# Minimum Spanning Trees on Polyhedra

Michael J. Spriggs
Department of Computer Science
University of Saskatchewan
michael.spriggs@usask.ca

J. Mark Keil
Department of Computer Science
University of Saskatchewan
keil@cs.usask.ca

**ABSTRACT**

In this paper, we consider the problem of generating a minimum spanning tree (MST) of a set of sites lying on the surface of an open polyhedron. The distance between any two sites is the length of a shortest path between them that is constrained to lie strictly upon the polyhedral surface. We present two algorithms, the first of which, when given m points on an n-faced polyhedron, produces an MST in $O(n^2 log m + m^2 log m)$ time and O(n+m) space. In our second algorithm the running time is $O(n^2 log m + mn(m/n)^{1/2} log(m/n) log m)$. Along with this improvement in the time bound, space requirements increase slightly, to O(n+m*log*m).

## 1. INTRODUCTION

In this paper, we consider the computation of a *minimum spanning tree* (MST) of a set of point sites lying on the surface of an open polyhedron. The distance between any two sites is defined as the length of a shortest path between the sites, which itself lies entirely upon the polyhedral surface. We call an MST generated in this distance metric a *polyhedral minimum spanning tree* (PMST).

One potential application of this work is to the area of network design. A PMST might better model the problem of designing a low-cost spanning network over mountainous terrain than its planar counterpart, where the terrain is approximated by a polyhedral surface.

The paper is organized as follows. In Section 2, we give a background to the problem, including the motivations for this research, and an introduction to the subject of shortest-paths on polyhedral surfaces. In Section 3 we introduce our PMST algorithm. In Section 4 we present modifications to our algorithm that reduce our time bound, with a slight increase in the space bound. Conclusions are stated in Section 5.

## 2. BACKGROUND

### 2.1 Motivation

The distance metric used in this paper constrains paths to lie entirely upon the polyhedral surface. This metric has a nice property: shortest paths can be computed in polynomial time. If paths were allowed to stray from the polyhedral surface, the problem of discovering a shortest path becomes NP-hard [2]. Thus, over the years there has been considerable interest in this distance metric (e.g. [1,3,5,7,8,9,10]).

One approach to generating a geometric MST is to first compute some (small) superset of MST edges, such that edges of this superset which define an MST can be selected quickly. For example, given m sites in the plane and using an $L_2$ distance metric, an algorithm might first compute a *Delaunay triangulation* (DT) of the sites. We know that MST $\subset$ DT in this metric. The DT is of size O(m) and can be computed in O(m*log*m) time. A MST can be derived from the DT in O(m) time.

When on a polyhedron, we might compute the DT as the dual of the Voronoi diagram of the sites. However, the complexity of storing each straight (or hyperbolic) line segment of the Voronoi diagram of m sites on an n-faced polyhedron (where n > m) is $\Theta(n^2)$ [8]. (For an arbitrary m and n, the complexity is $\Theta(n^2+mn)$.) The Voronoi diagram is computable in time $O(n^2 log n + m^2 log m)$ [7].

Thus, a PMST algorithm that first computes the DT via the Voronoi diagram could require quadratic space. Our algorithms significantly improve upon this space bound, using linear (and nearly linear space) in the generation of a PMST.

Our PMST algorithms improve slightly upon the time bounds for polyhedral Voronoi diagram generation. In the case where we assume n > m, our algorithms reduce the bound of [7] from $O(n^2 log n)$ to $O(n^2 log m)$. When m > n, the time bound stated in [7] is $O(m^2 log m)$. The running time of our first PMST algorithm is of the same complexity. Our second PMST algorithm reduces this to $O(mn(m/n)^{1/2} log(m/n) log m)$.

Aside from the papers dealing with the problem of generating the Voronoi diagram of sites on a polyhedral surface (e.g. [7,8]), we do not know of any previous work which addresses the problem of generating a PMST.

### 2.2 Shortest Paths on Polyhedrons

The shortest-path algorithm by Chen and Han [3] preprocesses the surface of an n-faced polyhedron relative to a source site, such a shortest path between the source site, and any other site on the surface can be computed in logarithmic time. The preprocessing step requires $O(n^2)$ time and O(n) space. Algorithms presented in this paper rely heavily upon techniques developed in [3].

Until recently, this was the smallest known bound for computing a shortest path on a polyhedral surface. Kapoor [5] has now announced an improvement upon this result. He claims that his algorithm can compute a shortest path between two sites on a polyhedron within O(n*log*n) time.

Many polyhedral shortest-path algorithms employ a technique called *unfolding* (e.g. - [3,5,7,8,10]). An unfolding is performed with respect to a polyhedral surface and an edge-sequence (edges of the polyhedron). Two edges, adjacent in the edge-sequence, must reside on the boundary of a common face of the polyhedron.

To perform an unfolding, take the first edge of the edge sequence. Consider the two faces of the polyhedron that are adjacent at this edge. Rotate these faces about the edge until they are coplanar. Do likewise for the remaining edges of the edge-sequence. In the end, what is left is a set of faces embedded in the plane, called a *planar map* (see Figure 2.1). Note that this figure may be self-overlapping.

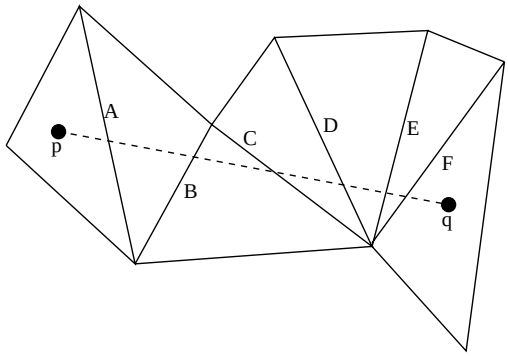


**Figure 2.1:** The planar map of a polyhedral surface with respect to edge-sequence (A, B, C, ..., F). The dots, p and q, represent sites on the polyhedral surface. The dashed line is a shortest path between p and q embedded in the planar map.

