuver was considered preferable to a complicated n point turn.

7.2 Planetary Rover Locomotion

Motion planning for planetary exploration rovers is in many ways similar to the terrestrial problem discussed above. However, it is also has many distinctive features. One significant difference is that rovers typically move fairly slowly, only several centimeters per second, for a variety of reasons including stringent weight, computational and power limitations. Moreover, the rovers are typically designed to move in stop and go

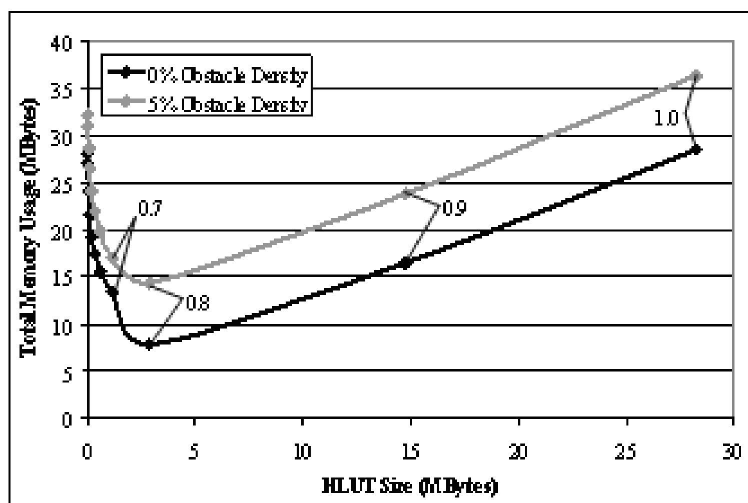

Figure 25: Memory Comparison of HLUT Sizes. Upper trim levels are annotated. Memory consumption varies with HLUT size, but a minimum occurs at trim level 0.8.

Figure 26: PerceptOR robot. A preliminary version of this algorithm was verified on PerceptOR vehicles, designed to navigate in natural environments.

fashion. GESTALT, the rover navigation algorithm that drives the MER rovers uses higher level perception, such as stereo vision, while it is stationary and selects the best arc path to follow. Afterwards, the rover commits to driving the selected arc without perception, using only lower level sensors such as current sensors [24].

Despite these differences, the lattice planner can be applied to the rover locomotion

problem in a straight-forward manner. One way to apply it is similar to the terrestrial case, i.e. as a backup behavior that is invoked only when the rover is no longer able to navigate further due to errors in perception, path following, state estimation, etc. In this case the rover would compute a rescuing path and get back on track autonomously. Future adoption of technologies such as this will undoubtedly reduce the effort and cost of manual rover path planning.

The presented planner may also be applied in other ways. For example, given its efficiency, it's reasonable to imagine that navigation algorithms such as GESTALT could be enhanced with lattice planning capability, such that instead of arc selection, they might perform lattice search and produce the complete path: not only an arc, but an "S" shape or anything else as required by the environment. Given the efficiency of the planner, the computation required is promising to be comparable to the conventional method of evaluating the requisite number of arcs and running an arbiter.

Figure 27: ROAMS simulation of the planner in action. A CLARAty implementation of the state lattice planner is controlling the ROAMS simulation of the Rocky8 rover in this screenshot. The planner generated a smooth path to go around the large rock next to rover in order to drive the rover to explore a crevasse between the two rocks at the left edge of figure.

As was discussed above, in simple to medium obstacle fields, the lattice planner compares in efficiency to grid search, primarily because the heuristic we designed accurately guides the search such that backtracking is minimized. The search can be visualized as proceeding in depth-first manner, but leading invariably towards the goal, thanks to the ideal heuristic encoded in the look-up table. In many implementations, obstacle avoidance arc evaluation is performed like breadth-first search to unity depth: many arcs are evaluated, but only is ultimately used. With careful design, the presented planner could save computation and extend the planning horizon to the limits of perception.

