Efficient Search and Hierarchical Motion Planning by Dynamically Maintaining Single-Source Shortest Paths Trees

Michael Barbehenn, *Student Member, IEEE*, and Seth Hutchinson, *Member, IEEE*,

Abstract— **Hierarchical approximate cell decomposition is a popular approach to the geometric robot motion planning problem. Planners that use this approach iteratively refine an approximate representation of the robot configuration space, searching each refined representation for a solution path, until a solution path is found. In many cases, the search effort expended at a particular iteration can be greatly reduced by exploiting the work done during previous iterations. In this paper, we describe how this exploitation of past computation can be effected by the use of a dynamically maintained single-source shortest paths tree.**

Our approach is as follows. We embed a single-source shortest paths tree in the connectivity graph of the approximate representation of the robot configuration space. This shortest paths tree records the most promising path to each vertex in the connectivity graph from the vertex corresponding to the robot's initial configuration. At each iteration, some vertex in the connectivity graph is replaced with a new set of vertices, corresponding to a more detailed representation of the configuration space. Our new, dynamic algorithm is then used to update the single-source shortest paths tree to reflect these changes to the underlying connectivity graph. Thus, at each iteration of the planning algorithm, a representation of the best path from the initial configuration to the goal configuration is computed by a set of local tree update operations. Our planner is fully implemented and we give empirical results to illustrate the performance improvements of the dynamic algorithms.

Index Terms—**Hierarchical Robot Motion Planning, Dynamic Single-source Shortest Paths**

I. Introduction

THE ultimate goal of robotics research is the creation of autonomous agents, capable of planning and executing tasks that are specified in terms of high level goals. One of the many requirements for such an agent is the ability to plan collision free paths through an environment populated with obstacles. In its most restricted form, this path planning problem considers only the geometry of the robot and the obstacles in the environment (it ignores, for example, constraints on dynamic response). This restricted version of the path planning problem is known in the robotics literature as the geometric motion planning problem, the Mover's problem, or the FindPath problem.

There have been many approaches to the geometric motion planning problem, including exact methods ([15], [45], [7]), artificial potential fields-based methods ([34], [43], [32], [11], [47]), and approximate methods ([13], [14], [23], [24], [31], [33], [49]). In this paper, we present a new approach to the geometric robot motion planning problem using hierarchical approximate cell decomposition.

Hierarchical approximate cell decomposition has several advantages over competing methods. First, a complete, explicit description of the configuration space is not computed *a priori*. Instead, a representation of the configuration space is constructed incrementally, and only to the resolution necessary to solve the current problem. Furthermore, approximate methods are complete to an arbitrary resolution. Secondly, hierarchical approximate cell decomposition does not suffer from the problems associated with the artificial potential fields method (the most well known problem being that of local minima in the potential field).

A number of issues must be addressed when designing a hierarchical approximate cell decomposition motion planner, including the representation of configuration space, cell subdivision, cell labeling, and search. The first three of these have been well-treated in many previous papers: representation in [39], [13], [14], [49], subdivision in [13], [14], [23], [24], [49], and labeling in [13], [14], [49]. In this paper we use the representation and labeling algorithm given in [13], [14], and octrees for cell-subdivision.

Search, in the context of hierarchical motion planning, has received less thorough attention, in spite of the fact that the dominant computational cost at each iteration of the planner is generally the time spent constructing a path from the initial to the goal configuration. To date, two basic search strategies for finding a path at each iteration of the planning algorithm have been reported: A* search [13], [14], and a *bridge the gap* strategy [13], [14], [33], [49] (these will be discussed in more detail in Section III-B). For each of these search methods, the planner often performs much redundant computation at successive iterations of the planning algorithm. Furthermore, the bridge the gap strategy may produce long and convoluted paths.

In this paper, we present a new search method that eliminates the redundancy between searches at successive iterations

Manuscript received April 26, 1993; revised June 23, 1994. This work was supported by grant NSF-IRI-9110270 from the National Science Foundation.

Michael Barbehenn is with the Department of Computer Science and the Artificial Intelligence Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign Urbana, IL 61801 USA.

Seth Hutchinson is with the Department of Electrical and Computer Engineering and the Artificial Intelligence Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign Urbana, IL 61801 USA.

IEEE Log Number 9408223.

of the planning algorithm. Our search method is founded on the ability to efficiently maintain a single-source shortest paths tree embedded in the connectivity graph, subject to the dynamic modifications that result from incremental subdivision of cells. Thus, at each iteration of the planning algorithm, a representation of the best path from the initial configuration to the goal configuration is computed by a set of local tree update operations. At each iteration, the path that is determined by our algorithm is the same as would be found by A* search, but the redundant computation that would be incurred by using A* at each iteration is avoided.

The computation of shortest paths trees is a fundamental, and well studied, problem in computer science; however, the problem of dynamically updating a shortest paths tree when the underlying graph undergoes some change (*e.g.* nodes are added to, or deleted from, the underlying graph) has only recently emerged as a topic of active research. Progress in this area could directly benefit many research areas, including communication networks, VLSI design, transportation networks, and scheduling in manufacturing shops. Thus, our results, although applied specifically to the problem of geometric robot motion planning, have potential application to a wide variety of problems.

The remainder of the paper is organized as follows. In Section II we present a formal specification of the robot motion planning problem, in the context of hierarchical approximate cell decomposition. In Section III we give a precise characterization of the search performed by hierarchical approximate cell decomposition planners. In Section IV we describe how changing the connectivity graph for the configuration space (by subdividing some cell in the representation of the configuration space) affects a single-source shortest paths tree that is used to maintain the best path to each vertex in the graph from the vertex that corresponds to the initial configuration. In Section V we present two new dynamic algorithms for the construction of a single-source shortest paths tree. Then, in Section VI we present results, and compare our new planner to those reported in [13], [14], [49]. In Section VII, we discuss how our solution overcomes previous criticisms of hierarchical approximate cell decomposition methods, and future work aimed at further reducing planning time. Finally, in Section VIII we present some conclusions about our work.

II. Problem Formulation

For a polygonal robot moving among polygonal obstacles in the plane, the motion planning problem can be expressed as follows. The configuration space of the robot is $\mathcal{C} = \mathbf{R}^2 \times S^1$. All configurations $q \in \mathcal{C}$ for which the robot is in contact with (or intersects) some obstacle belong to the set of configuration space obstacles (abbreviated C-obstacles), denoted by $\mathcal{CB}$. For all other configurations, the robot is in free space, denoted by $\mathcal{C}_{\text{FREE}}$. A planning problem is specified by an initial and a goal configuration, q_{init} and q_{goal}, respectively. A solution trajectory is a continuous mapping $\tau : [0,1] \rightarrow \mathcal{C}_{\text{FREE}}$, such that $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$.

We use a hierarchical subdivision algorithm to partition $\mathcal{C}$ into rectangloid cells, $\kappa_i = [x_i', x_i''] \times [y_i', y_i''] \times [\theta_i', \theta_i'']$ with $\theta = 0$ procedurally identified with $\theta = 2\pi$. We denote a partition of $\mathcal{C}$ as $\mathcal{P}$, and the subdivision of a cell κ as $\mathcal{P}^{\kappa}$ following [36]. Each cell is labeled EMPTY, FULL, or MIXED depending on whether it is completely contained in $\mathcal{C}_{\text{FREE}}$, completely contained in $\mathcal{CB}$, or not known to be completely contained in either $\mathcal{C}_{\text{FREE}}$ or $\mathcal{CB}$, respectively. The subdivision algorithm is approximate because MIXED cells below some user-specified resolution are considered to be FULL [13], [14] [23], [24], [33], [29], [49], [36].

```
procedure FindPath (q_init, q_goal:configuration; G_0:graph)
[1]   i ← 0
[2]   π ← an M-path from v_init to v_goal in G_i
[3]   until π is an E-path or π is NIL do
[4]       select vertices {v_i} ⊆ π
[5]       subdivide cells {κ_i} and construct G_{i+1}
[6]       i ← i + 1
[7]       π ← an M-path from v_init to v_goal in G_i
[8]   return π
end
```

Fig. 1. The traditional FindPath algorithm.

A sequence of adjacent cells, $\kappa_1 \kappa_2 \ldots \kappa_n$, is called a channel. A channel composed of EMPTY and MIXED cells is termed an M-channel, and an M-channel that has only EMPTY cells is an E-channel. Let κ_{init} and κ_{goal} denote those cells that contain q_{init} and q_{goal}, respectively. Then the planning process consists of subdividing MIXED cells until an E-channel, $\kappa_{init} \ldots \kappa_{goal}$ is found. Subdivision of cell κ in $\mathcal{P}_i$ produces $\mathcal{P}_{i+1}$ as follows: $\mathcal{P}_{i+1} = (\mathcal{P}_i - \{\kappa\}) \cup \mathcal{P}^{\kappa}$. A solution trajectory can be obtained from this E-channel subject to various criteria that we do not discuss here [13], [14], [23], [24], [33], [29], [36].

To facilitate efficient search for an E-channel, we maintain the connectivity graph $\mathcal{G}(V, E)$ of $\mathcal{P}$. Each vertex $v \in V$ has an associated non-FULL cell, κ. An edge $(v_i, v_j) \in E$ if and only if $\kappa_i \cap \kappa_j$ is a two dimensional boundary area. The connectivity relation is nonreflexive and symmetric. Associated with each path in $\mathcal{G}$ is a channel in $\mathcal{P}$. We refer to a path that corresponds to an M-channel (respectively, E-channel) as an M-path (respectively, E-path). Let v_{init} and v_{goal} be the vertices associated with κ_{init} and κ_{goal}, respectively.

Our planner initially subdivides κ_{init} and κ_{goal} until they are EMPTY, to ensure that q_{init} and q_{goal} can be represented within the given resolution. (This is also done in [13], [14].) As an aside, this means that κ_{init} and κ_{goal} will never be subdivided in the course of planning.

The traditional FindPath algorithm is shown in Fig. 1. The description of this algorithm is not unique, of course. Variations can be found in [36], [13], [14] for example. We chose this instance of the FindPath algorithm to emphasize the distinct implementation choices of Lines 4 and 7, and the ramifications of the selection of an M-path in Line 7 on the termination of the FindPath algorithm in Line 3.

In this paper we show how the performance of the traditional FindPath algorithm can be significantly improved through the use of a single-source shortest paths tree ($\mathcal{SP}$) to maintain potential solution paths at each iteration of the algorithm. Specifically, by dynamically maintaining $\mathcal{SP}$, we reduce the

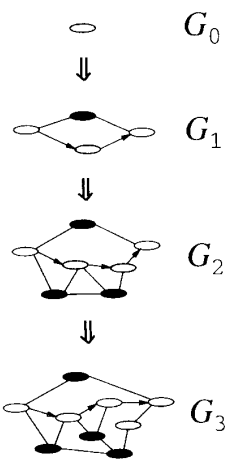


Fig. 2. Plan search space.

procedure FindPath (q_{init}, q_{goal}:configuration; $\mathcal{G}_0$:graph)
[1] $i \leftarrow 0$
[2] $\pi \leftarrow$ the least cost path from v_0 to v_{goal} in $\mathcal{SP}_i$
[3] **until** π is an E-path **or** π is NIL **do**
[4] $v_s \leftarrow v \in \pi$ such that cost(v) is maximum
[5] subdivide MIXED cell κ_s and construct $\mathcal{G}_{i+1}$
[6] propagate changes to construct $\mathcal{SP}_{i+1}$
[7] $i \leftarrow i + 1$
[8] $\pi \leftarrow$ the least cost path from v_0 to v_{goal} in $\mathcal{SP}_i$
[9] **return** π
end

Fig. 3. Our improved FindPath algorithm.

search for an M-path in Line 7 to a local tree update. In contrast, traditional implementations use some type of search at each iteration. Fig. 3 depicts our improved algorithm.

III. FINDPATH SEARCH

The FindPath algorithm shown in Fig. 1 searches for a graph $\mathcal{G}_n$ that contains an E-path from v_{init} to v_{goal}. The search for $\mathcal{G}_n$ begins with the initial connectivity graph $\mathcal{G}_0$ and progresses by subdividing some MIXED cells in the underlying partition $\mathcal{P}_0$ to obtain $\mathcal{G}_1$ (thereby producing a more detailed representation of the underlying configuration space, $\mathcal{C}$), and so on. In order to more clearly understand this search process, we can decompose the search into two components: the search for a solution graph $\mathcal{G}_n$, and the search within $\mathcal{G}_i$ for an E-path. An example of a possible execution of FindPath is illustrated in Fig. 2. In the figure, the search for a solution graph $\mathcal{G}_n$ is represented by a vertical sequence of graphs from the initial graph $\mathcal{G}_0$ to a solution graph $\mathcal{G}_3$. Within each $\mathcal{G}_i$, the search for an E-path produces either an M-path or an E-path. In the figure, these paths are indicated by arrows, and shading is used to represent vertex cost (our vertex cost function is described in Section VI-A).

