Efficient Search and Hierarchical Motion Planning Using Dynamic Single-Source Shortest Paths Trees

Michael Barbehenn   Seth Hutchinson*
Artificial Intelligence Group
The Beckman Institute for Advanced Science and Technology
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

In this paper we examine the search aspects of hierarchical motion planning. All previous robot motion planners based on approximate cell decomposition exibit redundancy between successive searches for a sequence of adjacent EMPTY *cells. In this paper we present a search method that eliminates this redundancy. Our search method is founded on the ability to efficiently maintain a single-source shortest paths tree embedded in the connectivity graph that is subject to the dynamic modifications that result from incremental subdivision of cells. The convergence of our algorithm is controlled by the vertex cost function, which relies on an estimate for the proportion of free space in a cell. The planner is fully implemented and we give empirical results to illustrate the performance improvements of the dynamic algorithm compared to Dijkstra's algorithm.*

1 Introduction

This paper addresses hierarchical motion planning using approximate cell decomposition. We examine the problem of a polygonal robot amid polygonal obstacles in the plane. We reduce this problem to that of finding a path in the three dimensional configuration space of the robot, $\mathcal{C} = \mathbf{R}^2 \times S^1$. All configurations $q \in \mathcal{C}$ for which the robot intersects some obstacle belong to the set of configuration space obstacles, denoted by $\mathcal{CB}$. For all other configurations, the robot is in free space, denoted by $\mathcal{C}_{free}$. 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 $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, such that $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$.

We use a hierarchical subdivision algorithm to partition $\mathcal{C}$ into rectangloid cells, $\kappa_i = [x'_i, x''_i] \times [y'_i, y''_i] \times [\theta'_i, \theta''_i] \subseteq \mathbf{R}^2 \times S^1$. We denote a partition of $\mathcal{C}$ as $\mathcal{P}$, and the subdivision of a cell κ as $\mathcal{P}^\kappa$ following [8]. Each cell is labeled EMPTY, FULL, or MIXED depending on whether it is completely contained in $\mathcal{C}_{free}$, completely contained in $\mathcal{CB}$, or not known to be completely contained in either $\mathcal{C}_{free}$ or $\mathcal{CB}$, respectively. The subdivision algorithm is approximate because MIXED cells below some user-specified resolution are considered to be FULL [3, 5, 7, 6, 11, 8]. We use the respresentation of $\mathcal{CB}$ from [9, 3] and the cell labeling algorithm given in [3].

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 [3, 5, 7, 6, 8].

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 non-reflexive 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 (resp. E-channel) as an M-path (resp. E-path). Let v_{init} and v_{goal} be the vertices associated with κ_{init} and κ_{goal} respectively. Planning is then reduced to searching $\mathcal{G}_i$ for an M-path and then selecting vertices on the M-path with MIXED cells to subdivide to obtain $\mathcal{G}_{i+1}$. The traditional FindPath algorithm is shown below.

```
procedure FindPath ($q_{init}, q_{goal}$:config; $\mathcal{G}_0$:graph)
[1]  $i \leftarrow 0$
[2]  $\pi \leftarrow$ an M-path from $v_{init}$ to $v_{goal}$ in $\mathcal{G}_i$
[3]  until $\pi$ is an E-path do
[4]       select vertices $\{v_i\} \subseteq \pi$
[5]       subdivide cells $\{\kappa_i\}$ and construct $\mathcal{G}_{i+1}$
[6]       $i \leftarrow i + 1$
[7]       $\pi \leftarrow$ an M-path from $v_{init}$ to $v_{goal}$ in $\mathcal{G}_i$
[8]  return $\pi$
end
```

In this paper we show how the performance of the traditional FindPath algorithm can be significantly im-

*This work was supported by the National Science Foundation under grant number NSF-IRI-9110270

proved 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 search for an M-path in Line 7 to a local tree update. In contrast, previous implementations use a global search at each iteration.

The remainder of the paper is organized as follows. In Section 2, we characterize the FindPath search space and present a more efficient search algorithm based on dynamically maintaining a single-source shortest paths tree. In Section 3 we introduce the terminology and concepts underlying our solution. Then in Section 4 we present a greedy solution to the dynamic single-source shortest paths problem. Section 5 reports empirical results showing the improvement of the dynamic algorithm over the traditional approach. Finally, Section 6 gives our conclusions.