Assume for now that we are dealing with an open, convex polyhedron. A shortest path between two sites on the polyhedral surface will pass through some edge-sequence. Note that the path will not pass through any of the corners (vertices) of the polyhedron [10]. When the surface is unfolded with respect to some shortest path's edge-sequence, the embedded shortest path unfolds to a straight-line segment [10] (as shown in Figure 2.1). Thus, finding a shortest path on a convex polyhedron entails merely finding the edge sequence through which a shortest path passes.

A shortest path will intersect with each face of the polyhedron at most once. Therefore, any valid shortest-path edge-sequence on an n-faced polyhedron will be a list of edges of length < n [10].

Chen and Han [3] give an algorithm for constructing an *edge-sequence tree* with respect to a single site on a polyhedral surface (the *source site*). For a convex polyhedron, an edge-sequence tree is a rooted tree that encodes all possible shortest-path edge sequences between the source site and every other point on the polyhedral surface. Each node of the edge-sequence tree corresponds to an edge of the polyhedron. Shortest-path edge-sequences are encoded as paths through the edge-sequence tree starting at the root node.

As mentioned earlier, in [3] we are shown how such an edge-sequence tree can be computed in time $O(n^2)$ and space O(n) for a single site on an open, convex, n-faced polyhedron.

Shortest paths on general polyhedra (possibly non-convex) behave similarly to those on convex polyhedra. The main difference is that shortest paths may pass through vertices of the polyhedron [7]. Thus, on a non-convex polyhedron, edge and vertex sequence information needs to be stored. Notice that when two faces are unfolded about a vertex there is no unique planar map, in the sense that the faces are free to rotate about their common vertex (see Figure 2.2).

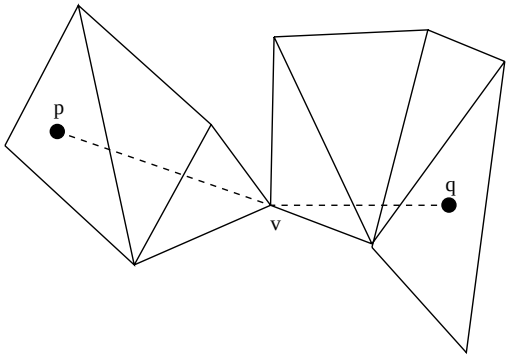


**Figure 2.2:** The planar map of an edge/vertex sequence on a non-convex polyhedron. The dashed line is an embedded shortest path between sites p and q. Notice that this shortest path passes through vertex v.

The edge-sequence tree algorithm of [3] can be modified to generate an edge/vertex-sequence tree for a given source point on a general (possibly non-convex) polyhedron. This modified version runs within the same complexity bounds as the original.

Note that our algorithms, as with other algorithms that employ polyhedral unfolding, assume that the model of computation used is a real-RAM machine that performs each infinitely precise arithmetic operation in unit time. An unfolding is essentially a rotation in 3-space. Unfolding does not work when using fixed-precision, floating-point arithmetic, as numerical errors are compounded.

# 3. AN ALGORITHM FOR THE PMST

## 3.1 The Basic MST Algorithm

Consider the following MST algorithm. Initially, the MST consists of the set of sites, which can be thought of as a forest with no edges. In each iteration, find a minimum-length edge between every disjoint tree (*partial-MST*) and a site not in the tree (a *nearest neighbor* of the partial-MST). For each edge found, add the edge to the MST edge-set so long no cycle is formed in the resultant graph. Iterate until a single partial-MST remains. This final partial-MST is a valid MST of the sites.

Notice that in each iteration of this algorithm, each partial-MST is connected to at least one other partial-MST. Thus, the number of remaining partial-MSTs is reduced by at least half with every iteration. In the worst case, the number of partial-MSTs decreases by exactly half in every iteration. Thus, given m sites, the bound on the number of iterations is $\Theta(\log m)$.

This is the underlying MST algorithm used in this paper. What is needed is an algorithm that can, at each iteration, efficiently compute a nearest neighbor for each partial-MST.

## 3.2 Computing the Nearest-Neighbors

For the sake of simplicity assume, as in [3], that the polyhedron consists solely of triangular faces. Note that this does not limit the generality of the algorithm. The surface of an n-faced polyhedron can be triangulated in linear time, resulting in an O(n)-faced polyhedron.

Also for the sake of simplicity, we make assumptions about the location of the sites. Sites in the *general position* obey the following.

- No site lies on an edge or vertex of the polyhedron.
- The distance between any two sites is not equal to the distance between any other two sites. This implies that there will be exactly one nearest neighbor for each partial-MST, and thus only one PMST is possible.
- Exactly one shortest path exists between any two sites. Note that, in general there may be multiple shortest-paths between any two sites.

The following lemma deals with sites in the general position on the surface of an open, convex polyhedron. In Section 3.3 this result is generalized to non-convex polyhedra and sets of sites which do not conform to the general position assumption.

**Lemma 3.1** *Given an open, convex, n-faced polyhedron, m sites on the surface of the polyhedron in the general position, and an edge set (PMST edges) which connects sites into a set of partial-MSTs. In $O(n^2+mn)$ time and $O(n+m)$ space, a set of edge-sequence trees can be computed such that:*

*(1) Each edge-sequence tree of the set corresponds to a particular source site (i.e. the paths through a tree from the root node correspond to shortest-path edge-sequences from the associated source site).*

*(2) A shortest-path edge-sequence between each partial-MST and its nearest neighbor is encoded in the set of edge-sequence trees.*

**Proof:** Our algorithm works in a manner similar to the edge-sequence tree algorithm of [3], except that instead of generating a single edge-sequence tree relative to one source site, multiple sources are considered. A set of edge-sequence trees is generated such that each edge-sequence tree corresponds to a particular source site.

In our algorithm, each node of an edge-sequence tree is a 4-tuple, (e,I,*Proj*(I,e),p), where e is an edge of the polyhedron, I is an image of the source site through some edge-sequence, and *Proj*(I,e) is the *projection* of I onto edge e. The projection of I on e is defined as the closed subsegment of e visible to the source image, I, through the interior of the planar map constructed with respect to some edge-sequence (see Figure 3.1). Likewise, the *shadow* of a source site is the region of a face visible to a site through the interior of the planar map constructed with respect to some edge-sequence. Finally, p stores the source site itself. This element of the 4-tuple is included so, given a node of an edge-sequence tree, the corresponding source site can be identified in unit time.