In the remainder of this section, we will develop a mathematical characterization of the search process, use this characterization to describe previous hierarchical approaches to motion planning, and give a brief overview of our new approach.

A. The Lattice of Connectivity Graphs

We assume that $\mathcal{C}$ is bounded, and is enclosed in a cell κ_0. A given deterministic subdivision algorithm uniquely partitions κ_0 into subcells. For example, using octrees κ_0 would be subdivided into eight cells. The following definitions provide a mathematical characterization of the space of all possible decompositions of $\mathcal{C}$.

Definition 1: Let $\mathcal{AP}$ be the space of all partitions that can be obtained by performing a sequence of subdivisions on cells in the representation of $\mathcal{C}$.

Definition 2: Given two partitions, $\mathcal{P}_1$ and $\mathcal{P}_2$ in $\mathcal{AP}$, $\mathcal{P}_1 \preceq \mathcal{P}_2$ if and only if for all $\kappa_1 \in \mathcal{P}_1$ there exists some $\kappa_2 \in \mathcal{P}_2$ such that $\kappa_1 \subseteq \kappa_2$.

The $\preceq$ relation imposes a partial order on $\mathcal{AP}$. Every pair of partitions has a greatest lower bound and a least upper bound in $\mathcal{AP}$. Therefore $\mathcal{AP}$ is a lattice. For a given minimum resolution on the size of a cell, or if the subdivision algorithm is exact, the lattice is finite. The least upper bound of the lattice is the initial partition $\mathcal{P}_0$ that contains only the cell κ_0, the initial unsubdivided cell that encloses $\mathcal{C}$. The greatest lower bound of the lattice is the completely subdivided (to resolution) partition $\mathcal{P}_\infty$.

For every partition $\mathcal{P} \in \mathcal{AP}$, there is a unique corresponding connectivity graph $\mathcal{G}$, which leads to the following definition.

Definition 3: For $\mathcal{AP}$, the space of partitions for some configuration space representation, denote by $\mathcal{AG}$ the corresponding space of connectivity graphs.

The bijective map between $\mathcal{AP}$ and $\mathcal{AG}$ implies that $\mathcal{AG}$ is also a lattice. (The mapping is a bijection because associated with each vertex in the connectivity graph is a spatially unique cell of the partition. Thus while two connectivity graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, with corresponding partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, may be topologically isomorphic, $\mathcal{G}_1 \neq \mathcal{G}_2$ since the vertices in the two graphs have unique, spatially dependent labels.) So, for every connectivity graph $\mathcal{G} \in \mathcal{AG}$, there is a corresponding underlying partition $\mathcal{P} \in \mathcal{AP}$. Let $\mathcal{G}_{i+1} \preceq \mathcal{G}_i$ if and only if $\mathcal{P}_{i+1} \preceq \mathcal{P}_i$.

We now characterize the set of solution graphs that exist within $\mathcal{AG}$.

Definition 4: A *solution graph* is any graph $\mathcal{G} \in \mathcal{AG}$ that contains an E-path from v_{init} to v_{goal}. Denote by $\mathcal{SG}$ the set of all solution graphs.

A sublattice is a subset of a lattice that is itself a lattice. If $\mathcal{SG}$ were a sublattice, a natural goal for the FindPath algorithm would be to find the least upper bound of $\mathcal{SG}$ as the solution graph. In general, the set $\mathcal{SG}$ does not form a sublattice of $\mathcal{AG}$ because there may be multiple incomparable solution graphs such that their least upper bound is not a solution graph. However, once an E-path exists in $\mathcal{G}$, further subdivisions to MIXED cells in the underlying partition $\mathcal{P}$ do not affect the existence of that E-path. This has also been noted in [49], [36]. We express this more formally in the following theorem.

Theorem 1 [Solution]: Let $\mathcal{G} \in \mathcal{SG}$, and let $G = \{\mathcal{G}_i \in \mathcal{AG} \mid \mathcal{G}_i \preceq \mathcal{G}\}$ (*i.e.* G is the set of all connectivity graphs

that can be obtained from $\mathcal{G}$ by a sequence of subdivisions in $\mathcal{P}$). Then $G \subseteq \mathcal{SG}$.

Proof: It is given that $\mathcal{G} \in \mathcal{SG}$. Assume that after some number of subdivisions to cells in the corresponding partition $\mathcal{P}$ we have obtained $\mathcal{G}_i \in \mathcal{SG}$. Let κ be the cell that is subdivided in $\mathcal{P}_i$ to produce $\mathcal{P}_{i+1}$. If κ is not in the E-channel in $\mathcal{P}_i$, then the E-channel in $\mathcal{P}_i$ exists in $\mathcal{P}_{i+1}$ so $\mathcal{G}_{i+1} \in \mathcal{SG}$. If κ is an EMPTY cell in the E-channel in $\mathcal{P}_i$, then κ subdivides into adjacent EMPTY cells and there is a solution E-channel in $\mathcal{P}_{i+1}$, so $\mathcal{G}_{i+1} \in \mathcal{SG}$. By induction on the number of subdivisions to $\mathcal{P}$, every $\mathcal{G}_i$ that can be obtained by subdividing cells in $\mathcal{P}$ is a solution graph. So $G \subseteq \mathcal{SG}$.

Definition 5: A *solution predicate* is a function $f : \mathcal{AG} \rightarrow \{\text{TRUE}, \text{FALSE}\}$ that maps some of the solution graphs to TRUE and all other graphs to FALSE.

Definition 6: We say that a solution graph $\mathcal{G} \in \mathcal{SG}$ is *recognizable* by a solution predicate if the solution predicate maps $\mathcal{G}$ to TRUE. Let $\mathcal{RG} \subseteq \mathcal{SG}$ be the set of all recognizable solution graphs.

Definition 7: A solution predicate is *omniscient* if and only if $\mathcal{RG} = \mathcal{SG}$.

In this paper, we use the solution predicate "the least cost path $\langle v_{init} \ldots v_{goal} \rangle$ is an E-path." Whether this particular solution predicate is omniscient or not depends on the choice of the vertex cost function. If a particular connectivity graph, $\mathcal{G}$, contains an M-Path $\langle v_{init} \ldots v_{goal} \rangle$ with lower cost than any such E-path in $\mathcal{G}$, then our solution predicate will return FALSE for $\mathcal{G}$. In all cases, by Theorem 1, if a solution graph exists in $\mathcal{AG}$, our algorithm will eventually produce a connectivity graph, $\mathcal{G}$, such that our solution predicate will return TRUE for $\mathcal{G}$ (since $\mathcal{P}_\infty$ contains only EMPTY cells).

B. Searching a Connectivity Graph

To date, most hierarchical motion planners based on approximate cell decomposition have used hill-climbing to find a solution graph in $\mathcal{AG}$, terminating when the selected path in the graph is an E-path. Where the planners have differed has been in how a path is selected at each iteration (Line 7 of the algorithm of Fig. 1).

The first reported method to find a path in $\mathcal{G}_i$ was A* search. A* search returns the least cost M-path in $\mathcal{G}_i$. The next connectivity graph, $\mathcal{G}_{i+1}$, is then generated by subdividing MIXED cells from the corresponding M-channel. This approach was first used in the planner described in [13], [14], which we will refer to as BLP1.

The major drawback of BLP1, as noted by the authors, is the inefficiency due to repeated use of A* search to find the least cost M-path. This led to the introduction of the *bridge the gap* strategy, which was first used by the planner reported in [13], [14]. We will refer to this planner as BLP2. The bridge the gap strategy works as follows. A vertex v_s with a MIXED cell in the previous M-path is selected for subdivision. This vertex is removed from the graph; its corresponding cell is subdivided; and the new vertices with non-FULL subcells are added to the connectivity graph. A local search for a bridge is then performed to connect the predecessor of v_s to the successor of v_s in the new M-path. In conjunction with the bridge the gap strategy, the authors also recommend increasing the number of cells subdivided per iteration to help combat the inefficiency.

The hierarchical planners given in [33], [49] also use the bridge the gap strategy. We will refer to these planners as KD and ZL, respectively. Both KD and ZL, which is a generalization of KD, restrict the search for a bridge to the vertices corresponding to the subcells of the just-subdivided cell. If no bridge is found, KD and ZL backtrack over decisions made at previous iterations. In the event that the bridge construction fails, ZL uses annotations to prune the search space in an attempt to limit the amount of search for an M-path. Once it has been determined that a bridge cannot be constructed between two vertices, no attempt will be repeated in the course of searching for an M-path connecting v_{init} and v_{goal}. The annotations of ZL limit the amount of time spent generating paths, rather than preventing their occurrence. The use of annotations does prevent some repeated search effort; however, annotations do not eliminate the repeated search effort between global searches across iterations (comparing the search for an M-path in $\mathcal{G}_i$ with that in $\mathcal{G}_{i+1}$), nor do they completely eliminate repeated search effort when failures require backtracking within an iteration.

The main disadvantage to the bridge the gap strategy is that, in the worst case, the construction of the bridge can be as difficult as performing a global search for an M-path. Furthermore, the construction of a bridge can lead to convoluted paths; and in no case is it guaranteed to find the least cost path from v_{init} to v_{goal} in $\mathcal{G}_i$. These effects are worsened by subdividing multiple cells per iteration. To prevent excessively convoluted paths, BLP2 puts resource limits on the search for a bridge in the form of a bound on the depth of the search. If no bridge is found within the resource limits, a global search for a new path from v_{init} to v_{goal} is undertaken. KD and ZL do not address this issue.

In this paper, we introduce a mechanism that eliminates the redundancy exhibited when using A* search at each iteration of FindPath, without resorting to the bridge the gap strategy. Specifically, we make use of the single-source shortest paths tree ($\mathcal{SP}$) embedded in $\mathcal{G}$, wherein the least cost path from the source, v_0, to every vertex is maintained. For our planning application, v_0 is v_{init}, and the path we are interested in is the path from v_0 to v_{goal}.

The relationship between A* search and computing $\mathcal{SP}$ can be seen by comparing the sequence of vertices expanded by A* search (with heuristic function $h = 0$) with the sequence of deletions in Dijkstra's algorithm, which is described in Section V. Both algorithms begin at the initial vertex v_{init}. Both algorithms expand the search by first adding all vertices adjacent to the current vertex to a set of generated but not yet expanded vertices. Both then select the vertex in this set with the least cost path from v_{init} to expand next. The difference is that Dijkstra's algorithm finishes when all of the vertices have been expanded, while A* search terminates when v_{goal} has been obtained. To this extent, a heuristic cost function, $h > 0$, may help A* terminate earlier. In summary, A* search computes the least cost path from v_{init} to v_{goal} while Dijkstra's algorithm computes the least cost path from v_{init} to every vertex.

At each iteration of our improved FindPath algorithm, a MIXED cell is subdivided and $\mathcal{SP}_{i+1}$ is computed. Once $\mathcal{SP}_{i+1}$ is computed, the least cost path from v_0 to v_{goal} is immediately available without the need to search. It may, in some cases, be computationally less expensive to use A* search than to compute $\mathcal{SP}_{i+1}$ since A* search may not examine all of the vertices in the graph to obtain a solution, as explained above. However, we note that the changes to $\mathcal{G}_i$ are local. Therefore, if $\mathcal{SP}_i$ is dynamically maintained (*i.e.* $\mathcal{SP}_{i+1}$ is obtained by efficiently updating $\mathcal{SP}_i$ to reflect the new changes), this yields a superior FindPath algorithm, which is depicted in Fig. 3.

IV. INCREMENTAL CHANGES IN $\mathcal{SP}$

When a MIXED cell is subdivided, the connectivity graph, $\mathcal{G}$, is modified to reflect the existence of the resulting new cells. In this section, we introduce the theory that describes the corresponding changes to a single-source shortest paths tree, $\mathcal{SP}$, embedded in $\mathcal{G}$. Specifically, we describe the differences between $\mathcal{SP}_i$ and $\mathcal{SP}_{i+1}$ given that (a) $\mathcal{SP}_i$ is a single-source shortest paths tree embedded in $\mathcal{G}_i$, (b) $\mathcal{G}_{i+1}$ is the connectivity graph obtained by subdividing some MIXED cell in $\mathcal{G}_i$, and (c) $\mathcal{SP}_{i+1}$ is a single-source shortest paths tree embedded in $\mathcal{G}_{i+1}$.