2 FindPath Search

The search for a collision-free path in $\mathcal{C}$ is carried out by finding an E-channel in a partition $\mathcal{P}$ of $\mathcal{C}$. The search for an E-channel in $\mathcal{P}$ is equivalent to searching the corresponding connectivity graph $\mathcal{G}$ for an E-path from v_{init} to v_{goal}. The FindPath algorithm performs this search by finding 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.

Before we discuss in detail the search space of the FindPath algorithm, we present the basic terminology that will be used throughout this paper to discuss paths in graphs.

Definition 1 *The* path cost *of a path π is given by,*

$$\text{pathcost}(\pi) = \begin{cases} \sum_{v\in\pi} \text{cost}(v) & \text{if } \pi \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

Definition 2 *Denote by $\Pi_{\mathcal{G}}(u,v)$ the set of all simple paths in $\mathcal{G}$ from u to v.*

Definition 3 *The* least path cost *from u to v in $\mathcal{G}$ is given by* $\min_{\pi\in\Pi_{\mathcal{G}}(u,v)} \text{pathcost}(\pi)$.

Definition 4 *Denote by $\Pi^*_{\mathcal{G}}(u,v) \subseteq \Pi_{\mathcal{G}}(u,v)$ the set of all simple least cost paths in $\mathcal{G}$ from u to v.*

Note that $\Pi_{\mathcal{G}}(u,v) = \emptyset$ if and only if $\Pi^*_{\mathcal{G}}(u,v) = \emptyset$. If there is any path $\pi = \langle u \dots v\rangle$ then there is a least cost path; if there is no least cost path, then there can be no path at all.

2.1 Finding a Solution Graph

Assume that $\mathcal{C}$ is enclosed in a cell κ_0. A given deterministic subdivision algorithm uniquely partitions κ_0 into subcells. We will restrict the problem by only allowing MIXED cells to be subdivided.

Definition 5 *Let $\mathcal{AP}$ be the space of all partitions that can be obtained by performing a sequence of subdivisions of* MIXED *cells.*

Definition 6 *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, the lattice is finite. The least upper bound of the lattice is the initial partition $\mathcal{P}_0$ that contains only one 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 $\mathcal{G}$. Thus the space of partitions $\mathcal{AP}$ defines a space of connectivity graphs which we denote by $\mathcal{AG}$. The bijective map between $\mathcal{AP}$ and $\mathcal{AG}$ implies that $\mathcal{AG}$ is also a lattice.

Definition 7 *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, we would want the FindPath algorithm 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. We express this more formally in the following lemma.

Lemma 1 *Let $\mathcal{G} \in \mathcal{SG}$, and let $G = \{\mathcal{G}_i \in \mathcal{AG} \mid \mathcal{P}_i \preceq \mathcal{P}\}$. Then $G \subseteq \mathcal{SG}$.*[1]

Hill-climbing is well-suited for this search space because $\mathcal{AG}$ is a lattice and all subdivisions eventually lead to $\mathcal{G}_\infty$, which is a recognizable[2] solution graph if $\mathcal{SG} \neq \emptyset$ (since there are only EMPTY cells in $\mathcal{G}_\infty$). Also, by Lemma 1, once a hill-climbing search is within $\mathcal{SG}$, it can never leave $\mathcal{SG}$.

The search performed by the FindPath algorithm can be characterized by how it searches $\mathcal{AG}$. We have identified three distinct heuristics in the FindPath algorithm as follows.

1. The selection of a particular M-path on Line 7 of the FindPath algorithm serves to restrict subdivision to cells in a particular M-channel. We refer to this restriction as the *channel heuristic*. This heuristic reflects the belief that the cells in the particular M-path from v_{init} to v_{goal} in $\mathcal{G}_i$ can be subdivided to produce an E-path from v_{init} to v_{goal}.
2. The *vertex cost heuristic* is a function that assigns a cost to each vertex. The function should reflect the

[1]Proofs of all lemmas in this paper can be found in [2].