An implementation of the state lattice planner has been integrated with the Coupled Layer Architecture for Robotic Autonomy (CLARAty) system at the Jet Propul-

sion Laboratory. This allowed us to test the application of the planner to rover locomotion through simulation using JPL's high-fidelity rover dynamics simulation system, ROAMS (Rover Modeling and Simulation). Figure 27 depicts a simulation of a Rocky8 research prototype rover performing the instrument placement task. The goal is to move toward the goal in a field spotted with large obstacles. Thanks to the general nature of CLARAty software architecture, the same code that drives the ROAMS simulation can also control the actual rover. Our plans for the near future include validating this algorithm on the actual rover vehicle at JPL's Mars Yard. An important motivation for this work is to assist in the single-cycle instrument placement task. Currently it takes three to four Martian days for rover drivers to move the rover to the desired science target. By using this planner, we believe the instrument placement task will be simplified by making the navigation efficient computationally, by fully utilizing high-level perception information by computing as long "blind drive" paths as perception allows, and by guaranteeing that the rover's navigator never gets stuck, as unlimited search will always find the path out of cul-de-sacs, whereas typical arc-based navigators typically are not designed for this purpose.

7.3 Reactive Motion Planning and Path Modification

By virtue of its efficiency, the presented planner can be useful in a variety of other tasks. In particular, it could be applied to automatic mobile equipment that performs routine and repetitive tasks, e.g. earth movers, harbor transporters, convoy vehicles, etc. It is worth noting that such systems can benefit from more structured environments than unknown natural terrain. It is possible to build a reactive motion planning system that will be able to modify its current path immediately upon perceiving an unexpected obstacle. Moreover, as we have shown in the discussion of admissibility of the heuristic, the new path will be optimal, which is likely to result in minimum deviation from the original course, just enough to negotiate the unexpected obstacle.

8 Summary

This work has proposed a novel approach to motion planning under differential constraints and a generative formalism for the construction of discrete control sets for differentially constrained motion planning. The inherent encoding of constraints in the resulting representation re-renders the problem of motion planning in terms of unconstrained heuristic search. The encoding of constraints is an offline process that does not affect the efficiency of on-line motion planning.

We have begun by carefully outlining the assumptions and the process of formulating constrained planning in terms of the search in a specialized search space. The policy of including curvature and possibly velocity in this space in order to guarantee their continuity was explained. We then discussed the requirements for the proposed search space: regularity, symmetry and feasibility, such that only feasible motions are included in this representation. Further, the concept of path equivalence classes was derived from the notion of state discretization.

From this discussion, it was a natural progression to define extended neighborhoods of states that are reachable by the system through the primitive motions, and it was noted that due to the connectivity of the state lattice, some of the primitives will be redundant. By combining this observation with our earlier notion of path equivalence we proceeded to introduce the process of path decomposition, whereby the redundant motions are efficiently culled from the state lattice.

Once the search space is clearly defined, we described how a basic unconstrained heuristic search algorithm can be used to search the state lattice very efficiently. This efficiency is due in large part to the heuristic used by the search. We were able to get appreciable performance gains by constructing a heuristic that was well-suited for efficient search, a Heuristic Look-Up Table (HLUT). Through a careful cost benefit analysis, we were able to extract maximum benefit from this structure while using an amount of memory that is relatively insignificant to modern motion planning systems. We further presented a comparison to other popular motion planners. Lastly, some notions for applying this technology in today's mobile robotics systems was also presented.

9 Conclusion and Future Work

This work has been motivated by a fairly acute need to endow our field robots with sufficient understanding of their own mobility to allow them to efficiently plan correct and intricate paths, consisting of many primitive motions, in response to their challenging surroundings.

Our approach inherits its understanding of mobility from a competent and high fidelity real-time trajectory generator. It uses this module to construct what amounts to an ideal discrete search space because:

- It is of high enough dimension to enforce state continuity.
- Its controls acquire goal states exactly.
- Its controls satisfy arbitrary differential constraints, so they encode only feasible motions.
- Its controls are minimal in a clearly defined path sampling sense, meaning unnecessary computation is avoided.

Furthermore, we search this ideal space using an ideal heuristic, constructed by the planner itself, for a world free of obstacles. With all of this in place, the daunting problem of optimal smooth nonholonomic motion planning in the presence of obstacles is reduced to an implementation of heuristic search of a graph.

The planner is resolution complete because the control set can be automatically adjusted to generate new controls as resolution increases, and the space is searched systematically. The planner is also optimal because precision controls permit the redirection of back-pointers which is fundamental to optimal heuristic search. The completeness and the optimality of the planner result from the precision of the trajectory generator.