In Section IV-A we introduce terminology, and a local decision criterion for determining whether a tree is a single-source shortest paths tree. In Section IV-B we identify the changes that can occur between $\mathcal{SP}_i$ and $\mathcal{SP}_{i+1}$ for the special case when $\mathcal{G}_{i+1}$ is obtained by changing the cost of a single vertex in $\mathcal{G}_i$. Then, in Section IV-C, we discuss the changes that can occur between $\mathcal{SP}_i$ and $\mathcal{SP}_{i+1}$ in the more general case where $\mathcal{G}_{i+1}$ is the connectivity graph that results from subdividing a MIXED cell that corresponds to a vertex of $\mathcal{G}_i$.

A. Terminology

A single-source shortest paths tree $\mathcal{SP}$ is a directed subgraph, $\mathcal{G}'(V', E')$, of $\mathcal{G}(V, E)$ such that V' consists of exactly those vertices in $\mathcal{G}$ that are reachable from the source vertex, and E' consists of exactly those edges that form the least cost paths from the source to every vertex.

In the standard *shortest paths problem*, we are given a weighted, directed graph $\mathcal{G}(V, E)$, with cost function $cost : E \rightarrow (0, \infty)$, mapping edges to positive, real-valued costs [3], [17]. For our problem, we associate costs with vertices rather than edges, since the cells associated with the vertices represent the physical space through which a trajectory must pass. We could equivalently assess the cost of each edge as the average of the costs of the vertices it connects. For our problem, the edges of a connectivity graph $\mathcal{G}$ are undirected; however, the algorithms developed in this paper do not depend on this fact. Also, in this paper we will use the term "least cost" instead of "shortest."

Throughout this paper, we will denote the root of a tree by the label v_0. We use the usual definitions for neighbor, parent, child and sibling. Note that the neighbor relation applies to the underlying graph $\mathcal{G}$, while the parent, child, and sibling relations are specific to the embedded tree $\mathcal{G}'$. We will use the term proper-neighbor to denote those neighbors of a vertex, v, that are neither the parent, child or sibling of v. We will use the notation $\text{parent}_{\mathcal{G}'}(v)$ to denote the parent of a vertex in the unique simple path from v_0 to v in the tree $\mathcal{G}'$. These relationships are illustrated in Fig. 4, which emphasizes the fact that there can be multiple children (C), siblings (S), and proper-neighbors (N) of a vertex (V), but only one parent (P). Note that siblings are not necessarily neighbors. Arrows in the figure represent directed tree edges on top of undirected edges in the underlying graph.

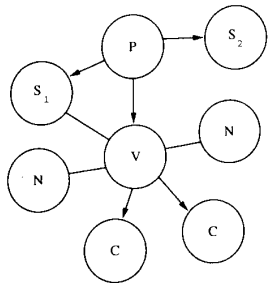


Fig. 4. Vertex relationships.

The following definition and lemma establish an alternative formulation of a single-source shortest paths tree. This formulation is central to the understanding of the algorithms presented in Section V.

Definition 8: Given a tree $\mathcal{G}'(V', E')$ embedded in a graph $\mathcal{G}(V, E)$, with root v_0, a vertex $v \in V$ is *locally* $\mathcal{SP}$ (abbreviated $l\mathcal{SP}$) if and only if

$$parent_{\mathcal{G}'}(v) = \begin{cases} \emptyset & \text{if } \pi_{\mathcal{G}'}(v_0, v) = \emptyset \\ p & \text{otherwise} \end{cases}$$

where p is such that $pathcost(\pi_{\mathcal{G}'}(v_0, p)) \leq pathcost(\pi_{\mathcal{G}'}(v_0, n))\ \forall n \in neighbors(v)$ and $\pi_{\mathcal{G}'}(v_0, v)$ denotes the unique simple path from the root v_0 to the vertex v.

Lemma 2 [$\mathcal{SP}$]: Given a tree $\mathcal{G}'(V', E')$ embedded in a graph $\mathcal{G}(V, E)$, with root v_0, $\mathcal{G}'$ is a single-source shortest paths tree embedded in $\mathcal{G}$ if and only if every $v \in V$ is $l\mathcal{SP}$.

The proof is very similar to that given for the correctness of Dijkstra's algorithm which can be found in many sources [18], [3], [17]. The complete proof of the lemma is given in [9].

The significance of the $\mathcal{SP}$ lemma is that it provides a *local* decision criterion for establishing and determining whether a particular tree, $\mathcal{G}'$, is a single-source shortest paths tree embedded in $\mathcal{G}$.

B. Changing Costs

We now examine the simple case where $\mathcal{G}_{i+1}$ is obtained by altering the cost of a single vertex v in $\mathcal{G}_i$ (and hence its path cost). In particular, in this section we identify a set of conditions that determine which vertices must remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$, and which vertices may not be $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Those vertices that may not be $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ must be considered by the dynamic single-source shortest paths tree algorithms presented in Section V. The major results of this section are contained in a number of theorems and their corollaries, which are summarized in Table I. Most of the proofs have been omitted for length considerations and may be found in [9]. The

TABLE I
SUMMARY OF RESULTS PRESENTED IN SECTION IV: LET $\mathcal{G}'_i$ BE A SINGLE-SOURCE SHORTEST PATHS TREE EMBEDDED IN A GRAPH $\mathcal{G}_i$; LET $\mathcal{G}_{i+1}$ BE THE GRAPH OBTAINED BY CHANGING COST (v) FOR SOME VERTEX $v \in \mathcal{G}'_i$ AND LET $\mathcal{G}'_{i+1}$ BE $\mathcal{G}_i$ WITH THE NEW VALUE FOR COST (v)

The parent of v remains $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.	Theorem 3 and Theorem 5
The siblings of v remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.	Theorem 3 and Theorem 5
For increases to cost(v), the non-descendants of v remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.	Theorem 4 and Theorem 5
For all decreases to cost(v), the descendants of v remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.	Theorem 6
If the path cost of a vertex v decreases by more than pathcost($\pi_{\mathcal{G}'_i}(v_0, v)$) - pathcost($\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(n))$), for a proper-neighbor n, then n remains $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ if it exchanges its parent for v.	Corollary 8

omitted proofs follow from standard facts on shortest paths and the theory introduced in this paper.

Throughout this section we will assume that the graph $\mathcal{G}_{i+1}$ is obtained by changing the cost of some vertex v in $\mathcal{G}_i$. Structurally, $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ are identical. The tree $\mathcal{G}'_i$ is embedded in $\mathcal{G}_i$ and is a single-source shortest paths tree. The tree $\mathcal{G}'_{i+1}$ is embedded in $\mathcal{G}_{i+1}$, and is structurally identical to $\mathcal{G}'_i$. However, $\mathcal{G}'_{i+1}$ has a new value for cost(v), and therefore, in general, $\mathcal{G}'_{i+1}$ is *not* a single-source shortest paths tree.

Definition 9: For a specific change to cost (v), we say that a vertex u is *independent* of v if and only if

1) *pathcost*($\pi_{\mathcal{G}'_{i+1}}(v_0, u)$) = *pathcost*($\pi_{\mathcal{G}'_i}(v_0, u)$)
2) $\pi_{\mathcal{G}'_{i+1}}(v_0, u) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$.

where $\Pi^*_{\mathcal{G}}(v_0, v)$ denotes the set of all simple least cost paths from v_0 to v in $\mathcal{G}$.

In other words, the least cost path from the source, v_0, to vertex u is insulated from the particular change to the cost of vertex v.

Definition 10: For a particular change to cost(v), the set of *affected* vertices, $\mathcal{AFF}(v)$, is given by

$$\mathcal{AFF}(v) = \{u \in V \mid u \text{ is not independent of } v\}.$$

The following theorem provides a local condition that is sufficient to establish that some vertex u is independent of v for a specified change in cost (v). The proof is provided in the Appendix.

Theorem 3 [Independence]: A vertex $u \neq v$ is independent of v for *all* changes to cost (v) if

$$\text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(u))) \leq \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(v))). \quad (1)$$

We can also state a weaker property that holds for nondescendants of v, which includes all proper-neighbors of v. Let $D(v)$ denote the descendents of v.

Theorem 4 [Nondescendants]: The nondescendants of v are independent of v for all increases in cost (v).

The previous theorems provide local conditions that are sufficient to test whether a vertex u is independent of some other vertex v for a specified change in the cost (v). We now present a theorem that relates independence to the $l\mathcal{SP}$ property.

Theorem 5: A vertex u that is independent of v in $\mathcal{G}_i$ for a particular change to cost (v) is $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.

The next theorem identifies conditions for the descendants of v to remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.

Theorem 6 [Descendants]: The set of descendants of v remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ for all decreases to cost (v).

Because the previous results provide only sufficient conditions, we cannot infer which vertices fail to remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. The final theorem of this section identifies sufficient conditions for a set of vertices to *not* remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Furthermore, the corollary specifies how to rectify the situation.

Theorem 7 [Proper-Neighbor]: A vertex n that is $l\mathcal{SP}$ in a tree $\mathcal{G}'_i$ embedded in a graph $\mathcal{G}_i$ is not $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ if the path cost of a proper-neighbor v of n changes such that

$$\text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, v)) < \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(n))).$$

Corollary 8 [Proper-Neighbor]: A proper-neighbor n that is no longer $l\mathcal{SP}$ by the proper-neighbor theorem, will become $l\mathcal{SP}$ by exchanging its parent for the vertex v of the proper-neighbor theorem.

The results of this section show that when the cost of a single vertex v *decreases*, the parent, siblings, children, and some proper-neighbors of v remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Those proper-neighbors n of v that are not $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ will become $l\mathcal{SP}$ if the parent of n becomes v. The corresponding decrease in the path cost of n can be propagated to its neighbors in a similar fashion. When the cost of v *increases*, by Theorem 4, the nondescendants remain $l\mathcal{SP}$, while the descendants may not remain $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. These results are illustrated in Fig. 5. In the figure, vertex v is depicted together with its descendants, $D(v)$. When the cost of v decreases, the preexisting descendants remain descendants, and those proper-neighbors that are no longer $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ become descendants of v. When the cost of v increases, the descendants may not be $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ and may therefore obtain alternate paths from the source. The arrows in the figure illustrate the general trends in the changing paths.

The importance of the theorems and corollaries presented in this section stems from the $\mathcal{SP}$ lemma. We want to compute a single-source shortest paths tree in $\mathcal{G}_{i+1}$. The $\mathcal{SP}$ lemma tells us that if every vertex is $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ then we have solved the problem. The preceding theorems and their corollaries identify

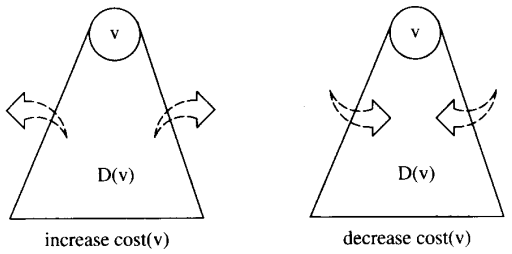


Fig. 5. Effects of changing the cost of a vertex.

large, easily recognized subsets of vertices that are already $lS\mathcal{P}$ after a given change to the graph and therefore need no further processing.

C. Changing the Graph

In our planning application, a selected vertex v_s is deleted from $\mathcal{G}_i$, its corresponding MIXED cell κ_s is subdivided, and the vertices corresponding to the non-FULL cells of $\mathcal{P}^{\kappa_s}$ are added to construct $\mathcal{G}_{i+1}$. The new vertices also inherit a subset of the neighbors of v_s. The cell adjacency tests take only a constant amount of time for rectangloid cells. So computing $\mathcal{G}_{i+1}$ from $\mathcal{G}_i$ is a local operation of complexity on the order of the number of neighbors of v_s. All edges incident on, or emanating from, v_s are removed from $\mathcal{G}'_i$ as well as from $\mathcal{G}_i$. As a result of this modification to $\mathcal{G}'_i$, children (v_s) do not have parents in $\mathcal{G}'_{i+1}$, and $\text{parent}_{\mathcal{G}'_i}(v_s)$ has one less child in $\mathcal{G}'_{i+1}$. Also, each newly created vertex has no parent in $\mathcal{G}'_{i+1}$.