[2]Due to the choice of M-path on Line 7 of the FindPath algorithm, a given graph may or may not be recognized as a solution graph.

odds of finding a collision-free trajectory in the corresponding cell. The desired properties of a good cost function are (1) rapid growth for small $\rho(v)$, (2) a preference for larger cells over smaller cells, (3) positive values, and (4) continuously valued, where $\rho(v)$ measures the proportion of $\mathcal{C}_{free}$ in the cell associated with v. This function is described in more detail in [2]. Vertex cost defines which M-path is the one of least cost at any iteration of the FindPath algorithm.

3. The *cell selection heuristic* is used to identify which MIXED cells to subdivide on Line 4 of the FindPath algorithm. Selecting the maximum cost vertex with a MIXED cell in the *least cost* path $\langle v_{init} \dots v_{goal} \rangle$ forces the planner to give precedence to bottlenecks. This causes the planner to work out the most difficult constraints on the solution trajectory first, the remaining constraints, prioritized by the size of the bottleneck presented, are incrementally easier to satisfy.

These three heuristics combine to form a single *vertex selection heuristic* that is used by FindPath to expand a node in the search. In some sense, the channel heuristic and cell selection heuristic can be viewed as fixed, and the vertex cost function can be viewed as the crucial parameter that controls the search.

2.2 Finding an M-Path

There have been two methods for finding an M-path at each iteration of the FindPath algorithm: A* search, and what we call the bridge the gap strategy. The bridge the gap strategy was developed by [3] and embellished by [7, 11] to avoid the work of repeated A* search.

The bridge the gap strategy can be explained as follows. Consider the vertex v_s on a path in $\mathcal{G}_i$ with a MIXED cell $\kappa_s \in \mathcal{P}_i$. The M-path has the form

$$\pi_i = \langle v_{init} \dots v_{s-1}, v_s, v_{s+1} \dots v_{goal} \rangle.$$

Suppose the cell κ_s is subdivided to produce $\mathcal{P}_{i+1}$. The corresponding connectivity graph $\mathcal{G}_{i+1}$ is the same as $\mathcal{G}_i$ except that vertices adjacent to v_s in $\mathcal{G}_i$ are now adjacent to vertices $\{v_j\}$, corresponding to the non-FULL cells of $\mathcal{P}^{\kappa_s}$. The bridge the gap strategy is to form a new path π_{i+1} by connecting v_{s-1} to v_{s+1}. In the case of [7, 11], if no path $\langle v_{s-1}, v_1 \dots v_2, v_{s+1} \rangle$ exists through $\{v_j\}$, then the bridge construction fails and the search backtracks over path decisions made in previous iterations of the FindPath algorithm. In the case of [3], a local detour from v_{s-1} to v_{s+1} is attempted; and if bridge construction fails, a new path from v_{init} to v_{goal} is found from scratch. Unlike A* search, the bridge the gap strategy is not guaranteed to find the least cost path from v_{init} to v_{goal} in $\mathcal{G}_{i+1}$.

We introduce a mechanism to eliminate the exhibited redundancy of repeatedly applying A* search without sacrificing obtaining a least cost path. 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}. This path can be returned in time $\mathcal{O}(l)$, where l is the length of the path, by following the unique pointers from v_{goal} to v_0. This path is equivalent to the path found by A* search.

3 Computing $\mathcal{SP}$

In this section we introduce the background material for computing a single-source shortest paths tree, $\mathcal{SP}$, embedded in a connectivity graph, $\mathcal{G}$.

3.1 Terminology

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 [1, 4]. For our problem, we associate costs with vertices rather than edges. The reason for this is that in a connectivity graph, the cells associated with the vertices represent the physical space through which a trajectory must pass. The edges are associated with the infinitesimal boundary between two cells. We could equivalently assess the cost of each edge as the average of the costs of the vertices it connects.