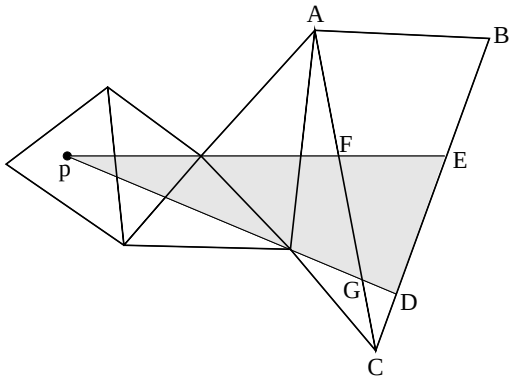


**Figure 3.1:** The planar map of an edge-sequence from site p. With respect to the edge-sequence, the segment DE is the projection of p on edge BC, and the polygon ΔDEFG is the shadowed region of face ΔABC.

Our algorithm begins by initializing an edge-sequence tree for each site, s. Initialization consists of generating a dummy root for s. Then, for each edge of the polyhedron bounding the face upon which s lies, e, insert node (e,s,e,s) as a child of the dummy root. We say that this edge-sequence tree *corresponds* to site s.

Let dummy-root nodes lie at level zero of their respective edge-sequence trees. The general step is as follows. At the $i^{th}$ iteration, for all nodes at level i in the set of edge-sequence trees, v = (BC,I,*Proj*(I,BC),p), compute the *potential children* of v as follows.

Suppose that face $\Delta$ABC is the face shadowed by node v (see Figure 3.2).

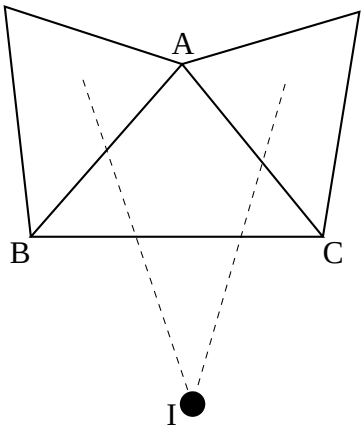


**Figure 3.2:** The dashed lines denote the boundaries of the projection of source image I onto edge BC, and across shadowed face $\Delta$ABC.

Unfold I about BC until coplanar with $\Delta$ABC. Call this new source image, I′. Compute the projections of I′ upon edges AB and AC. These projections are denoted *Proj*(I′,AB) and *Proj*(I′,AC) respectively. The potential children of v are the 4-tuples (AB,I′,*Proj*(I′,AB),p) and (AC,I′,*Proj*(I′,AC),p).

If one of *Proj*(I′,AB) and *Proj*(I′,AC) is an empty line segment, then the corresponding child node need not be inserted into the tree, since the subtree rooted at this child cannot possibly encode any shortest-path edge-sequence (i.e. the corresponding edge sequences, when unfolded, will not contain any straight-line segments from the source site).

If both projections are non-empty, append both children to node v. We know that each shortest path passes through each face of the polyhedron at most once [10]. Therefore, growth of the edge-sequence trees may be halted upon reaching height n without fear of pruning any valid edge-sequences from the set of edge-sequence trees.

This algorithm generates a set of edge-sequence trees that encodes all possible valid shortest-path edge-sequences from every site. However, in the worst case, the resulting data structure is of exponential size. A little more sophistication is required.

In the case of the single-source edge-sequence tree, Chen and Han [3] notice that, for each angle interior to a face of the polyhedron, we need allow only one of the nodes in the edge-sequence tree to fork to two children. A node which contains an angle $\phi$ in the shadow of its source image, and whose source image lies nearest the apex of the angle is said to *occupy* $\phi$ (see Figure 3.3).

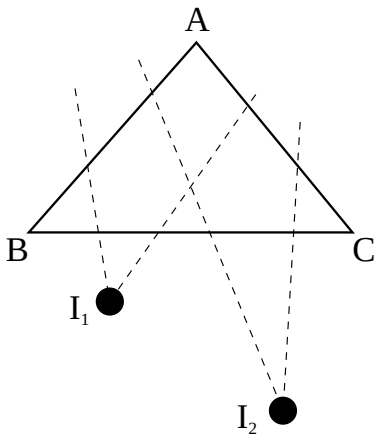


**Figure 3.3:** Source image $I_1$ is closer to point A than is source image $I_2$. $I_1$ will be closer than $I_2$ to every point past edge AB. The edge-sequence tree node containing $I_1$ (and corresponding projection on BC) is said to occupy angle $\angle$BAC over the node containing $I_2$ (with its corresponding projection on BC).

Notice that the lying closest to the apex of the angle may contain shortest-path edge-sequence information in both subtrees, while any other node whose source image lies opposite the angle can only contain potential shortest-path information in one of its subtrees.

Throughout the construction of the edge-sequence tree of [3], the property “one-angle, one-split” is maintained, where only the node which occupies an angle is allowed to fork to two children. Since there are O(n) angles on the polyhedron, there are O(n) splits in the edge-sequence tree. A node whose shadow covers an angle, but cannot split to two children because of another occupying node is said to have one of its subtrees *blocked* by the occupying node.

We modify this idea slightly in our algorithm. Instead of allowing only one split per angle in a single edge-sequence tree, we allow two splits per angle throughout the entire set of edge-sequence trees.

Define the *primary occupant* of an angle as a node of an edge-sequence tree which occupies the angle by the definition from [3] (i.e. the primary occupant of an angle is the node whose shadow covers the angle, and whose source image lies closest the apex of the angle). Define the *secondary occupant* of an angle as a node such that:

(1) The node’s shadow covers the angle; AND

(2) The corresponding source-site of the node is not in the same partial-MST as that of the primary occupant; AND

(3) The source image is closer to the apex of the angle than that of any other node whose source-site is not in the same partial-MST as the primary occupant's source-site.