If we could make a local repair to $\mathcal{G}'_i$ to integrate the new and orphaned vertices, then we could propagate the changes to the graph as if we had merely changed the cost of the deleted vertex. Unfortunately, integrating the new and orphaned vertices into $\mathcal{SP}_{i+1}$ in their permanent positions is as difficult as computing $\mathcal{SP}_{i+1}$ from scratch. The general case is more difficult than the case of a single vertex changing its cost, because the cumulative effect that introducing the new vertices has on the affected vertices in $\mathcal{G}'_{i+1}$ is uncertain. There are two sources for this difficulty: the net change in the vertex costs relative to cost (v_s), and the topological changes from $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$.

The parent of v_s together with the new and orphaned vertices do not necessarily form a connected subgraph of $\mathcal{G}_{i+1}$. Therefore, locally, the neighbor with smallest path cost is not necessarily known. For example, any new vertex v that is adjacent to $p = \text{parent}_{\mathcal{G}'_i}(v_s)$ becomes a child of p. However, there may not be any such v. If we greedily choose the nonchild neighbor with the minimum path cost, it is unlikely that this local repair is globally consistent. Although propagation guarantees global consistency, this greedy repair may result in an inefficient algorithm [10].

So in the general case we must propagate both increases and decreases in parallel. A decrease may override an increase, or may cancel it out. Or, due to the altered topology of the graph, the separate changes may be independent of one another. An algorithm for propagating path costs must be able to account for this behavior as well as what was observed for the simple case. These issues are addressed in the next section in the context of our specific implementations.

V. ALGORITHMS

Dijkstra's algorithm [18] is a well-known solution to the single-source shortest paths problem for a given graph $\mathcal{G}(V, E)$ with nonnegative edge weights. Dijkstra's algorithm can be viewed as a *direct implementation* of the $\mathcal{SP}$ lemma. The algorithm begins by initializing the pathcost of the source vertex to 0[1] and the pathcost of all other vertices to ∞.

The algorithm proceeds by progressing from the source, incrementally outward. At each iteration, a vertex v_s is selected and added to the list of vertices for which the least cost path from the source is known. We will refer to such vertices as "stable." The neighbors of v_s are then tested to see whether their current least cost path can be bettered by going through v_s. At this time, a neighboring vertex with its initial infinite path cost will obtain a finite path cost. Since the algorithm progresses from the source incrementally outward, by the time a vertex is added to the list, it has been notified of all neighbors with lower cost paths and has therefore attained the least cost path to the source, and is therefore also $lS\mathcal{P}$.

The original algorithm [18] is understood to use a $|V| \times |V|$ adjacency matrix to represent the graph and runs in time $\mathcal{O}(|V|^2)$. Since the number of edges can be as large as $\mathcal{O}(|V|^2)$, in general, this is optimal [48]. However for sparse graphs, where $|E| \ll |V|^2$, Dijkstra's algorithm can be modified to run in time $\mathcal{O}(|E| \log |V|)$ through the use of a priority queue [3], [17], where the priority of a vertex is the value of pathcost. (The use of Fibonacci heaps [28], for example, leads to further theoretical improvements.) An additional improvement to average case running time can be achieved by a slight modification to Dijkstra's algorithm: initially place only the source vertex, v_0, in the queue, and add vertices to the queue as they are reached by the propagation. We will refer to this modified Dijkstra's algorithm as "the static Dijkstra's algorithm." We use this algorithm as a benchmark for our comparisons in Section VI.

In the worst case, there can be $\mathcal{O}(|V|^2)$ edges in a connectivity graph. However, as is shown in [8], the connectivity graph $\mathcal{G}(V, E)$ of an octree is sparse. Specifically $|E| < \frac{30}{7}|V|$. Therefore, the complexity of Dijkstra's algorithm is bounded by $\mathcal{O}(|V| \log |V|)$.

In the remainder of this section we will describe two new algorithms. In Section V-A we present a modification of Dijkstra's algorithm that takes advantage of the existing structure of $\mathcal{G}'_{i+1}$. Then in Section V-B we present a second modification of Dijkstra's algorithm that also takes advantage of $\mathcal{G}'_{i+1}$ but is a less conservative algorithm. Finally, in Section V-C we discuss other solutions to the dynamic single-source shortest paths problem and to related problems.

Throughout this section, we assume that a vertex v and the relationships between vertices described in Section IV-A are represented by a data structure with the following fields: *cost*, the assessed cost of κ, *pathcost*, the globally best known least path cost, and *localpc*, the locally best known least path cost. The *localpc* field is only used by the dynamic algorithm of Section V-B. The vertex relationships are represented by the *parent, children,* and *neighbors* fields. During the execution of

[1] Any value less than ∞ will serve the same purpose.

the algorithms a vertex may not be $lS\mathcal{P}$, but each algorithm developed below first establishes and then maintains this relationship for all vertices in $\mathcal{G}$, to construct $\mathcal{SP}$. This is discussed in more detail below.

It is important to note that the complexity analysis of the algorithms presented in this section is a worst case analysis. It may be that an average case analysis, and, for example, the use of dynamic data structures, will yield better expected time bounds. As we note below, in the worst case, the entire graph may change and the dynamic algorithms perform no better than the static Dijkstra's algorithm. However, for problems where this is not typical, such as hierarchical motion planning, the dynamic algorithms do much better.

A. A Dynamic Batch Algorithm

The static Dijkstra's algorithm suffers from the drawback of initializing all of the graph vertices and computing $\mathcal{SP}_{i+1}$ from scratch for each $\mathcal{G}_{i+1}$. In this section we present an algorithm that makes use of the existence of $\mathcal{G}'_i$ to update only those vertices that might be affected by a given modification to $\mathcal{G}_i$. Specifically, we apply the static Dijkstra's algorithm only to the potentially small set of affected vertices.[2] This yields a dynamic batch algorithm, which we will refer to as $\mathcal{DB}$.

We identified in Section IV those vertices that are independent of a given change to the graph. There is no need to re-initialize such vertices as they are already correct. The descendants of v_s, the vertex that has changed, are not independent of v_s, and may require modification.

We know (by Theorems 3, 5, and 7) that if the new vertices act as if the vertex cost of v_s were increased in $\mathcal{G}_{i+1}$, then only the descendants of v_s might not be $lS\mathcal{P}$ in $\mathcal{G}'_{i+1}$. By collecting the descendants and initializing their *pathcost* to ∞, as is done for the static Dijkstra's algorithm, we avoid having to consider the troublesome case of increasing a *pathcost*. Relative to the initialized *pathcosts*, the effective change to the graph is to decrease the *cost* of a vertex. While there is only one vertex with an initially bounded *priority* in the static Dijkstra's algorithm, in $\mathcal{DB}$ there are many. This means that the initialization process of the static Dijkstra's algorithm must be extended somewhat. After every descendant is initialized to have infinite *pathcost*, every descendant is then tested to see whether or not it is adjacent to some vertex in the set of stable vertices, and, if so, what its least cost path is.

In the case of an effective decrease in the cost of the vertex, the descendants decrease their path costs, and some proper-neighbors may no longer be $lS\mathcal{P}$. In this case, $\mathcal{DB}$ functions similar to the static Dijkstra's algorithm, incrementally adding such vertices to the priority queue.

In the pseudocode for $\mathcal{DB}$, shown in Fig. 6, Divide-Vertex is responsible for the following five tasks:

1) Delete the vertex v_s and associated edges from $\mathcal{G}_i$ and $\mathcal{SP}_i$.
2) Subdivide the cell κ_s associated with v_s.
3) Label the new cells of $\mathcal{P}^{\kappa_s}$ EMPTY, MIXED, or FULL.
4) Create new vertices for the non-FULL cells of $\mathcal{P}^{\kappa_s}$.
5) Add the new vertices to construct $\mathcal{G}_{i+1}$.

[2] A similar idea is presented in [16].

```
procedure Split-Vertex (v:vertex; G:graph)
[1]   D ← Init-and-Return-All-Descendants(v)
[2]   N ← Divide-Vertex(v, G)
[3]   DB (D ∪ N)
end

procedure DB (affected:vertex-list)
[1]   DB-Initialize(affected)
[2]   Q ← BuildPQ(affected)
[3]   while Q ≠ ∅ do
[4]       u ← DeleteMin(Q)
[5]       foreach v ∈ u.neighbors do
[6]           Relax(u, v, Q)
end

procedure DB-Initialize (vertices:vertex-list)
[1]   foreach v ∈ vertices do
[2]       p ← Min-Neighbor(v)
[3]       v.pathcost ← v.cost + p.pathcost
end

procedure Relax (u, v:vertex; Q:priority-queue)
[1]   if v.pathcost > v.cost + u.pathcost
[2]   then v.parent ← u
[3]        v.pathcost ← v.cost + u.pathcost
[4]        Adjust-Vertex (v, Q)
end

procedure Adjust-Vertex (v:vertex, Q:priority-queue)
[1]   if v ∈ Q
[2]   then UpdatePriority(v, Q, v.pathcost)
[3]   else Insert(v, Q, v.pathcost)
end
```

Fig. 6. Dynamic batch algorithm for constructing a single-source shortest paths tree.

Divide-Vertex returns the set of newly created vertices. Min-Neighbor returns the neighboring vertex with the minimum pathcost (the posted least path cost from the source). Init-and-Return-All-Descendants performs a tree-traversal, collecting the descendants. As a side-effect, each descendant vertex has its *pathcost* reset to infinity and its *parent* reset to NIL. Split-Vertex is the top-level function that is called by the FindPath algorithm.

In the complexity analysis for $\mathcal{DB}$ that follows, let v_s be the vertex that is subdivided, let $|v|$ be $|\mathcal{AFF}(v_s)|$, and let $|e|$ be the number of edges incident on $\mathcal{AFF}(v_s)$. We use $|v|$ and $|e|$ as subset analogs of $|V|$ and $|E|$.

The complexity analysis for $\mathcal{DB}$ is very similar to that of the static Dijkstra's algorithm. There is a linear amount of work during initialization: DB-Initialize calls Min-Neighbor which yields an $\mathcal{O}(|e|)$ instead of the $\mathcal{O}(|V|)$ preliminary step of the static Dijkstra's algorithm; and BuildPQ requires only $\mathcal{O}(|v|)$ time instead of $\mathcal{O}(|V|)$. Each vertex is deleted only once from the priority queue and Relax is called once for each neighbor. Thus the complexity of $\mathcal{DB}$ is $\mathcal{O}(|e| \log |v|)$. Again,

in the worse case, when the source vertex changes its cost, $\mathcal{DB}$ takes time $\mathcal{O}(|E| \log |V|)$. In the event that v_0 changes its cost, $\mathcal{DB}$ is equivalent to the static Dijkstra's algorithm, although $\mathcal{DB}$ will perform some extraneous initializations without undo overhead. However, on average, it is expected that $|e| \ll |E|$ and $|v| \ll |V|$.

B. A Dynamic Incremental Algorithm

In this section, we present a dynamic incremental algorithm, $\mathcal{DI}$, that incrementally propagates changes to only that subset of the graph that is actually affected. The critical distinction between $\mathcal{DI}$ and $\mathcal{DB}$ is that $\mathcal{DI}$ must propagate increases as well as decreases. Most notably, Split-Vertex initializes only those vertices that have been modified, as opposed to those that are potentially affected, as for $\mathcal{DB}$. But as we shall see below, $\mathcal{DI}$ handles vertices that increase their path cost in much the same fashion as $\mathcal{DB}$: by resetting the *pathcost* field to ∞, thereby converting the vertex to one whose *pathcost* is monotonically decreasing. Here it becomes clear why we need separate *localpc* and *pathcost* fields: we need to distinguish between the local properties of a vertex and the global properties that we are attempting to establish.

There are two modifications to $\mathcal{DB}$ that must be made to accommodate the case of a parent that increases its path cost. The first modification is that the relaxation procedure must explicitly test for, and handle, vertices whose parents have increased their *pathcost*. When a parent propagates an increase, the locally best path of the child is reevaluated. At this point there is not sufficient information to determine whether the child's path cost will eventually increase or decrease.

The conservative solution is to make the *priority* of a vertex to be the smaller of the *pathcost* and the locally best path cost (which may change as other changes are propagated). This avoids committing to an apparent increase too early.