Definition 8 *We say that* $\mathcal{G}'(V', E')$ is a tree embedded in $\mathcal{G}(V, E)$, rooted at v_0, *if and only if* $V' \subseteq V$ *and* $E' \subseteq E$ *are such that every vertex* $v \in V'$ *is reachable from* v_0 *by a unique, simple path in* E'*. More precisely,* $\forall v \in V', \Pi_{\mathcal{G}'}(v_0, v)$ *is a singleton set.*

Given a tree $\mathcal{G}'(V', E')$ embedded in $\mathcal{G}(V, E)$, then for a vertex $v \in V'$ we have the following relevant tree and graph definitions.

$$\begin{array}{rcl} \text{neighbors}(v) & = & \{u \in V \mid (u, v) \in E\} \\ \text{parent}(v) & = & u \in V' \text{ such that } (u, v) \in E' \\ \text{children}(v) & = & \{u \in V' \mid (v, u) \in E'\} \\ \text{siblings}(v) & = & \{u \in V' \mid u \neq v \text{ and } \exists p \in V' \\ & & \text{such that } (p, u) \in E' \text{ and } (p, v) \in E'\} \end{array}$$

proper-neighbors(v) = neighbors(v) − ({parent(v)} ∪ children(v) ∪ siblings(v)).

The relationships defined above are illustrated in Figure 1. The figure 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.

Definition 9 *Given a tree* $\mathcal{G}'(V', E')$ *embedded in a graph* $\mathcal{G}(V, E)$*, rooted at* v_0*, denote by* $\pi_{\mathcal{G}'}(v)$ *the unique simple path from the root* v_0 *to the vertex* v *in* $\mathcal{G}'$.

Denote by parent$_{\mathcal{G}'}(v)$ the parent of a vertex in $\pi_{\mathcal{G}'}(v)$.

Definition 10 *A* single-source shortest paths tree *($\mathcal{SP}$) is a tree,* $\mathcal{G}'(V', E')$ *embedded in a graph* $\mathcal{G}(V, E)$*, rooted at* v_0*, such that*

- $V' = \{v \in V \mid \Pi_{\mathcal{G}}(v_0, v) \neq \emptyset\}$.
- $\forall v \in V' \;\; \pi_{\mathcal{G}'}(v) \in \Pi^*_{\mathcal{G}}(v_0, v)$.

In other words, $\mathcal{SP}$ is a directed subgraph $\mathcal{G}'(V', E')$ of $\mathcal{G}(V, E)$ such that V' consists of exactly those vertices

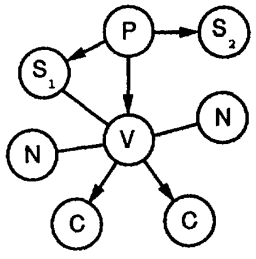

Figure 1: Vertex Relationships.

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.

Definition 11 *Given a tree $\mathcal{G}'(V', E')$ embedded in a graph $\mathcal{G}(V, E)$, rooted at v_0, a vertex $v \in V$ is* locally $\mathcal{SP}$ *(abbreviated $l\mathcal{SP}$) if and only if*

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

where p is such that $\forall\ n \in$ neighbors(v), pathcost$(\pi_{\mathcal{G}'}(p)) \leq$ pathcost$(\pi_{\mathcal{G}'}(n))$.

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

3.2 Changing Costs

Throughout this section we will use the following notation. The graph $\mathcal{G}_{i+1}$ is obtained by changing the cost of some vertex v in $\mathcal{G}_i$. Structurally, the two graphs are identical. The tree $\mathcal{G}'_i$ is embeddeded in $\mathcal{G}_i$ and is a single-source shortest paths tree. We will sometimes refer to $\mathcal{G}'_i$ as $\mathcal{SP}_i$ to emphasize this fact. The tree $\mathcal{G}'_{i+1}$ is embedded in $\mathcal{G}_{i+1}$ and is structurally the same as $\mathcal{G}'_i$ but has a different cost for v. In general $\mathcal{G}'_{i+1}$ is *not* a single-source shortest paths tree.

Definition 12 *We say that a vertex u is* independent *of v if and only if the following* independence equations *hold for all possible changes to cost(v):*

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

Definition 13 *We say that a vertex u is* relatively independent *of v for some specific set changes to cost(v), if and only if the independence equations hold for all such changes.*

Lemma 3 *A vertex u is independent of v, $v \neq u$, if* pathcost$(\pi_{\mathcal{G}'_i}(\text{parent}_{\mathcal{G}'_i}(u))) \leq$ pathcost$(\pi_{\mathcal{G}'_i}(\text{parent}_{\mathcal{G}'_i}(v)))$.