The concept of "blocking" is likewise extended for our definitions of angle occupancy. A child node can be blocked by either one, or both occupants of an angle. The blocking rules are slightly more complex than those of the single-source algorithm and are discussed below.

For a node of one of the edge-sequence trees, v, let *site(v)* denote the unique site corresponding to the edge-sequence tree containing v.

In our algorithm, both the primary and secondary occupants of an angle are allowed fork to two children at that angle. Other nodes whose shadow contains the angle are allowed at most one child each.

A set of edge-sequence trees generated in this manner encodes the shortest-path edge-sequence between every partial-MST, t, and its nearest-neighbor as a path through the edge-sequence tree of the nearest-neighbor. To set up a proof by contradiction, suppose that no such path exists in the nearest-neighbor's edge-sequence tree. This means that the path encoding the shortest-path edge-sequence is blocked at some angle, $\phi$.

At an angle of blockage, $\phi$, there must be two blocking nodes: the primary and the secondary occupant of $\phi$. Suppose that the primary occupant is a node, u, and the secondary occupant is a node, v. Further, suppose that the blocked node of the shortest-path edge-sequence is node w, and thus, *site*(w) is the nearest-neighbor of t.

If *site*(u) is in t, then by the definition of secondary occupant, *site*(v) is not in t. Obviously, *site*(w) is not in t, by the definition of a nearest-neighbor of a partial-MST. If *site*(v) = *site*(w), then w is not on the shortest-path edge-sequence, since a shorter route to t passes through v. If *site*(v) $\neq$ *site*(w), then *site*(v) is closer to t then is site(w), contradicting the fact that *site*(w) is the nearest-neighbor of t.

Similarly, if *site*(u) is not in t, then a path from t's nearest-neighbor to t cannot pass through the edge-sequence corresponding to blocked node w. It would be shorter to go from *site*(u) to t (passing through u) than from *site*(w) to t (passing through w).

By pruning only those paths that are blocked by both the primary and secondary occupants of each angle, we are guaranteed that the resulting set of edge-sequence trees encodes all shortest-path edge-sequences between each partial-MST and its nearest-neighbor. The edge-sequences are encoded as a path through the edge-sequence tree of each nearest-neighbor.

In our algorithm, the current primary and secondary occupants of each angle are stored in an array, and can thus be determined for a given angle in O(1) time.

As in the exponential-time edge-sequence tree algorithm described above, the processing of each node, v, at the $i^{th}$ level of an edge-sequence tree involves first computing the potential children of the node. If v has a non-empty projection in only one of the child nodes, then only the child with the non-empty projection is inserted.

If v has more than one potential child, then there are two cases: (1) *site*(v) is in the same partial-MST as either the current primary or secondary occupant of shadowed angle, $\phi$, or (2) *site*(v) does not share a common partial-MST with either the primary or secondary occupant of $\phi$.

In Case (1), let v′ denote the node which is either the primary or secondary occupant of $\phi$, such that site(v′) resides in the same partial-MST as site(v). (It might be the case that site(v) = site(v′).) If v can block a subtree of v′, then the subtree of v′ that is blocked by v is removed and deleted. If, on the other hand, v′ is able to block one of v's children, then only the child of v that is not blocked by v′ need be added.

Since *site*(v) and *site*(v′) are both in the same partial-MST, only one of them is allowed to occupy the angle, by the definitions of primary and secondary occupants. If v blocks one of v′'s children, then put v in v′'s former occupancy position (primary or secondary occupant of $\phi$). Since each angle stores only one primary and one secondary occupant, v′ is no longer stored as an occupant of $\phi$.

Following this procedure, the secondary occupant of $\phi$ might be set to node v. However, it might be the case that v's source image is closer to the apex of $\phi$ than the source image of the current primary occupant of $\phi$. If this is the case, then this inconsistency can be remedied by simply swapping the primary and secondary occupants of $\phi$.

In Case (2), *site*(v) shares a partial-MST with neither of the sites corresponding to the occupants of $\phi$. Let v′ denote the primary occupant of $\phi$, and let v″ denote the secondary occupant of $\phi$. There are three subcases.

(1) Nodes v′ and v″ can both occupy $\phi$ over v. In this case, add only the child of v not blocked by either v′ or v″. This may mean that v has no children if both subtrees of v are blocked.

(2) Node v can occupy $\phi$ over v″, but v′ can occupy $\phi$ over v. Node v″ loses one of its subtrees, and the new secondary occupant of $\phi$ is v, which forks to both children.

(3) Node v can occupy $\phi$ over v′ (and therefore, v can also occupy $\phi$ over v″). Since v′ is the primary occupant of $\phi$, it can block v″. Node v must now be the primary occupant, implying v′ is secondary occupant. Node v″ loses the subtree blocked by v′.

These three subcases are complete. Notice that the situation in which v can occupy $\phi$ over v″, and v cannot occupy $\phi$ over v′ can never arise.

Once a node has been processed, the current primary and secondary occupants of the shadowed angle $\phi$ are correctly set (given all nodes currently in the set of edge-sequence trees, whose shadow covers $\phi$). Also, the occupants of $\phi$ split to two children, while all others have at most one child. Our algorithm is summarized in Figure 3.4.

The time bound of our algorithm can be proven in manner similar to the time bound of the edge-sequence tree algorithm of [3]. At first, m edge-sequence trees are initialized, requiring O(m) time. Consider now the time taken to insert nodes over the course of the entire run (time for the removal of nodes will be considered shortly). Since we allow (at most) two splits per angle of the polyhedron, after the $i^{th}$ level has been added to the set of edge-sequence trees, the set contains O(n+m) leaf nodes at the $i^{th}$ level.

Computing the children of each node can be accomplished in O(1) time, since each node is of constant complexity and spawns at most two children. Since there are O(n+m) nodes at the previous level, the time required to add an additional level to the set of trees is O(n+m). Trees are grown to height n. Therefore, the time required for inserting nodes into the edge-sequence trees over the course of the entire execution is $O(n^2+mn)$.