The only time we can be *sure* that the path cost will increase is when the child is selected as the vertex with the minimum *priority* (the purportedly least path cost) to be added to the set of stable vertices. At that time, if the vertex is adjacent to some stable vertex and also has the least cost path, then it is correct to add the vertex to the set of stable vertices. However, if the *pathcost* is smaller than the *localpc*, we are guaranteed that the vertex path cost *must* increase. In other words, the *pathcost* has been held artificially low in case a lower cost path is discovered to avoid an unnecessary increase, but no such path was found.

So, the second modification to $\mathcal{DB}$ is to test that when a vertex is deleted from the priority queue, if its *pathcost* is less than its *localpc*, the vertex is initialized to have infinite *pathcost*, re-inserted into the priority queue (with *priority* equal to the locally least path cost, unless it is not reachable from the source), and the increase is propagated to its neighbors. A vertex is therefore deleted from the priority queue at most twice. This argument is also given in [42].

We consider the case of a neighbor with no parent to be the same as if a parent has increased to accommodate new vertices. Also, if after relaxation, a vertex is found to have its *localpc* equal to its *pathcost*, then it need not be inserted, and can even be deleted from the priority queue, as this condition entails that the vertex is effectively unchanged by the change to the graph [42]. The pseudocode for $\mathcal{DI}$ is shown in Figs. 7 and 8.

In the complexity analysis for $\mathcal{DI}$ that follows, let v_s be the vertex that is subdivided, let $|v|$ be $|\mathcal{AFF}(v_s)|$, and let $|e|$ be the number of edges incident on $\mathcal{AFF}(v_s)$. Let M be the initial set of modified vertices (*i.e.* the new vertices from $\mathcal{P}^{\kappa_s}$, and the orphaned vertices, $v_s.children$), and let $|N|$ be the number of edges incident on M. We assume without loss of generality that $|N| < |e|$ and $|M| < |v|$.

BuildPQ is used to create an empty priority queue, which takes constant time. DI-Init calls Min-Neighbor for each vertex in M and then inserts the vertex into the priority queue. The only situation in which a vertex is not inserted into the priority queue is when it is no longer a part of the connected subgraph containing the source. Therefore DI-Init takes time $\mathcal{O}(N + M \log M)$. In the worst case, DI-Relax on Line 13 requires $\mathcal{O}(d + \log |v|)$ time, where d is the average degree of a vertex (*i.e.* $|E| = d|V|$). This is the result of one call to Min-Neighbor and one call to Adjust-Vertex, respectively.

The while loop on Line 3 of $\mathcal{DI}$ iterates $\mathcal{O}(|v|)$ times. Each iteration produces a call to DeleteMin, and, in the worst case, a call to Min-Neighbor and to Insert. Each iteration also produces d calls to DI-Relax. This yields an $\mathcal{O}(|v|(\log |v| + d + d(d + \log |v|))) = \mathcal{O}(|v|d(d + \log |v|)) = \mathcal{O}(|e|(d + \log |v|))$ time complexity for $\mathcal{DI}$. Assuming that $d \leq \log |v|$, the complexity is $\mathcal{O}(|e| \log |v|)$. Because $|E| = \mathcal{O}(|V|)$, this assumption is reasonable. An alternative solution with this bound can be found in [42].

In the worst case $|v|$ is $\mathcal{O}(|V|)$ which occurs when v_s is near v_0 and the subcells of $\mathcal{P}^{\kappa_s}$ produce a drastic effective change in path costs relative to $v_s.pathcost$. $\mathcal{DI}$ behaves like $\mathcal{DB}$ in this case: first accumulating the descendants in the priority queue, initializing their path costs, and then monotonically decreasing the path costs until their minimum value is obtained.

When compared to $\mathcal{DB}$, $\mathcal{DI}$ examines a potentially smaller set of vertices at a slightly higher cost per vertex. The higher cost is due in part to examining each vertex more than once, but is predominantly due to the need to assess the locally least cost path during relaxation.

C. Related Problems

Much of the research on computing single-source shortest paths trees focuses on arbitrarily changing a single edge [3], [17], [1], [41]. While we can arbitrarily change the cost of a vertex, this effectively results in a uniform change to the weights of all incident edges. Also many researchers limit themselves to the case of unit cost or small integer cost edges. For example, the assumption of small integer costs yields a time bound of $\mathcal{O}(|E| + |V|\sqrt{\log W})$ to compute $\mathcal{SP}$, where W is the number of bits required to represent the largest cost [17]. An efficient solution to the dynamic single-source shortest paths problem where multiple simultaneous changes to the graph are allowed can be found in [42].

An incremental dynamic algorithm was independently developed in [42]. We will call this algorithm $\mathcal{RR}$. $\mathcal{DI}$, as originally proposed, differs from $\mathcal{RR}$ in three ways.

```
    procedure DI (modified:vertex-list)
[1]     DI-Init(modified, Q)
[2]     Q ← BuildPQ(NIL)
[3]     while Q ≠ ∅ do
[4]         u ← DeleteMin(Q)
[5]         if u.pathcost ≥ u.localpc
[6]         then u.pathcost ← u.localpc
[7]         else u.pathcost ← ∞
[8]              u.parent ← Min-Neighbor(u)
[9]              u.localpc ← u.cost + u.parent.pathcost
[10]             if u.pathcost ≠ u.localpc
[11]             then Insert(u, Q, u.localpc)
[12]        foreach v ∈ u.neighbors do
[13]             DI-Relax(u, v, Q)
    end

    procedure DI-Init (vertices:vertex-list, Q:priority-queue)
[1]     foreach v ∈ vertices do
[2]         v.parent ← Min-Neighbor(v)
[3]         v.localpc ← v.cost + v.parent.pathcost
[4]         if v.pathcost ≠ v.localpc
[5]         then Insert(v, Q,min(v.pathcost, v.localpc))
    end

    procedure DI-Relax (u, v:vertex; Q:priority-queue)
[1]     c ← v.cost + u.pathcost
[2]     if v.parent = NIL or (v.parent = u and v.localpc < c)
[3]     then v.parent ← Min-Neighbor(v)
[4]          v.localpc ← v.cost + v.parent.pathcost
[5]          Adjust-Vertex(v, Q)
[6]     elseif v.localpc > c
[7]     then v.parent ← u
[8]          v.localpc ← v.cost + v.parent.pathcost
[9]          Adjust-Vertex(v, Q)
    end
```

Fig. 7. Dynamic incremental algorithm for constructing a single-source shortest paths tree.

1) $\mathcal{RR}$ deletes vertices from the priority queue as soon as they are discovered to be correct ($v.pathcost = v.localpc$). This results in a smaller priority queue (and hence the primitive priority queue operations are more efficient) and vertices are not relaxed unnecessarily. We have included this modification to $\mathcal{DI}$ in the pseudocode presented in Section V-B.
2) $\mathcal{DI}$ keeps track of the least path cost neighbor, which saves time computing the path cost of a vertex. $\mathcal{RR}$ addresses this problem by keeping a subset of the neighbors of a vertex in a local priority queue so that the least cost neighbor is always immediately available. Although we considered this solution, we have not implemented it, in part, because a standard implementation of priority queues would also increase the overall complexity of the algorithm. It is also the case for $\mathcal{RR}$ that the use of local queues increases the complexity of their algorithm for a standard implementation of priority queues. The authors suggest implementing the local priority queues as Fibonacci heaps [28]; but then Fibonacci heaps should be used for the global priority queue as well.

```
    procedure Split-Vertex (v:vertex; G:graph)
[1]     M ← v.children
[2]     N ← Divide-Vertex(v,G)
[3]     DI (N ∪ M)
    end

    procedure Adjust-Vertex (v:vertex, Q:priority-queue)
[1]     if v.pathcost = v.localpc
[2]     thenif v ∈ Q
[3]         then Delete(v, Q)
[4]     elseif v ∈ Q
[5]         then UpdatePriority(v, Q,min(v.pathcost, v.localpc))
[6]         else Insert(v, Q,min(v.pathcost, v.localpc))
    end
```

Fig. 8. Dynamic incremental algorithm for constructing a single-source shortest paths tree (continued).

3) $\mathcal{RR}$, when applied to our problem, will initially consider all of the neighbors of the vertex that has been deleted from the graph. $\mathcal{DI}$ only considers the children of the vertex. Since the modification to the graph is to insert new vertices with initially infinite path costs, the other neighbors maintain their correctness initially. $\mathcal{RR}$ will later detect this and not insert such vertices into the queue.

Some research on dynamic shortest paths trees focuses on the restricted problem of planar graphs [25] for which it is known how to efficiently compute good separators [26]. Our analysis would not benefit from considering planar graphs since our connectivity graphs are already sparse. The ability to compute good separators in connectivity graphs is considered in Section VII which discusses future research.

There is a considerable amount of research dedicated to the dynamic all-pairs shortest paths problem [44], [22], [6], [38], [41], [42], [35]. This may be of some interest if we want to consider many different planning problems within the same environment. Dynamically maintaining the all-pairs shortest paths may be more efficient than computing $\mathcal{SP}$ for each new problem.

There is also considerable research on minimum spanning trees (MST) [3], [26], [17], [27], [2], [21]. Of these [21] presents the best known bound for dynamically maintaining a MST.

Other related problems include network flows, matching, the simplex method, and computational circuit analysis [37], [30], [4], [5]. The connectivity graph looks like the dual (Delaunay triangulation) of the Voronoi diagram [20] constructed from the centroid of each cell with the distance metric an individually weighted city-block distance. This observation may lead to insights and an alternate formulation of the problem, perhaps based on a topological plane sweep [20].

VI. Experimental Results

In this section we present a number of experimental results. In Section VI-A we describe a number of implementation issues. In Section VI-B we compare the different algorithms to compute a single-source shortest paths tree on randomly generated problems. The results show that the dynamic al-

gorithms, $\mathcal{DB}$ and $\mathcal{DI}$, are comparable to each other, and that they are better than the static Dijkstra's algorithm. In Section VI-C we compare our algorithms in a planning context with the best known approximate cell decomposition planners reported in the literature. Finally, in Section VI-D, we present several additional typical problems that have been solved by our planner.

Statistics for each experiment were collected for random planning problems within twelve of the environments taken from the motion planning literature [13], [14], [7], [49]. The problems were created by generating random (x, y, θ) tuples and testing each to ensure the distance from $\mathcal{CB}$ was at least the minimum resolution. The twelve environments and sample planning problems are shown in Fig. 9. The illustrated problems are similar to those given in [49]. The trajectories shown are not optimized, smoothed or interpolated. The trajectories are illustrated by drawing the robot at a discrete set of points along the trajectory. These points are q_{init}, q_{goal}, and the centroids of the intersection of two adjacent cells of the solution E-channel, $\overline{\kappa_i \cap \kappa_{i+1}}$. Thus, a large gap between successively drawn robots implies that the underlying cell is quite large. Similarly, where successively drawn robots are densely packed, there is a very fine underlying subdivision of $\mathcal{C}$.

We have also compared our algorithms on thousands of random problems within dozens of randomly generated environments. We found these problems to be unrepresentative of practical problems. A large proportion of the problems were either trivial or impossible. Simple measures of environmental complexity are also unsatisfactory. For example, [49] define complexity as the number of edges in the input workspace, and sparsity as the minimum vertical distance between workspace obstacles. This complexity measure does not distinguish between a square with a small "step" taken out, an octogon, and a four-pointed star; and this sparsity measure does not account for the density of obstacles or for the surface area of the limiting gap between critical obstacles. Ideally, we would like to control for the number and lengths of potential passageways between the initial and goal configurations, however, it is difficult to parameterize an environment generator with these variables.

A. Implementation Issues

The vertex cost function determines which path is the least cost path and therefore the function is instrumental to the convergence of the FindPath algorithm. In our experiments, we have used the vertex cost function

$$\text{cost}(v) = \alpha\,(e^{-\beta\rho(v)} - e^{-\beta})\,(\text{max-volume} - \text{volume}(v)) + 1 \tag{2}$$

where $\rho(v) = 1 - \dfrac{\text{full-volume}(v)}{\text{volume}(v)}$, volume$(v)$ is the volume of the cell associated with v, max-volume is the volume of κ_0, and full-volume(v) is the relative volume of $\mathcal{CB}$ in the cell. The parameters α and β are used to adjust the relative effects of the terms. This function grows rapidly for cells that contain small proportions of FREE, prefers larger cells over smaller cells, is positively valued, and is continuous. It is also the case that the cost of a vertex with an EMPTY cell is no greater than the cost of a vertex with a MIXED cell. We estimate full-volume(v) by averaging slices of $\mathcal{CB}$ within κ. In particular, each slice in θ yields a set of (overlapping) polygons [39], and we compute the area of the union of these polygons using a plane sweep algorithm [40], [20]. Our estimate of full-volume(v) has the property that cells labeled EMPTY or FULL are correctly estimated. In [8] we show that α has little effect on performance, and that any reasonable value of β gives good performance.