A consequence of Lemma 3 is that the parent, siblings, and some proper-neighbors of a vertex v are independent of v. The following corollary extends the lemma to the rest of the proper-neighbors.

Corollary 4 *A vertex u is relatively independent of v, $u \neq v$ and* pathcost$(\pi_{\mathcal{G}'_i}(\text{parent}_{\mathcal{G}'_i}(u))) \leq$ pathcost$(\pi_{\mathcal{G}'_i}(v))$, *for those changes δ in cost(v) such that* pathcost$(\pi_{\mathcal{G}'_i}(\text{parent}_{\mathcal{G}'_i}(u))) \leq$ pathcost$(\pi_{\mathcal{G}'_i}(v)) + \delta$.

Lemma 5 *A vertex u that is relatively independent of v in $\mathcal{G}_i$ for the given change, is $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.*

It follows that a vertex u that is independent of v in $\mathcal{G}_i$, is $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$.

We have now established a local sufficiency condition for vertex independence and hence a local condition for whether a vertex is $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. We have not identified *all* independent vertices, however. Furthermore, we have not yet identified the dependent vertices nor indicated whether they are $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Knowing that a vertex u is not relatively independent of v for a given change to cost(v) is not sufficient information to show whether u is either $l\mathcal{SP}$ or not $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Instead, we examine two complementary subsets of V': descendents and non-descendents, which are denoted by $D(v)$ and $V' - D(v)$ respectively, subject to changing cost(v).

Lemma 6 *The descendents of v are $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ for all decreases to cost(v).*

Lemma 7 *A proper-neighbor n of v is not $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ if cost(v) decreases by more than* pathcost$(\pi_{\mathcal{G}'_i}(v)) -$ pathcost$(\pi_{\mathcal{G}'_i}(\text{parent}_{\mathcal{G}'_i}(n)))$.

Lemma 8 *The non-descendents of v are relatively independent of v for all increases in cost(v).*

A direct consequence of Lemma 8 and Corollary 4 is that non-descendents are $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ when a vertex cost increases. Knowing that a vertex is a descendent is insufficient to show whether the vertex is $l\mathcal{SP}$ or not $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$ for an increase in cost(v). Furthermore, we know of no precise local conditions.

The preceding lemmas identify large, easily recognized subsets of vertices that are already $l\mathcal{SP}$ after a given change to the graph and therefore need no further processing. In the case of a decrease to cost(v), Lemma 7 gives a local criterion for identifying those vertices that are no longer $l\mathcal{SP}$ in $\mathcal{G}_{i+1}$. Lemma 7 also suggests that the local repair is also the globally correct repair. Such repairs can be propagated outward from the source of the change. However, the above lemmas do not sanction any specific action on the part of an algorithm to construct $\mathcal{SP}_{i+1}$ for an increase in cost(v), they only restrict our attention to the set $D(v)$. Using a local decision criterion to propage arbitrary changes the rest of the graph, and in particular to assign a new parent to a vertex whose initial parent in $\mathcal{G}'_{i+1}$ has increased its path cost, may lead to an unbounded algorithm. Cycles of arbitrary length can form and the algorithm will loop until the path cost of the cycle becomes large enough for a neighbor of some vertex in the cycle to provide a lower cost path.

3.3 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 (interconnected) 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 parent$_{\mathcal{G}'_i}(v_s)$ has one less child in $\mathcal{G}'_{i+1}$. Also, each newly created vertex has no parent in $\mathcal{G}'_{i+1}$.

The graph is repaired and a new single-source shortest paths tree is computed by the algorithm given in the next section.

4 A Dynamic Algorithm

We identified in Section 3 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 algorithm given in this section can be seen to be a direct implementation of Lemmas 3-8, with a greedy local criterion for the case of a parent increasing its path cost. Split-Vertex is the top-level function to be called from FindPath; the other procedures are styled after [4] to emphasize the similarity to Dijkstra's algorithm [1, 4].