As for the removal of subtrees, the same counting trick as in [3] is applied. It is clear that if each node of the tree is given two time units, one for addition and one for removal, then overall, the time for the addition and removal of each node generated is $O(n^2+mn)$. Some of the generated nodes will not be removed, and in this case, their second allocated time unit remains unused upon completion of the algorithm.

The space bound of the algorithm described in Figure 3.4 is also $O(n^2+mn)$. However, by using another trick from [2], the paths through the edge-sequence trees can be compressed by storing only those nodes which are interior to the tree, and which split to two children. Since there are O(n) splits in the entire set of edge-sequence trees and O(m) edge-sequence trees, the space is bounded by O(n+m). It is not difficult to implement the entire algorithm so as to run within this linear space bound.

With this, Lemma 3.1 is proven. ❑

Recall that once the set of edge-sequence trees is computed, the edge-sequence of a shortest path between each partial-MST and its nearest-neighbor (site) is encoded as a path through the edge-sequence tree corresponding to the nearest-neighbor.

**Algorithm:** *Edge-Sequence Tree Generator*.
**Input:** *A convex, n-faced polyhedron, P, m sites in the general position and lying on the surface of P, S, and an edge set which partitions S into a set of partial-MSTs.*
**Output:** *A set of edge-sequence trees, such that a shortest path between each partial-MST and its nearest neighbor (site) is encoded as a path through the edge-sequence tree corresponding to the nearest-neighbor.*

```
For all s ∈ S.
  Initialize an edge sequence tree for s.
For i := 1 to n.
   For every edge-sequence tree node, v, distance i from a root.
      Compute the potential children of node v.
      (Let φ denote the angle opposite the source image of v.)
      If v has only one potential child, then
         Insert v's child node.
      Else
         If site(v) resides in the same partial-MST as site(v′),
           such that v′ is currently the primary (or secondary)
           occupant of φ, then:
            If v can occupy φ over v′, then
               Remove the subtree of v′ blocked by v.
               Mark v as occupant of φ in v′'s old position.
               Insert both children of v.
               Update the labels of primary and secondary occupants
                 of φ. (i.e. swap φ's occupants if necessary.)
            Else
               Insert the child of v not blocked by v′.
         Else
            (let v′ denote the primary, and v″ the secondary
              occupant of angle φ.)
            Case #1: v′ and v″ can occupy φ over v.
               Insert child of v blocked by neither v′ nor v″
               (note that, as a result, v might have no children).
            Case #2: v can occupy φ over v″, and v′ can occupy
              φ over v.
               Remove the subtree of v″ which is blocked by v.
               Mark v as secondary occupant of φ.
               Insert both children of v.
            Case #3: v can occupy φ over both v′ and v″.
               Remove the subtree(s) of v″ blocked by v and v′.
               Mark v as the primary occupant of φ.
               Mark v′ as the secondary occupant of φ.
```

**Figure 3.4:** A description of an algorithm for the generation of a set of edge-sequence trees.

All of the shortest-path edge-sequences required for one iteration of the MST algorithm of Section 3.1 are encoded in the set of edge-sequence trees. However, further processing is needed in order to determine the nearest-neighbor of each partial-MST. The following theorem shows how this can be done in a brute-force manner.

**Theorem 3.1** *A PMST of m sites in the general position on an open, convex n-faced polyhedron can be computed in time* $O(n^2 \log m + m^2 \log m)$ *and space* $O(n+m)$.

**Proof:** Consider the basic MST algorithm described in Section 3.1. In each iteration, generate a set of edge-sequence trees as described in the proof of Lemma 3.1. Since there are O(m) roots, and O(n) splits in the entire set of edge-sequence trees, there are a total of O(n+m) paths running through the entire set of rooted trees.

Suppose that for each site, q, the following variables are stored: *nearest_neighbor*(q), and *distance*(q). The variable *nearest_neighbor*(q) is initialized to NULL, and *distance*(q) is initialized to $+\infty$.

For an edge-sequence tree, let p denote corresponding source site. For every path through the tree, unfold the polyhedral surface along the edge-sequence defined by the path, and compute the straight-line ($L_2$) distance between p and every site, q, embedded on a face of the planar map, such that:

- Site q is visible to site p through the interior of the resulting planar map (see Figure 3.5).
- Sites p and q are not members of the same partial-MST.

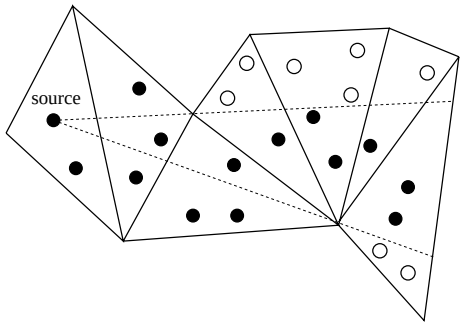


**Figure 3.5:** A planar map of a portion of a polyhedral surface with embedded sites. The sites visible to the source through a planar map are drawn as solid circles, and sites not visible to the source are drawn as hollow circles.

If *distance*(q) is greater than the distance between q and p, then assign *nearest_neighbor*(q) the site p, and assign *distance*(q) the distance between p and q.

Note, by the edge-sequence tree algorithm of Lemma 3.1, paths are grown to length n. However, in doing so, a single path may cross through same face of the polyhedron multiple times. Such edge-sequence tree paths cannot contain any polyhedral shortest-paths, since a polyhedral shortest-path on a convex polyhedron intersects each face at most once [10]. To avoid examining sites on a face more than once per path, simply mark each face as "visited" as the path is unfolded. Stop searching along the path as soon as a face is met for the second time.

Repeat this process for all O(m+n) paths through the set of edge-sequence trees. By Lemma 3.1, the edge-sequence between each partial-MST and its nearest-neighbor is stored as a path through the edge-sequence of the nearest-neighbor. In the above procedure, all paths through all edge-sequence trees are examined. Therefore, upon completing this procedure, each partial-MST's nearest-neighbor, will be stored as the nearest-neighbor of one of its component sites.