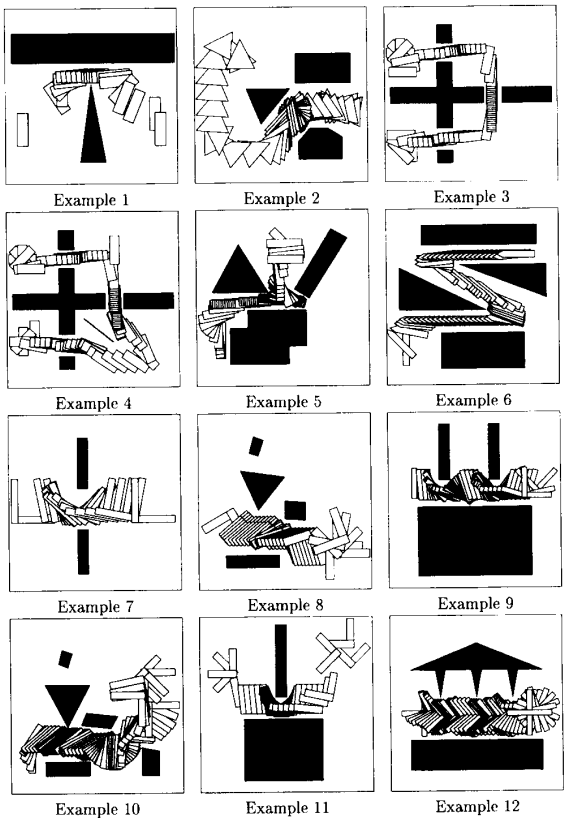


Fig. 9. Twelve environments from the literature.

At each iteration of FindPath, the MIXED cell in the least cost path with the maximum cost is selected for subdivision. This cell selection policy causes the planner to work out the most difficult constraints on the solution trajectory first (squeezing through narrow gaps) the remaining constraints, prioritized by the size of the bottleneck presented, are incrementally easier to satisfy. In [8] we show that cost is the best criterion for cell selection, followed closely by the proportion of free space in a cell (minimizing). Cell volume was a reasonable selection method (maximizing); and cell location within the least cost path proved to be quite poor. Indeed, all policies based on location produced larger solution graphs than vertices chosen at random (within the least cost path).

B. Static versus Dynamic Algorithms

In this experiment, (random) graphs were generated and randomly perturbed in order to measure the inherent differences between the static Dijkstra's algorithm and our dynamic

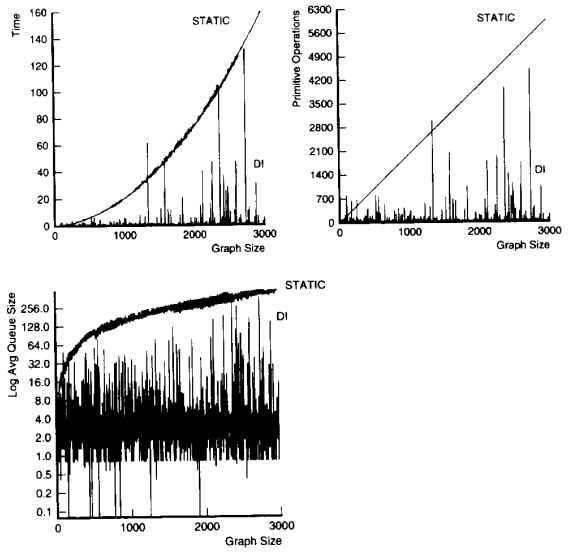


Fig. 10. Total work spent repairing $\mathcal{SP}$ during random subdivisions.

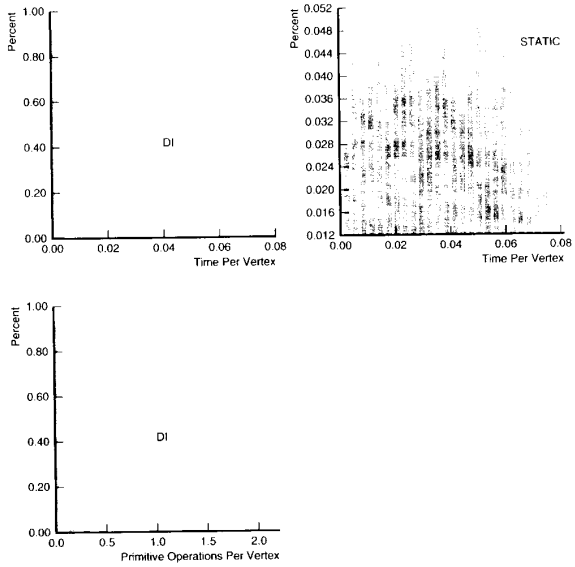


Fig. 11. Work spent per vertex repairing $\mathcal{SP}$ during random subdivisions.

algorithm $\mathcal{DI}$. We implemented this by maintaining a single-source shortest paths tree subject to a fixed number of random subdivisions for each of the twelve environments. Specifically, first a random point was selected as the initial configuration, for each of the twelve environments. Then, at each iteration, a cell was randomly selected for subdivision, and a single-source shortest paths tree constructed. There were 600 iterations for each environment and initial configuration.

For the dynamic algorithm, independent of the size of the graph, vertices on the fringe of the single-source shortest paths tree are inherently cheaper to repair than those near the source. In other words, the hardest problem in a large graph is much harder than the hardest problem in a small graph. At the same time, the easiest problem is the same for all graphs. It is also the case, in general, that the number of hard problems does not grow as quickly as the number of easy problems as the size of the graph increases.

During each experiment, several quantities were measured after each subdivision:

- The total time spent constructing $\mathcal{SP}_{i+1}$ is a rough measure of the amount of work performed by each algorithm.
- The total number of invocations to adjust the priority queue (the primitive operations are DeleteMin, Insert, UpdatePriority, and Delete), and the average size of the priority queue at each invocation, are intended to directly measure the performance of the $\mathcal{SP}$ algorithms. These quantities are machine independent.

The results are summarized in Figs. 10 and 11. Fig. 10 illustrates algorithm performance as a function of the size of the connectivity graph. For both time and priority queue operations, we see that the dynamic algorithm typically far outperforms the static Dijkstra's algorithm, occasionally is as bad as the static Dijkstra's algorithm, and rarely is worse than the static Dijkstra's algorithm. The base two logarithm of the average size of the priority queue for the algorithms is shown in the lower left of Fig. 10. The dynamic algorithm maintained a nearly constant queue size. This suggests that empirically, on average, the cost of each primitive queue operation is constant, implying an overall average time complexity of $\mathcal{O}(e)$ for the dynamic algorithm.

The histograms in Fig. 11 were constructed as follows:

1) We normalized the attribute values by dividing by the size of the graph. This yields "time per vertex" for example. Thus each attribute value is independent of the size of the graph.
2) A histogram is then constructed. On the horizontal axis of the histogram are the attribute values, and on the vertical axis we plot frequency of occurrence.
3) The histograms are then normalized so that the total area sums to 1. This renders the frequencies of occurrence into percentages of occurrence.

We observe that, on average, very little time is spent per vertex for the dynamic algorithm. Likewise, there are very few primitive operations performed per vertex for the dynamic algorithm. This is consistent with the fact that there are more vertices near the leaves of a single-source shortest paths tree than near the source. Since the static Dijkstra's algorithm does not take advantage of the preexisting single-source shortest paths tree, the amount of time per vertex is nearly uniform. Note that the static Dijkstra's algorithm spends exactly two primitive operations per vertex [8].

C. Planner Efficiency

In this section we compare our planner to BLP2 [13], [14] and ZL [49].[3] Both BLP2 and ZL use a bridge the gap strategy

[3]BLP2 results were obtained on an MIT Lisp Machine; ZL results were obtained on a Macintosh II running Allegro Common Lisp; our results were obtained on a SUN IPC running Lucid Common Lisp version 3. The sweepline code we use to estimate the volume of free space in a cell is implemented in C.

TABLE II
SUMMARY OF 12 PROBLEMS

example	plan time (min)				total cells			empty cells			channel length		
	ZL	BH	A*	BLP2	ZL	BH	BLP2	ZL	BH	BLP2	ZL	BH	BLP2
1	0.6	1.2	6.4		98	236		35	62		12	24	
2	0.9	2.0	13.2		140	349		74	148		18	56	
3	1.5	5.4	34.7		210	572		104	167		10	66	
4	2.5	8.1	47.5		264	662		134	228		28	81	
5	5.5	6.5	36.3	10s	293	602	644	96	241	120	36	79	87
6	5.0	4.5	24.4		218	487		116	140		29	71	
7	0.9	0.7	3.9		160	170		77	51		22	18	
8	2.5	3.3	14.4	10s	205	219	782	88	85	62	17	32	29
9	6.5	4.1	29.2		170	529		25	168		13	60	
10	9.8	7.5	31.3		206	442		46	159		19	67	
11	11.0	3.2	22.8	10s	312	451	727	72	138	127	21	50	79
12	10.5	2.6	12.8		369	287		121	112		18	46	

to select a path at each iteration of the FindPath algorithm. BLP2 employs the same representation of configuration space as our planner; but it uses binary subdivision for its spatial decomposition, and it subdivides every MIXED cell in the M-channel obtained at each iteration of the FindPath algorithm. ZL employs a more precise configuration space representation, and a novel spatial decomposition. With so many differences, the meaningful comparisons that can be made between our results and those given for ZL and BLP2 are limited to results given for ZL using octrees, and to the ratio of planning time to the number of vertices produced.

To compare planners, we reproduced the experiments given in [13], [14], [7], [49] and measured the total planning time (with the time to generate an initial representation of an environment separated from the actual planning time), the numbers of EMPTY and MIXED cells in the solution graph, and the number of cells in the solution E-channel, which we refer to as channel length. Our results are summarized in Table II (an extension of the table given by [49]). Execution times should be considered approximate due to the different implementations, architectures, and other experimental conditions. We use the abbreviation BH for our planner (using the $\mathcal{DI}$ algorithm). We also give the times for our planner using an uninformed A* search, which is faster than using the static Dijkstra's algorithm. Data for BLP2 was taken from [13], [14].

Note especially Examples 3 and 12. In Example 3, the bridge the gap strategy used by ZL is advantageous when the heuristic fits the problem and there are several good alternatives to consider. In Example 12, the bridge the gap strategy hinders the planner. Note also that there is no correspondence between the number of cells generated and the total planning time. In Example 9, for instance, our planner generated over three times the number of cells but took only 60% of the time to find a solution. For the twelve problems BH averaged about 6 times faster than A* (h=0), which is probably comparable to BLP2.

We compare the total number of cells generated by our planner to the estimates given in [49] for the total number of cells generated by ZL using an octree representation, for four of the twelve problems shown in Fig. 9. Their estimates for the total number of cells generated in the course of planning far exceed the number of cells generated by our planner. That our planner can contain the explosion in the number of cells that typifies planners based on octrees [7], [49], [13], [14] illustrates the tight control over the search process exhibited by our planner. This is illustrated in Table III, an extension of the table presented in [49].

TABLE III
ZHU AND LATOMBE PLANNER USING OCTREES VERSUS OUR PLANNER

example	total cells	
	ZL	BH
2	> 500	349
6	> 2000	487
11	> 5000	451
12	> 5000	287

Finally, we can also look at the average time spent per vertex, estimated from Table II. This shows ZL and BLP2 spend roughly twice as much time per vertex as our planner. This is especially important in the context of the quality of solutions generated. While the example solutions reported in the literature are certainly of very good quality, they are not optimal in the sense of least cost paths, as is our planner. It may be that for some cost function, the solutions reported in the literature are optimal, however there is no way to guarantee this performance because those planners rely on the bridge the gap strategy.

D. Additional Examples

In Figs. 12-14 and Table IV we present a few more examples and results from our planner. The environments were designed with the intention of making our planner work very hard. Long convoluted paths and many distributed gaps that are nearly or just barely wide enough for the robot to pass through should at least force the planner to consider many reasonable alternatives at each iteration. As can be seen in the table, we were partially successful at this endeavor.