This algorithm takes advantage of the sparseness of the graph. Therefore we give the following lemma for octrees, the spatial decomposition used in this paper.

Lemma 9 *The connectivity graph $\mathcal{G}(V, E)$ of an octree is sparse. Specifically $|E| < \frac{30}{7}|V|$.*

In the pseudocode for dynamically constructing $\mathcal{SP}_{i+1}$ which follows, Divide-Vertex is responsible for deleting the vertex and associated edges from $\mathcal{G}_i$ and $\mathcal{SP}_i$, subdividing the cell associated with the vertex, labeling the new cells, and creating and adding the new vertices to construct $\mathcal{G}_{i+1}$. Divide-Vertex returns the set of newly created vertices. Min-Neighbor returns the neighboring vertex with the minimum path cost (the posted least cost path to the source). BuildPQ, UpdatePriority, Insert, and DeleteMin are the standard priority queue functions [1, 4], modified to mark and unmark vertices as they are inserted and deleted from the priority queue. This modification allows for an $\mathcal{O}(1)$ priority queue membership test.

```
procedure Split-Vertex (v:vertex; $\mathcal{G}$:graph)
[1] $p \leftarrow v.parent$
[2] $C \leftarrow v.children$
[3] $N \leftarrow$ Divide-Vertex($v, \mathcal{G}$)
[4] Dynamic-SSSP ($p, N \cup C$)
end
```

```
procedure Dynamic-SSSP (p:vertex; new:vertex-list)
[1] Dyn-Initialize(new)
[2] $Q \leftarrow$ BuildPQ($\{p\} \cup new$)
[3] while $Q \neq \emptyset$ do
[4]     $u \leftarrow$ DeleteMin($Q$)
[5]     foreach $v \in u.neighbors$ do
[6]         Dyn-Relax($u, v, Q$)
end
```

```
procedure Dyn-Initialize (vertices:vertex-list)
[1] foreach $q \in vertices$ do
[2]     $q.parent \leftarrow$ Min-Neighbor($q$)
[3]     $q.pathcost \leftarrow q.cost + q.parent.pathcost$
end
```

```
procedure Dyn-Relax (u, v:vertex; Q:p-queue)
[1] $c \leftarrow v.cost + u.pathcost$
[2] if $c < v.pathcost$
[3] then Update-Vertex($u, v, Q$)
[4] elseif $u = v.parent$ and $v.pathcost < c$
[5]     then Update-Vertex(Min-Neighbor($v$), $v, Q$)
end
```

```
procedure Update-Vertex (u, v:vertex; Q:p-queue)
[1] $v.parent \leftarrow u$
[2] $v.pathcost \leftarrow v.cost + u.pathcost$
[3] if $v \in Q$
[4] then UpdatePriority($v, Q$)
[5] else Insert($v, Q$)
end
```

In the complexity analysis that follows, let v_s be the vertex that is subdivided, let $|v|$ be the number of vertices that are not independent of v_s, and let $|e|$ be the number of edges incident on those vertices. $|e|$ and $|v|$ are subset analogs of E and V. Dyn-Initialize and BuildPQ are each called once in Lines 1 and 2 of Dynamic-SSSP respectively. The while loop at Line 3 results in at best $\mathcal{O}(|e|)$ calls to Dyn-Relax and $\mathcal{O}(|v|)$ calls to DeleteMin.

BuildPQ takes $\mathcal{O}(n)$ time where n is the number of vertices initially inserted into the queue [4]. Min-Neighbor requires, on average, $\mathcal{O}(d)$ time where d is the average vertex degree. Dyn-Initialize calls Min-Neighbor for each vertex which is $\mathcal{O}(d^2)$ time. However, we can charge the cost of finding the minimum path cost neighbor to the loop at Line 5 of Dynamic-SSSP. Therefore the first two steps of Dynamic-SSSP require $\mathcal{O}(d)$ time. DeleteMin and Update-Vertex require $\mathcal{O}(\log n)$ time where n is size of the priority queue which is bounded by $|v|$, so Dyn-Relax takes time takes time $\mathcal{O}(\log |v|)$. If every vertex is guaranteed to be deleted from the queue at most a constant number of times, the time complexity of Dynamic-SSSP would be $\mathcal{O}(|e| \log |v|)$. As we indicated in Section 3, we cannot guarantee the local criterion employed when the parent of a vertex increases is a good choice. A poor choice simply increases the amount of time required to relax