It takes O(1) time to compute the distance from the source site to each site, and to determine whether or not it is visible to the source. In the processing of each path, O(n) faces are unfolded, and O(m) sites examined. Thus, each path requires O(n+m) processing time. The time required to process all O(n+m) paths of the set of edge-sequence trees is:

$$O(n+m) * O(n+m) \quad (3.1)$$

$$= O(n^2 + m^2) \quad (3.2)$$

This bound supercedes the $O(n^2+mn)$ time bound on the generation of the set of edge-sequence trees. Since each path is "decompressed" and processed individually, the algorithm can be run in O(n+m) space.

In the preceding discussion, we have shown how every partial-MST can be attached to its nearest-neighbor in a single iteration of the MST algorithm of Section 3.1. Since the MST algorithm iterates O(*log*m) times, the overall time bound on our PMST generation algorithm is:

$$O(\log m) * O(n^2 + m^2) \quad (3.3)$$

$$= O(n^2 \log m + m^2 \log m) \quad (3.4)$$

The only information stored from iteration to iteration is the set of edges of partial-MSTs. Each edge of these partial-MSTs is stored as a pair: the two sites denoting the endpoints of the edge. Thus, each edge requires constant space and the storage of all m edges requires O(m) space. Note that, unlike many geometric graphs, the edges of our PMST have no spatial component. This is discussed briefly in Section 3.3.

From the space bounds of the component algorithms, and the space bound of PMST storage, the overall bound on space is O(n+m). ❑

As a corollary of Theorem 3.1, a *polyhedral nearest neighbor graph* (PNNG) in which each site is connected to (one of) its nearest neighbor(s) by an edge, is computable in $O(n^2+m^2)$ time and O(n+m) space.

## 3.3 Generalizations and Discussion

It is not difficult to extend the PMST algorithm of Section 3.2 to accommodate general (possibly non-convex) polyhedra by adopting techniques similar to those found in [3].

Recall that on a non-convex polyhedron, a shortest path may pass through a vertex of the polyhedron. This behavior can be dealt with by allowing two varieties of nodes into the edge-sequence trees: edge nodes and vertex nodes. Edge/vertex-sequence trees may be generated as before, however, when one of the paths passes through a vertex of the polyhedron, the vertex is considered a *pseudosource*. In the subtree of a vertex node, the corresponding vertex becomes the source image stored.

Edge nodes are now 5-tuples, (e,I,*Proj*(I,e),p,$\delta$), such that e is an edge, I is the image of a source site (or pseudosource vertex). *Proj*(I,e) and p are defined as before. The last element of the 5-tuple, $\delta$, is the smallest known distance between site p, and the point (either the source site, or a vertex of the polyhedron) to which I corresponds. Vertex nodes are triples, (v,p,$\delta$), where v is the vertex of the polyhedron to which the node corresponds, p is a site, and $\delta$ is the shortest-path distance between v and p.

Each edge-node may fork (up to) three children; two edge-nodes and one vertex node. The vertex node corresponds to the vertex across the shadowed face of the edge node. Each vertex node may have up to twice the number of children as is the degree of the corresponding polyhedral vertex; half of the children correspond to vertices adjacent to the vertex on an edge of the polyhedron, and half correspond to edges opposite the vertex.

The algorithm of Section 3.2 can be modified so that the new splitting rules are observed in the construction of the set of edge/vertex-sequence trees. Particularly, for each angle of the polyhedron, allow the primary and secondary occupants to split to two edge nodes. Similarly, for each vertex of the polyhedron, allow two nodes to occupy the vertex; a primary node, and a secondary node. A similar proof to that in Lemma 3.1 can be applied to show that the resulting set of edge/vertex-sequence trees will encode all shortest-path edge sequences between partial-MSTs and their nearest-neighbors.

As in [3], the PMST algorithm can be adapted to non-convex polyhedra without an increase to the complexity bounds of the algorithm.

The PMST algorithm of Section 3.2 requires that sites conform to general position assumptions. These assumptions imply that, during the construction of the PMST, each partial-MST has exactly one nearest-neighbor, and one shortest-path to that nearest-neighbor. Further, by these assumptions, sites do not lie on edges or vertices of the polyhedron. We will now show that the general position assumptions can be discarded.

If we remove the assumption that the distance between any two sites is not equal to the distance between any other two, then it may be the case that a partial-MST has more than one nearest-neighbor. However, all of the paths from all nearest-neighbors of a partial-MST cannot be blocked.

Lemma 3.1 can be altered slightly to state that for each partial-MST, the set of edge-sequence trees encodes the edge-sequence of the shortest-path between the partial-MST and (at least) one of its nearest neighbors. Likewise if we drop the assumption that there is only one shortest path between any two sites, the set of edge-sequence trees is guaranteed to encode at least one such shortest path.

For each iteration of the basic MST algorithm of Section 3.1, only one nearest neighbor of each partial-MST is required. Thus, our PMST algorithm needs no changes to accommodate the loss of these assumptions.

We also discard the assumption that no site lies upon a vertex or an edge of the polyhedron. When a site lies upon a vertex, its root is not a dummy node, but a vertex node. If a site lies upon an edge of the polyhedron, e, then the children of the dummy root are the edge-nodes corresponding to the four remaining edges bounding the two faces adjacent about e.

Note that in this paper, unlike many treatments of geometric graphs, the edges of the PMST do not have a spatial component (i.e. an edge is specified entirely by its two endpoints; how to get from one endpoint to the other is not stored). This is due to the fact that storing spatial information might require greater-than-linear space, depending upon the method of storage. For some applications, it may be desirable to store the shortest-path information with the edges. Below, we discuss two techniques by which this can be achieved.

First, recall that on an open, convex polyhedron, all shortest paths become straight-line segments upon unfolding. One possibility is that PMST edges are stored as triples, (p, D, q), such that p is the starting point of the edge (a site), D is an initial direction of travel across the surface, and the edge ends at site q. Thus, by specifying starting and ending points, and an initial direction of travel, a single embedded edge can be reconstructed in O(n) time by simply following the straight-line path across the surface of the convex polyhedron. Storing a PMST using this approach requires O(m) space. Note that this approach works only with convex polyhedra.