In the figures, all robots are single polygons except for Examples 4, 5, 6, and 9. The robot in Example 4 is a pair

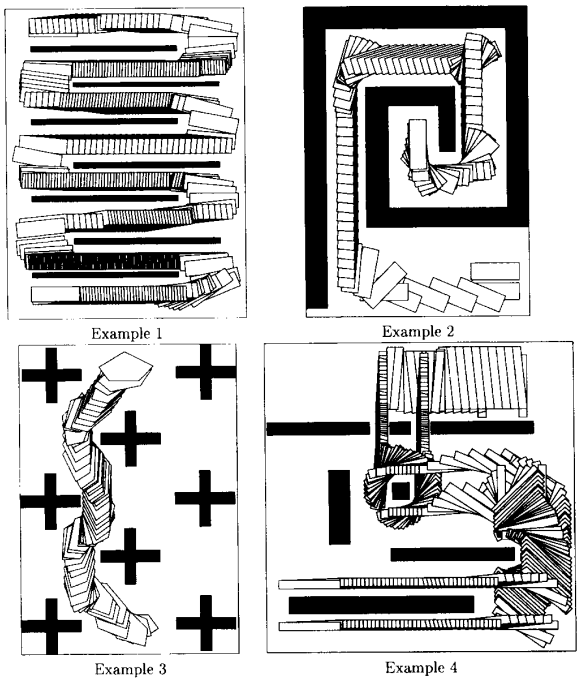


Fig. 12. First four extra environments.

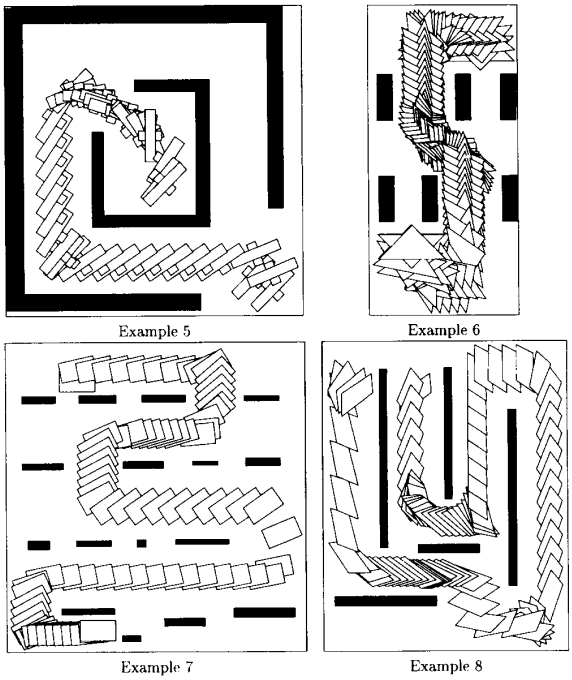


Fig. 13. Second four extra environments.

of parallel rectangles, the robot in Example 5 is a rectangle with a small cross piece, and the robot in Examples 6 and 9 is a pair of overlapping triangles. Note the asymmetry in the trajectory in Example 1. This is a side effect produced

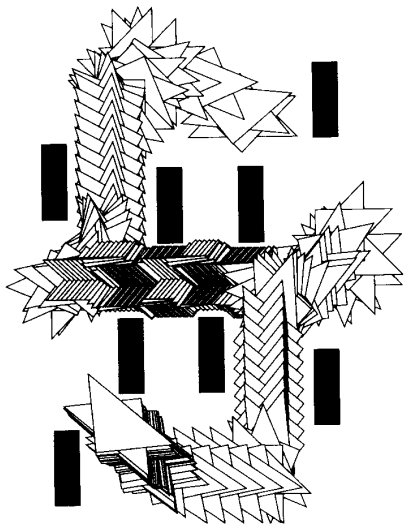


Fig. 14. Ninth extra environment.

TABLE IV
SUMMARY OF EXTRA PROBLEMS

example	plan time (min) BH	plan time (min) A*	total cells	empty cells	channel length
1	621.0	2316.1	3928	1088	446
2	7.0	43.8	666	216	98
3	8.2	34.2	515	167	62
4	56.3	284.0	1618	378	177
5	4.1	17.8	378	97	42
6	5.4	29.9	523	187	68
7	6.4	37.9	614	169	79
8	11.3	61.8	800	203	91
9	34.1	180.6	1322	407	158

by the octree decomposition. Again, the trajectories given are not optimized in any way so as to illustrate the underlying partition of $\mathcal{C}$.

VII. DISCUSSION

Hierarchical Approximate Cell Decomposition (ACD) methods have a computational advantage over the Exact methods. This advantage stems from not having to explicitly reason about interactions among constraints, and not having to explicitly reason about the full detail of the entire configuration space. Therefore, for problems without very high degrees of freedom, ACD methods are often the method of choice. For problems with high degrees of freedom, a binary subdivision method is preferred over a 2^d-tree. This avoids generating extraneous cells and reduces the amount of memory dedicated to representing configuration space.

Our solution is optimal in the sense of least cost path in a given subdivision. This solution may not be optimal in the sense of the shortest path that exists at resolution. In order for our algorithm to generate the shortest path, an appropriate cost function must be used.

The computational bounds presented in this paper are worst case. Further improvement may be made through the use of Fibonacci heaps [28], for example. It might be possible to employ dynamic trees [46] to maintain the set of paths in the tree, so that the maximal cost mixed cell on the least cost path is also immediately available at each iteration. This information could also be used to speed up the termination predicate. As mentioned earlier, the algorithm might benefit from alternative approaches, such as maintaining a set of good separators in the graph [26], [21], [12]. We should also note that it should be fairly straightforward to generalize our results to changing edge weights instead of vertex costs, as is done in [41], [42].

Donald argues that ACD methods fail to take advantage of the "coherence" of configuration space [19]. He specifically addresses the imprecision due to the simple labeling scheme (MIXED, EMPTY, and FULL) used in the representation. Our volume estimates and cost function overcomes this specific lack of coherence. Another form of coherence appears in the FindPath algorithm. The initial implementations of the FindPath algorithm either suffered from uncorrelated search efforts, or the inability to guarantee the quality of the solution and the rate of convergence of the search algorithm. Our improved FindPath algorithm (shown in Fig. 3) overcomes these difficulties by efficiently maintaining the shortest path between two points, as better approximations are obtained.

There are at least three important directions for future research. Two directions are aimed at further work reductions in the FindPath algorithm. The third direction attempts to show that our improved FindPath algorithm actually performs only a constant amount of work to maintain the shortest paths tree.

Our improved FindPath algorithm maintains the least cost path from the source to every vertex in the connectivity graph. At successive iterations of the planning algorithm, the least cost path might alternate among multiple paths until an E-path is found. If the costs of the various paths only differ by ϵ, these alternations may entail much effort to maintain the tree, with small returns. Instead, we could require that the change in path cost exceed ϵ before changes to the single-source shortest paths tree are made. This would allow the planner to concentrate its attention on a single potential path, with the guarantee that the cost of the final solution path would be no more than $(1+\epsilon)$ times the cost of the least cost path. A similar result is obtained in [35] and mentioned in [12].

Another way to avoid extra work in the FindPath algorithm is to monitor the quality of the least cost path, where quality is related to the vertex costs. At some point, the least cost path may be based on MIXED cells of very low cost. Because our cost function has a heavy penalty based on the relative volume of the cell occupied by C-obstacles, and because C-obstacles are compact, a low cost cell has an excellent chance of containing a collision-free trajectory. Rather than continue to iterate, subdividing a single cell at a time, and updating the shortest paths tree, a fast collision detection algorithm may quickly test whether the current M-path does in fact contain a solution trajectory. Thus with a minimal amount of overhead, we may be able to cut the number of iterations of the FindPath algorithm in half.

Our dynamic algorithm is based on the idea that the connectivity graph changes nominally at each iteration of the FindPath algorithm, with minimal impact on the shortest paths tree. Occasionally, this requires a great deal of computation. It is possible that an amortized analysis will show that over the course of planning, our improved FindPath algorithm performs a constant amount of work to maintain the shortest paths tree at each iteration. An extension of this analysis to the number of constraints that are applicable to a given cell might also show that our improved FindPath algorithm takes only constant time for an entire iteration on average.

VIII. CONCLUSIONS

In this paper we introduced the use of a single-source shortest paths tree to maintain the least cost path $\langle v_{init} \ldots v_{goal} \rangle$, thereby avoiding the exhibited redundancy of repeated A* search, without sacrificing the quality of the solution or the convergence properties of the FindPath algorithm. We presented two new algorithms for dynamically maintaining a single-source shortest paths tree efficiently. Our results show that on average, the dynamic algorithms do very well compared to the static Dijkstra's algorithm. We compared the resulting planner to previous planners. Most previously reported results are given in terms that are highly implementation and machine specific (*e.g.* execution times). Our results show that the improvements to the FindPath algorithm can be significant. Note that we have only improved the search aspect of hierarchical motion planning. Further improvements can be made through an improved representation of configuration space and improved cell subdivision and labeling, as evidenced by [49].

APPENDIX

The following lemmas are well-known in the literature and are stated here without proof. Also, we remind the reader that $\pi_{\mathcal{G}'}(v_0, v)$ denotes the unique simple path in the tree $\mathcal{G}'$ from the root v_0 to the vertex v; and $\Pi^*_{\mathcal{G}}(v_0, v)$ denotes the set of all simple least cost paths from v_0 to v in $\mathcal{G}$.

Lemma 9 [Ancestor]: For $\pi' = \langle u \ldots w \rangle$ a proper subpath of some simple path $\pi = \langle u \ldots w \ldots v \rangle$, pathcost($\pi$) > pathcost($\pi'$).

Lemma 10 [Least Cost Paths]: Subpaths of least cost paths are themselves least cost paths:

Theorem 3 [Independence]: A vertex $u \neq v$ is independent of v for *all* changes to cost (v) if

$$\text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(u))) \leq \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(v))). \quad (3)$$

Proof:

We must show that the two independence conditions hold for every vertex that satisfies the claim.

1) pathcost($\pi_{\mathcal{G}'_{i+1}}(v_0, u)$) = pathcost($\pi_{\mathcal{G}'_i}(v_0, u)$).
   Since $\mathcal{G}'_{i+1}$ is structurally the same as $\mathcal{G}'_i$, if $v \notin \pi_{\mathcal{G}'_i}(v_0, u)$ then $v \notin \pi_{\mathcal{G}'_{i+1}}(v_0, u)$. It then follows immediately that pathcost($\pi_{\mathcal{G}'_{i+1}}(v_0, u)$) = pathcost($\pi_{\mathcal{G}'_i}(v_0, u)$) since v is the only vertex

whose cost is not the same in $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$. We need only therefore show that $v \notin \pi_{\mathcal{G}'_i}(v_0, u)$ for all u that satisfy the inequality given by (3). We show this by contradiction. Suppose $v \in \pi_{\mathcal{G}'_i}(v_0, u)$. Then v is an ancestor of u and by the ancestor lemma $\text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, v)) < \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, u))$. Furthermore, by applying the ancestor lemma to the parents of u and v, we obtain

$$\text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(v))) < \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(u)))$$

which is a contradiction of (3).

2) $\pi_{\mathcal{G}'_{i+1}}(v_0, u) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$. We prove this in the following three steps.

(2a) We show that since the structure of $\mathcal{G}_{i+1}$ is the same as $\mathcal{G}_i$, $\Pi^*_{\mathcal{G}_{i+1}}(v_0, v) = \Pi^*_{\mathcal{G}_i}(v_0, v)$ where v is the vertex whose cost has changed.

(2b) We then show that the parent of v satisfies $\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v)) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))$.