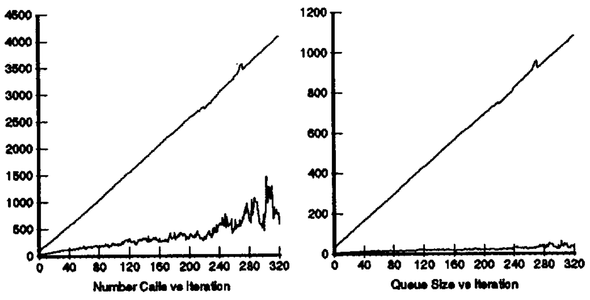

Figure 2: Number of Calls to Primitive Operations and the Growth in the Priority Queue Size.

all of the vertices into their minimum path costs. For certain vertex cost functions, this local criterion works very well, as is shown in Section 5. This bound can be guaranteed by a less greedy implementation [2, 10].

5 Results

Statistics were collected for random planning problems within environments taken from the motion planning literature [3, 11]. Since Dijkstra's algorithm is comparable to using A* search to find a least cost path, the dramatic improvement of the dynamic algorithm over Dijkstra's algorithm can be used to show the improvement of our method over previous results. Two of our results are illustrated in Figure 2.

The left side of Figure 2 shows the number of calls to the primitive priority queue operations UpdatePriority, DeleteMin, and Insert, after each subdivision. This is the crucial unit of measure of comparison since it is the basic time-consuming operation of the algorithms and is machine independent. The dynamic algorithm exhibits a slight linear trend while Dijkstra's algorithm shows linear growth in the number of calls.

The right side of Figure 2 shows the average size of the priority queue for each call to the primitive queue operations after each subdivision. The size of the priority queue affects the average time to rebalance as a result of a primitive operation, each $\mathcal{O}(\log n)$ time complexity where n is the size of the queue. A consistently smaller queue results in consistently faster overall performance. The dynamic algorithm showed near constant growth, and Dijkstra's algorithm, which inserts all vertices into the queue at every iteration, exhibited the expected linear growth. If the dynamic algorithm maintains at most a constant sized queue, then the complexity analysis for the algorithm becomes linear.

6 Conclusions

In this paper we presented a complete formulation of the FindPath search problem which has not been adequately addressed for approximate cell decomposition methods. We presented an algorithm for dynamically maintaining a single-source shortest paths tree ($\mathcal{SP}$) efficiently. The ability to maintain $\mathcal{SP}$ under dynamic modifications to the underlying connectivity graph is an important source of the time improvements presented here. Finally, we introduced an improved vertex cost function that more accurately measures the complexity of the underlying, unexplored space. This gives our search algorithm better convergence properties than other approaches.

References

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

[2] M. Barbehenn. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees. Master's thesis, University of Illinois at Urbana-Champaign Dept. of Computer Science, January 1993.

[3] R. Brooks and T. Lozano-Perez. A subdivision algorithm in configuration space for findpath with rotation. In *Proc. Int. Joint Conf. on Art. Intell.*, pages 799–806, 1983.

[4] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[5] B. Faverjon. Object level programming of industrial robots. In *IEEE International Conference on Robotics and Automation*, pages 1406–1412, San Francisco, 1986.

[6] K. Fujimura and H. Samet. Hierarchical strategy for path planning among moving obstacles. *IEEE Trans. on Robotics and Automation*, February 1989.

[7] S. Kambhampati and L. Davis. Multiresolution path planning for mobile robots. *IEEE Journal of Robotics and Automation*, 2(3):135–145, September 1986.

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

[9] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Trans. on Computers*, February 1983.

[10] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. Computer Science Department Technical Report 1087, University of Wisconsin, Madison, May 1992.

[11] D. Zhu and J.-C. Latombe. New heuristic algorithms for efficient hierarchical path planning. *IEEE Trans. on Robotics and Automation*, 7(1):9–20, February 1991.