The planner is also efficient due to a principled implementation of a path sampling policy. We have shown that it is up to two orders of magnitude faster than the BL planner while remaining optimal. It is not as fast as a grid planner when obstacles are present. It is roughly an order of magnitude slower than a 16 connected grid. However, problems with grid planners producing infeasible plans motivated the work to start with. According to these results, a mere 3 years of operation of Moore's law should have been enough to allow grids to be abandoned long ago, at least for near field planning.

The planner is also capable of planning motions over rough terrain, if the terrain is known at planning time, or of adjusting its controls to preserve the topology of the primitive motion sequence as the terrain becomes known during execution.

In fact, the contribution of this work is not a planning algorithm. The contribution is a principled mechanism to construct an efficient, precision, differentially constrained search space upon which any planner may operate. We have also presented a compelling case for why it is superior to prevailing alternatives.

Future work includes the evaluation of the search space in both applications mentioned earlier in addition to introducing real-time replanning by replacing the A* algorithm with D*. We also intend to develop a search space whose fidelity decreases with distance from the present position in order to scale our results to be applicable to kilometers of traverse.

References

[1] Agarwal, Aronov, and Sharir. Motion planning for a convex polygon in a polygonal environment. *GEOMETRY: Discrete and Computational Geometry*, 22, 1999.

[2] P. Agarwal, N. Amenta, B. Aronov, and M. Sharir. Largest placements and motion planning of a convex polygon. In *Proc. 2nd Annu. Workshop Algorithmic Foundations of Robotics*, Toulouse, France, 1996.

[3] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Naher, S. Schirra, and C. Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. In *Proc. of the ACM Symposium on Computational Geometry*, pages 281–289, 1990.

[4] D. A. Anisi, J. Hamberg, and X. Hu. Nearly time-optimal paths for a ground vehicle. *Journal of Control Theory and Applications*, 2003.

[5] C. Baker, A. Morris, D. Ferguson, S. Thayer, C. Whittaker, Z. Omohundro, C. Reverte, W. Whittaker, D. Hahnel, and S. Thrun. A campaign in autonomous mine mapping. In *Proc. of the IEEE Conference on Robotics and Automation*, 2004.

[6] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for robot path planning. In G. Giralt and G. Hirzinger, editors, *Proc. of the 7th International Symposium on Robotics Research*, pages 249–264. Springer, New York, NY., 1996.

[7] J. Barraquand and J.-C. Latombe. On nonholonomic mobile robots and optimal maneuvering. In *Proc. of the IEEE International Symposium on Intelligent Control*, 1989.

[8] J. Barraquand and J.-C. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. *Proc. of the IEEE International Conference on Robotics and Automation*, pages 1712–1717, 1990.

[9] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: controllability and motion planning in the presence of obstacles. *Proc. of the IEEE International Conference on Robotics and Automation*, 1991.

[10] A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. *Hybrid Systems: Computation and Control (Lecture Notes in Computer Science, no. 2993)*, pages 67–78, 2004.

[11] A. Bicchi, A. Marigo, and B. Piccoli. On the reachability of quantized control systems. *IEEE Transactions on Automatic Control*, 47(4):546–563, 2002.

[12] R. Bohlin. Path planning in practice; lazy evaluation on a multi-resolution grid. *Proc. of the IEEE/RSJ International Conference on Intelligent Robots & Systems*, 2001.

[13] R. Bohlin and L. Kavraki. Path planning using lazy PRM. *Proc. of the IEEE International Conference on Robotics & Automation*, 2000.

[14] M.S. Branicky, S.M. LaValle, S. Olson, and L. Yang. Quasi-randomized path planning. *Proc. of the International Conference on Robotics and Automation*, 2001.

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

[16] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.

[17] A. Casal. *Reconfiguration planning for modular self-reconfigurable robots*. PhD thesis, Aeronautics and Astronautics Dept., Stanford U., 2001.

[18] M. Cherif. Kinodynamic motion planning for all-terrain wheeled vehicles. In *Proc. of the IEEE International Conference on Robotics & Automation*, 1999.

[19] 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.

[20] C. Fernandes, L. Gurvits, and Z. X. Li. A variational approach to optimal nonholonomic moton planning. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 680–685, 1991.