(2c) We then show that any other vertex $u \neq v$ that satisfies the inequality given by (3), satisfies $\pi_{\mathcal{G}'_{i+1}}(v_0, u) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$.

a) Consider the set $\Pi_{\mathcal{G}_i}(v_0, v)$ of all simple paths from v_0 to v in $\mathcal{G}_i$, sorted by path cost. The path cost of every path $\pi = \langle v_0 \ldots p. v\rangle \in \Pi_{\mathcal{G}_i}(v_0, v)$ is given by cost (v) + pathcost($\langle v_0 \ldots p\rangle$). Changing cost (v) by some amount uniformly changes the path cost of every path in $\Pi_{\mathcal{G}_{i+1}}(v_0, v)$ by that amount (since these are simple paths, v occurs exactly once in each such path). Therefore, $\Pi_{\mathcal{G}_{i+1}}(v_0, v) = \Pi_{\mathcal{G}_i}(v_0, v)$ and the order of the paths in $\Pi_{\mathcal{G}_{i+1}}(v_0, v)$, sorted by path cost, is the same as in $\Pi_{\mathcal{G}_i}(v_0, v)$. Therefore $\Pi^*_{\mathcal{G}_{i+1}}(v_0, v) = \Pi^*_{\mathcal{G}_i}(v_0, v)$.

b) Now consider the path $\pi_{\mathcal{G}'_{i+1}}(v_0, v) = \pi_{\mathcal{G}'_i}(v_0, v)$. Since $\Pi^*_{\mathcal{G}_{i+1}}(v_0, v) = \Pi^*_{\mathcal{G}_i}(v_0, v)$ and $\pi_{\mathcal{G}'_i}(v_0, v) \in \Pi^*_{\mathcal{G}_i}(v_0, v)$, then $\pi_{\mathcal{G}'_{i+1}}(v_0, v) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, v)$. Then $\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v)) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))$ because subpaths of least cost paths are themselves least cost paths by the least cost paths lemma.

c) Finally, consider a vertex $u \neq v$ such that the inequality given by (3) holds. We show that $\pi_{\mathcal{G}'_{i+1}}(v_0, u) \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$ by contradiction. Assume that $\pi_{\mathcal{G}'_{i+1}}(v_0, u) \notin \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$. Then there is some $\pi \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$ such that

$$\text{pathcost}(\pi) < \text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, u)).$$

There was no such $\pi \in \Pi^*_{\mathcal{G}_i}(v_0, u)$ because $\mathcal{G}'_i$ is a single-source shortest paths tree and $\pi_{\mathcal{G}'_i}(v_0, u) \in \Pi^*_{\mathcal{G}_i}(v_0, u)$. Since only the cost of v has changed, any such $\pi \in \Pi^*_{\mathcal{G}_{i+1}}(v_0, u)$ must include v. For if $v \notin \pi$, then pathcost(π) has not changed from $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$ and the claim that pathcost(π) < pathcost($\pi_{\mathcal{G}'_{i+1}}(v_0, u)$) = pathcost($\pi_{\mathcal{G}'_i}(v_0, u)$) is contradicted. Then π has the form

$$\pi = \langle v_0 \ldots \text{parent}_{\mathcal{G}'_{i+1}}(v), v \ldots u\rangle.$$

Since π is a least cost path, all subpaths of π must be least cost paths by the least cost path lemma. In particular, the subpath $\pi_1 = \langle v_0 \ldots \text{parent}_{\mathcal{G}'_{i+1}}(v)\rangle$ is a least cost path. But $\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))$ is also a least cost path from statement (2b) above. Therefore pathcost(π_1) = pathcost($\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))$).

Now, the path cost of π can be written as the sum of the costs of its subpaths.

$$\text{pathcost}(\pi) = \text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))) + \text{cost}(v) + \text{cost}(u) + \gamma$$

where $\gamma \geq 0$ is the cost of the least cost path connecting v to u. Since π is a least cost path, pathcost(π) $\leq$ pathcost($\pi_{\mathcal{G}'_{i+1}}(v_0, u)$). We also know that

$$\text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, u)) = \text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(u))) + \text{cost}(u).$$

Finally, we know that both

$$\text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(v))) = \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(v)))$$

and

$$\text{pathcost}(\pi_{\mathcal{G}'_{i+1}}(v_0, \text{parent}_{\mathcal{G}'_{i+1}}(u))) = \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(u)))$$

by statement (1) of the proof, above. Therefore, since vertex costs are positively valued, by simple algebraic manipulation we have:

$$\text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(v))) = \text{pathcost}(\pi_{\mathcal{G}'_i}(v_0, \text{parent}_{\mathcal{G}'_i}(u)))$$

which is a contradiction of (3). □

REFERENCES

[1] D. Adler, "Switch-level simulation using dynamic graph algorithms," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 3, pp. 346-355, Mar. 1991.

[2] P. K. Agarwal, D. Eppstein, and J. Matouvek, "Dynamic algorithms for half-space reporting, proximity problems, geometric minimum spanning trees," in *Proc. 33rd IEEE Symp. Foundations of Comput. Sci.*, 1992, pp. 80-89.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

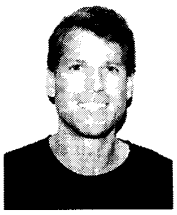
[4] R. Ahuja and J. Orlin, "The scaling network simplex algorithm," *OR*, vol. 40, pp. 1, pp. S5–S12, Jan.-Feb. 1992.

[5] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F. Zadeck, "Incremental evaluation of computational circuits," in *Proc. 1st Annu. ACM-SIAM Symp. on Discrete Algs*, 1990.

[6] G. Ausiello, G. F. Italiano, A. Maretti-Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," *J. Algorithms*, vol. 12, pp. 615-638, 1991.

[7] F. Avnaim, J. D. Boissonnat, and B. Faverjon, "A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles," in *IEEE Int. Conf. Robot. and Automat.*, 1988.

[8] M. Barbehenn, "Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees," Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Jan. 1993.
[9] M. Barbehenn and S. Hutchinson, "Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees," Technical Report UIUC-BI-AI-RCV-93-04, Beckman Institute, University of Illinois, Apr. 1993.
[10] ——, "Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees," in *IEEE Int. Conf. Robot. and Automat.*, 1993.
[11] J. Barraquand and J. C. Latombe, "A Monte-Carlo algorithm for path planning with many degrees of freedom," in *IEEE Int. Conf. Robot. and Automat.*, 1990, pp. 1712-1717.
[12] D. P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
[13] R. Brooks and T. Lozano-Perez, "A subdivision algorithm in configuration space for findpath with rotation," Technical Report AIM 684, MIT AI Laboratory, 1982.
[14] ——, "A subdivision algorithm in configuration space for findpath with rotation," in *Proc. Int. Joint Conf. on Art. Intell.*, pp. 799-806, 1983.
[15] J. F. Canny, *The Complexity of Robot Motion Planning.* Cambridge, MA: MIT Press, 1988.
[16] P. Chen and Y. Hwang, "Practical path planning among movable obstacles," in *Proc. IEEE Int'l Conf. on Robot. and Automat.*, 1991.
[17] T. H. Corman, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.
[18] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.
[19] B. R. Donald, "Motion planning with six degrees of freedom," Technical Report AI-TR-791, MIT AI Lab, 1984.
[20] H. Edelsbrunner, *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.
[21] D. Eppstein, "Dynamic Euclidean minimum spanning trees and extrema of binary functions," to appear in *Discrete and Comp. Geometry*, 1994.
[22] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Oper. Res.*, vol. 49, pp. 371-387, 1985.
[23] B. Faverjon, Obstacle avoidance using an octree in the configuration space of a manipulator," in *IEEE Int. Conf. Robot. and Automat.*, 1984, pp. 504-512, Atlanta.
[24] ——, "Object level programming of industrial robots," in *IEEE Int. Conf. Robot. and Automat.*, 1986, pp. 1406-1412.
[25] E. Feuerstein and A. Maretti-Spaccamela, "Dynamic algorithms for shortest paths in planar graphs," in *Proc. 17th Int. Workshop on Graph-Theoretic Concepts in Comput. Sci.*, 1991. (Also appears in LNCS 570, Springer-Verlag, Berlin.)
[26] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM J. Comput.*, vol. 14, no. 4, pp. 781-798, Nov. 1985.
[27] G. N. Frederickson, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees," unpublished, Jan. 1992.
[28] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596-615, 1987.
[29] K. Fujimura and H. Samet, "Hierarchical strategy for path planning among moving obstacles," *IEEE Trans. Robot. and Automat.*, Feb. 1989.
[30] D. Goldfarb, J. Hao, and S. Kai, "Efficient shortest path simplex algorithms," *Oper. Res.*, vol. 38, no. 4, pp. 624-628, Jul-Aug 1990.
[31] V. Hayward, "Fast collision detection scheme by recursive decomposition of a manipulator workspace," in *IEEE Int. Conf. on Robot. and Automat.*, 1986, pages 1044-1049.
[32] Y. K. Hwang and N. Ahuja, "Path planning using a potential field representation," Technical Report UIUC-ENG-88-2251, University of Illinois at Urbana-Champaign, 1988.
[33] S. Kambhampati and L. Davis, "Multiresolution path planning for mobile robots" *IEEE J. Robot. and Automat.*, vol. 2, no. 3, pp. 135-145, Sept. 1986.
[34] O. Khatib, "Real time obstacle avoidance for manipulators and mobile robots," *Int. J. Robot. Res.*, vol. 5, no. 1, pp. 90-96, 1986.
[35] P. Klein and S. Sairam, "A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs," in *Proc. 3rd Workshop on Algs. and Data Struct.*, 1993. (Also appears in LNCS 709, Springer-Verlag, Berlin.)
[36] J. C. Latombe, *Robot Motion Planning.* Boston: Kluwer Academic Publishers, 1991.
[37] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids.* New York: Holt, Rinehart and Winston, 1976.
[38] C. C. Lin and R. C. Chang, "On the dynamic shortest path problem," *J. Informat. Proc.*, vol. 13, no. 4, pp. 470-476, 1990.
[39] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE Trans. Comput.*, Feb. 1983.
[40] F. P. Preparata and M. I. Shamos, *Computational Geometry.* New York: Springer-Verlag, 1985.
[41] G. Ramalingam and T. Reps, "On the computational complexity of incremental algorithms," Technical Report 1033, Computer Science Department, University of Wisconsin, Madison, Aug. 1991.
[42] ——, "An incremental algorithm for a generalization of the shortest-path problem," Technical Report 1087, Computer Science Department, University of Wisconsin, Madison, May 1992.
[43] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential functions," *IEEE J. Robot. and Automat.*, vol. 8, no. 5, pp. 501-518, Oct. 1992.
[44] H. Rohnert, "A dynamization of the all pairs least cost path problem," in *Proc. 2nd Annu. Symp. Theoretical Aspects of Comput. Sci.*, 1985, pp. 279-286. Also appears in *Lecture Notes in Computer Science 182.* Berlin: Springer-Verlag, 1985.
[45] J. T. Schwartz and M. Sharir. "On the piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds." in *Planning, Geometry, and Complexity of Robot Motion*, J. T. Schwartz, M. Sharir, and J. Hopcroft, Eds. Norwood, NJ: Ablex, 1987, pp. 51-96.
[46] D. Sleator and R. Tarjan, "A data structure for dynamic trees," *J. Comput. and Syst. Sci.*, vol. 26, pp. 362-391, 1992.
[47] R. Spence and S. A. Hutchinson, "Dealing with unexpected moving obstacles by integrating potential field planning with inverse dynamics control," in *Proc. IEEE Int. Conf. Intell. Robot. and Syst.*, 1992, pp. 1485-1490.
[48] P. M. Spira and A. Pan, "On finding and updating spanning trees and shortest paths," *SIAM J. Comp*, vol. 4, pp. 375-380, 1975.
[49] D. Zhu and J.-C. Latombe, "New heuristic algorithms for efficient hierarchical path planning," *IEEE Trans. Robot. and Automat.*, vol. 7, no. 1, pp. 9-20, Feb. 1991.



Michael Barbehenn (S'91) received a B.A. in Mathematics and Computer Science from Cornell University in 1987. He received an M.S. in Computer Science from the University of Illinois at Urbana-Champaign in 1993. He is currently completing a Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign.

His current research interests include robot motion planning in dynamic environments, sensor-based robot motion planning, machine learning for robot navigation, and incremental graph algorithms.

Mr. Barbehenn is a Student Member of IEEE.

Seth Hutchinson (S'85-M'88) received his Ph. D. degree from Purdue University in West Lafayette, Indiana in 1988.

He spent 1989 as a Visiting Assistant Professor of Electrical Engineering at Purdue University. In 1990 Dr. Hutchinson joined the faculty at the University of Illinois in Urbana-Champaign, where he is currently an Assistant Professor in the Department of Electrical and Computer Engineering, and the Beckman Institute for Advanced Science and Technology. Professor Hutchinson's current research interests include: integration of vision, force and position sensing for robot motion planning and control; dynamic planning of sensing strategies; constraint based reasoning; task planning for automated assembly; evidential reasoning applied to model based object recognition; and sensor integration.

Dr. Hutchinson is a Member of IEEE.