Another approach is to store individual line segments of each edge of the PMST. Since each edge is a shortest-path between two sites, each PMST edge may cross up to n faces of the polyhedron. In this manner storing a single PMST edge requires O(n) space, and so O(mn) space is sufficient for the storage of all m paths.

## 4. MODIFICATIONS

The linear space bound of the PMST algorithm of Section 3 significantly improves upon the quadratic complexity needed if a Voronoi diagram is computed prior to the generation of a PMST. There is also a small reduction in the time bound when n>m, from $O(n^2 log n)$ (as stated in [7]) to $O(n^2 log m)$. However, while an iteration of the PMST algorithm of Section 3 requires $O(n^2+m^2)$ time, the set of edge-sequence trees can generated within only $O(n^2+mn)$ time. This fact motivated us to seek a more efficient method of extracting nearest-neighbor information from the set of edge-sequence trees.

In this section, we show that by preprocessing the sites upon each face, the time bound can be reduced significantly (for m>n) while suffering a slight increase in the space bound.

Given a set of sites in the plane, divide the space about a site, p, into a set of wedges, such that the angle on each wedge <60°. Yao [11] tells us that the set of sites nearest p in each of the wedges is a superset of the sites to which p can be adjacent in a planar MST. Notice that the number of wedges about p is of size O(1).

We can make use of this knowledge in finding potential PMST edges between sites that lie on a single face. For each face of the polyhedron, consider the set of sites lying upon this face, S. For each $p \in S$, find an O(1) sized superset of potential edges between p and the other sites in S. The edges discovered in this manner are a superset of PMST edges between sites lying on a single face of the polyhedron. For m sites, this can be accomplished in O(m*log*m) time and linear space [11].

We can deal with PMST edges between sites on different faces in a similar manner. As in Section 3, first generate a set of edge-sequence trees of all sites.

Recall, when "decompressing" and examining a path through the edge-sequence trees, we are interested in computing distances between source site of the path and all sites on the planar map defined by the path, such that the other sites are:

(1) not in the same partial-MST as the source, and

(2) visible through the interior of the planar map

Notice that when searching the paths through the set of edge-sequence trees, the portion of a face visible to a source site through a planar map is defined by a wedge with angle <180° at the apex (see Figure 4.1). This region can be divided into three (or fewer) wedges of angle <60° each. The set of nearest-neighbors within the set of smaller wedges forms a O(1)-sized superset of PMST edges between the query point and sites on the face.

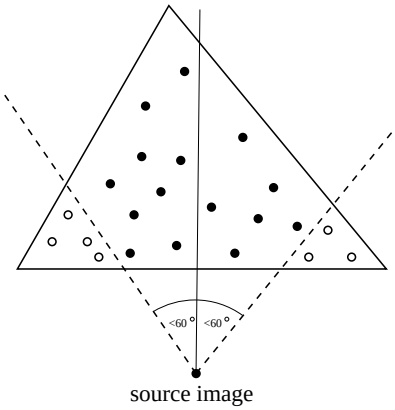


**Figure 4.1:** The heavy dashed lines denote the boundary of the region visible to the source through some edge sequence. This region is divided into two wedges, each less than 60°.

This approach allows for a quick determination of a superset of PMST edges between two sites that lie on different faces. We require a means of quickly computing the nearest-neighbor within each wedge.

The modifications to our PMST algorithm involve the use of a linear-space, simplex range-search structure. Such range-search structures are basically trees in which each node corresponds to a *simplical partition* of the space. Each partition contains a subset of the sites. The set of sites in a node's simplical partition is known as *the canonical subset* of the node. The canonical subsets of the children of a node are disjoint.

**Lemma 4.1** *Given k sites in the plane, the sites can be preprocessed in $O(k^{1+\varepsilon})$ time and O(k*log*k) space (for some arbitrary $\varepsilon$>0), such that, when queried with a wedge, in $O(k^{1/2}$log*k) time the site can be found which both:*

*(1) lies within the wedge*

(2) *lies closer to the apex of the wedge than any other site within the wedge*

**Proof:** Matousek [6] shows how a simplex range-search structure can be computed in time $O(k^{1+\varepsilon})$, for some arbitrary $\varepsilon > 0$. The resulting data structure requires linear space, and when queried with a particular range of constant complexity (i.e. the query wedge) returns a set of $O(k^{1/2})$ nodes. The canonical subsets of these returned nodes are disjoint and contain all sites within the query range.

Often, each node of a range-search structure stores an integer corresponding to the size the node's canonical subset. This is useful in answering *range-counting* queries, in which the number of sites within an arbitrary polygonal region is computed.

In contrast, in our modified PMST algorithm, each node of the range-search structure stores a point-location data structure of the Voronoi diagram of the sites in the corresponding canonical subset. The point-location data structure is used to compute the Voronoi cell containing a query point (i.e. the nearest-neighbor of the query point).

A point-location structure can be implemented to use linear space, and return answers to queries in logarithmic time. Given k′ sites, such a structure can be generated in O(k′*log*k′) time (see [4], for example).

In summary, our data structure is a range-search data structure, with point-location data structures at each node. When queried with a wedge, the $O(k^{1/2})$ nodes whose canonical subsets contain all sites in the range are found. For the canonical subset of each of these nodes, the site nearest the apex of the query wedge can be found in logarithmic time, with respect to the number of sites in the subset. Thus, given k sites in the plane, the time required to find the site nearest the apex in a query wedge is $O(k^{1/2}\log k)$.

Preprocessing the sites involves first generating the range-search tree (requiring $O(k^{1+\varepsilon})$ time, for some arbitrary $\varepsilon > 0$). The simplex range-search data structure of [6] is essentially a set of trees of height O(*log*k). Since the sites are disjoint at each level, and there are O(*log*k) levels overall in the range-search trees, the required preprocessing time is:

$$O(k^{1+\varepsilon}) + O(\log k) * O(k \log k) \quad (4.1)$$

$$= O(k^{1+\varepsilon}) \quad (4.2)$$

The space required for storing the data structure is the number of levels of the structure, multiplied by the size of the point-location structure per level (linear). Thus, space is:

$$O(k \log k) \quad (4.3)$$

With this, Lemma 4.1 is proven. ❑

From the discussion preceding Lemma 4.1, it should be clear how the data structure described in Lemma 4.1 is used in reducing the time bound of our PMST algorithm. The sites on each face, f, are preprocessed so that an O(1)-sized superset of potential PMST edges between sites on f, and a site not on f can be determined without examining every site on f.