[21] T. Fraichard and J.-M. Ahuactzin. Smooth path planning for cars. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3722–3727, 2001.

[22] T. Fraichard and A. Scheuer. From Reeds and Shepp's to continuous-curvature paths. *IEEE Transactions on Robotics*, 20(6):1025–1035, 2004.

[23] E. Frazzoli, M.A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. In *Proc. of the American Control Conference*, 2001.

[24] S. Goldberg, M. Maimone, and L. Matthies. Stereo vision and rover navigation software for planetary exploration. In *Proc. of IEEE Aerospace Conference*, 2002.

[25] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM SIGGRAPH*, pages 171–180, 1996.

[26] T. Howard and A. Kelly. Trajectory generation on rough terrain considering actuator dynamics. In *Proc. of the 5th International Conference on Field and Service Robotics*, 2005.

[27] D. Hsu. *Randomized single-query motion planning in expansive spaces*. PhD thesis, Computer Science Dept., Stanford University, 2000.

[28] F. Jean. Complexity of nonholonomic motion planning. *International Journal of Control*, 74(8):776–782, 2001.

[29] L.E. Kavraki. *Random networks in configuration space for fast path planning*. PhD thesis, Computer Science Dept., Stanford University, 1994.

[30] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics & Automation*, 12(4):566–580, 1996.

[31] A. Kelly, O. Amidi, M. Happold, H. Herman, T. Pilarski, P. Rander, A. Stentz, N. Vallidis, and R. Warner. Toward reliable off-road autonomous vehicle operating in challenging environments. In *Proc. of the International Symposium on Experimental Robotics*, 2004.

[32] A. Kelly and B. Nagy. Reactive nonholonomic trajectory generation via parametric optimal control. *International Journal of Robotics Research*, 22(7/8):583–601, 2002.

[33] R. Kindel. *Motion planning for free-flying robots in dynamic and uncertain environments*. PhD thesis, Aeronaut. & Astr. Dept., Stanford University, 2001.

[34] J.J. Kuffner. *Autonomous agents for real-time animation*. PhD thesis, Computer Science Dept., Stanford University, 1999.

[35] A. Lacaze, Y. Moscovitz, N. DeClaris, and K. Murphy. Path planning for autonomous vehicles driving over rough terrain. In *Proceedings of the IEEE International Symposium on Intelligent Control*, 1998.

[36] F. Lamiraux and J.-P. Laumond. Smooth motion planning for car-like vehicles. *IEEE Transactions on Robotics and Automation*, 2001.

[37] J.-C. Latombe. *Robot Motion Planning*. Kluwer, Boston, 1991.

[38] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[39] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 473–479, 1999.

[40] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.

[41] S.M. LaValle, M. Branicky, and S. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research*, 23(7/8):673–692, 2004.

[42] S.R. Lindemann and S.M. LaValle. Current issues in sampling-based motion planning. In *Proc. of the International Symposium of Robotics Research*, 2003.

[43] S.R. Lindemann and S.M. LaValle. Steps toward derandomizing RRTs. In *Proc. of the Fourth International Workshop on Robot Motion and Control*, 2004.

[44] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983.

[45] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[46] A. Morris, D. Silver, D. Ferguson, and S. Thayer. Towards topological exploration of abandoned mines. In *Proc. of the IEEE International Conference on Robotics*, 2005.

[47] B. K. Natarajan. The complexity of fine motion planning. *International Journal of Robotics Research*, 7(2):36–42, 1988.

[48] S. Pancanti, L. Pallottino, and A. Bicchi. Motion planning through symbols and lattices. In *Proc. of the Int. Conf. on Robotics and Automation*, 2004.

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

[50] J. Reif. Complexity of the mover's problem and generalizations. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[51] G. Sanchez and J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proc. of International Symposium on Robotics Research*, 2001.

[52] G. Sanchez and J.-C. Latombe. On delaying collision checking in prm planning: Application to multi-robot coordination. *International Journal of Robotics Research*, 21(1):5–26, 2002.

[53] A. Scheuer and T. Fraichard. Collision-free and continuous-curvature path planning for car-like robots. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 867–873, 1997.

[54] A. Scheuer and Ch. Laugier. Planning sub-optimal and continuous-curvature paths for car-like robots. In *Proc. of the International Conference on Robotics and Automation*, 1998.