We now prove the complexity bounds of our modified PMST algorithm.

**Theorem 4.1** *A PMST of m points on an n-faced polyhedron can be computed in* $O(n^2\log m + mn(m/n)^{1/2}\log(m/n)\log m)$ *time and* $O(n+m\log m)$ *space.*

**Proof:** Recall that the potential PMST edges between sites sharing a single face can be accomplished using nearly linear time. This bound is superceded by the complexity of computing potential PMST edges between different faces. Thus, we ignore this part of the algorithm in the following proof, and concentrate solely upon the complexity of computing PMST edges between sites lying on different faces.

Our modified PMST algorithm commences by preprocessing the sites on each face into the data structure described in Lemma 4.1. For each face, $f_i$, let $m_i$ denote the number of sites upon $f_i$, where $1 \le i \le n$. The space required of the data structure is, for all n faces:

$$\sum_{i=1}^{n} \left( O(1) + O\left( m_i \log m_i \right) \right) \quad (4.4)$$

The O(1) term in Equation 4.4 refers to the constant amount of storage required per face. An upper bound on Equation 4.4 is:

$$O\left( n + m \log m \right) \quad (4.5)$$

Recall that in every iteration of the basic MST algorithm (Section 3.1), a set of edge-sequence trees is generated, and then each path through each edge-sequence tree is "decompressed" and processed. In processing, we find potential PMST edges between the source-site of the path, and sites on faces crossed by the path. The time required for processing a single path though an edge-sequence tree is (by Lemma 4.1):

$$\sum_{i=1}^{n} \left( O(1) + O\left( m_i^{1/2} \log m_i \right) \right) \quad (4.6)$$

Equation 4.6 is largest when the sites are evenly distributed among the faces. This is because the complexity bound is less than linear for sites on each face.

The time required to unfold a single path through an edge-sequence tree, when sites are equally distributed over all faces is (from Equation 4.6):

$$\sum_{i=1}^{n} \left[ O(1) + O\left( \left( m/n \right)^{1/2} \log\left( m/n \right) \right) \right] \quad (4.7)$$

$$= O\left( n + n\left( m/n \right)^{1/2} \log\left( m/n \right) \right) \quad (4.8)$$

Multiplying the bound for each path unfolding (Equation 4.7) by the number of paths, O(n+m), gives the time bound for one iteration of the basic MST algorithm (Section 3.1).

$$O\left( n + n\left( m/n \right)^{1/2} \log\left( m/n \right) \right) * O\left( n + m \right) \quad (4.9)$$

$$= O\left( n^2 + nm\left(m/n\right)^{1/2} \log\left(m/n\right) \right) \quad (4.10)$$

Finally, multiplying this value by O(*log*m) iterations of the MST algorithm leads to the overall time bound on our modified PMST algorithm:

$$O\left( n^2 \log m + mn\left(m/n\right)^{1/2} \log\left(m/n\right) \log m \right) \quad (4.11)$$

Thus, the theorem is proven. ❑

The above result implies that a polyhedral nearest-neighbor graph (PNNG) can be computed in $O(n^2+mn(m/n)^{1/2}log(m/n))$ time and O(n+m*log*m) space.

## 5. CONCLUSIONS

In this paper we have considered the problem of generating a minimum spanning tree of a set of sites lying on the surface of an open polyhedron. This problem may have applications in the area of network planning over real terrain, modeled as a polyhedral surface.

The results obtained improve upon the bounds that might be required if a Voronoi diagram of the sites was generated before the PMST. The major improvement is in terms of the storage requirements. A Voronoi diagram may require quadratic storage space, while our algorithms use approximately linear space.

It would be interesting to discover whether the Kapoor's approach to computing polyhedral shortest-paths [5] can be applied to generating a PMST within a lower time bound than our algorithm. An approximation algorithm for generating a PMST would also be an interesting extension of this work.

## 6. REFERENCES


[1] Aronov, B., van Kreveld, M., van Oostrum, R., and Kasturirangan, V. Facility location on terrains. In Proceedings of 9th Annual International Symposium on Algorithms and Computation (ISAAC '98), 1998.

[2] Canny, J., and Reif, J. New lower bound techniques for robot motion planning problems. Proc. 28th IEEE Symp. Foundations of Computer Science, 1987, 49-60.

[3] Chen, J., and Han, Y. Shortest paths on a polyhedron. Internat. J. Comput. Geom. Appl. 6, 1996, 127-144.

[4] Edelsbrunner, H., Guibas, L. J., Hershberger, J., Seidel, R., Snoeyink, J., and Welzl, E. Implicitly representing arrangements of lines or segments. Discrete and Comput. Geom. 4, 1989, 433-466.

[5] Kapoor, S. Efficient computation of geodesic shortest paths. In proceedings of STOC '99, 1999, 770-779.

[6] Matousek, J. Range searching with efficient hierarchical cuttings. Discrete and Comput. Geom. 10, 1993, 157-182.

[7] Mitchell, J. S. B., Mount, D. M., and Papadimitriou, C. H. The discrete geodesic problem. SIAM J. Comput. 16, 1987, 647-668.

[8] Mount, D. M. Voronoi diagrams on the surface of a polyhedron. Technical Report 1496, Department of Computer Science, University of Maryland, 1985.

[9] Pankaj, K., Aronov, B., O'Rourke, J., and Schevon, C. A. Star unfolding of a polytope with applications. SIAM J. Comput. 26, 1997, 1689-1713.

[10] Sharir, M., and Schorr, A. On shortest paths in polyhedral spaces. SIAM J. Comput. 15, 1986, 259-263.

[11] Yao, A. C. On constructing minimum spanning trees in k-dimensional space and related problems. SIAM J. Comput. 11, 1982, 721-736.