The Report of Delaunay's Death Has Been Greatly Exaggerated

Author name commented out

Address commented out

Abstract

Although much of the early research in position-based routing for ad-hoc networks focused on the two-dimensional Delaunay triangulation, recent work has abandoned the Delaunay triangulation because no suitable algorithm for calculating it was available. This paper presents a new algorithm for computing the Delaunay graph in *any* number of dimensions. The algorithm is proved to be self-stabilizing, which means that, from any initial state, the system's state will converge to the Delaunay graph. Because it is self stabilizing, the algorithm is suited for distributed applications where processors are unreliable and locations are changing, such as mobile ad-hoc networks in two or three dimensions.

1 Introduction

Position-based routing is a class of routing algorithms for ad-hoc networks where messages are routed to the (expected) location of the destination.[7] Position-based routing has the advantage that, as compared to other approaches to routing, it scales well as the number of nodes in the network increases. During operation, nodes learn of their location from GPS or some other location service. Nodes learn the location of distant nodes via a name-to-location mapping provided by a directory service.[13]

Early work on position-based routing, such as [3], focused on forwarding messages only over edges in the Delaunay triangulation. (See Figure 1 for an example of a Delaunay triangulation.) The Delaunay triangulation was popular because it has a number of good properties[1], including:

- **Few neighbors.** On average, a node in a Delaunay triangulation is connected to less than 6 other nodes, referred to as ***neighbors***. For randomly placed nodes[1], the chances that a node has more than 12 neighbors is less than 1 in 10 000.[8] Nodes located on a rectangular grid have at most 8 neighbors. Fewer neighbors means that less memory and processor time is used to keep track of other nodes and that fewer nodes need to be considered when making routing decisions.

- **Simple routing.** In ***greedy routing***, a node forwards a message to the neighbor that is closest to the destination. Greedy routing will deliver a message from any source to any destination on Delaunay graphs of any dimension.

- **Short paths.** For any placement of points, the Delaunay triangulation contains a path less than $\frac{4\sqrt{3}}{9}\pi \approx 2.42$ times the straight-line distance between any endpoints.[11] Greedy routing generates paths usually no more than 3 times the straight-line distance when nodes are placed randomly in a square[3], and always less than $\sqrt{2}$ times the straight-line distance for Delaunay triangulations with nodes placed on a rectangular grid.

- **Redundancy.** There exist multiple disjoint paths between all pairs of nodes.

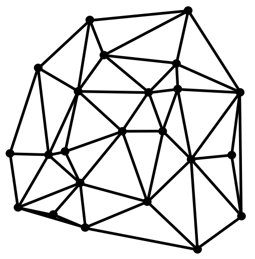

Figure 1: Delaunay triangulation (Delauny graph in 2 dimensions).

However, existing distributed algorithms for the Delaunay triangulation require some form of mutual exclusion, such as locks, to maintain a global property, such as "the unlocked portion of the graph is a triangulation".[14, 4] Since maintaining a global property in an unreliable environment is difficult (if not impossible) and recovering a lost lock is complex, most researchers have given up on using the Delaunay triangulations for position-based routing.

[1] "random" refers to a homogeneous Poisson point process.

Without the Delaunay triangulation, research has been done on using topologies other than the Delaunay triangulation and routing strategies other than greedy routing. These approaches often cannot guarantee that messages will be delivered, or resort to flooding messages, which is inefficient.[13, 7]

Currently, the only position-based routing approach that advertises to guarantee delivery without flooding messages is planar-subgraph recovery.[2, 10] Planar-subgraph recovery normally routes a message using greedy routing on the communication graph. The communication graph contains an edge between two nodes if the two nodes can directly communicate with each other. If greedy routing fails, that is, if a message is at a node which cannot directly communicate with a node that is closer to the destination, then the message enters the recovery phase.

In the recovery phase, messages are only forwarded over edges in a planar subgraph of the communication graph. This planar subgraph is usually a subset of the Gabriel graph[2, 10] or of the Delaunay triangulation[6, 12]. (The planar subgraph is able to be calculated without calculating the complete Gabriel graph or the complete Delaunay triangulation by using an assumption about the properties of the communication graph; this assumption is discussed below.) Routing on the planar subgraph is done using the "right-hand rule": the message traces counter-clockwise around a face until it reaches a node that is closer than the one where the message entered the recovery phase. At that point, the message leaves the recovery phase and returns to greedy routing on the communication graph.

Planar-subgraph recovery has four problems:

1. **Inability to find a path where one exists.** The techniques used to create the planar subgraphs assume that there exists a communication range r such that all nodes closer than r units apart are connected in the communication graph and all nodes farther than r units apart are not connected. In the real world, interference, cancellations, or obstructions can cause reception distances to vary. When these occur, the techniques can create a subgraph that is either disconnected or not planar, and prevent a path from being found.[9]

2. **Long paths.** It is easy to construct cases where the right-hand rule causes messages to circle the long way around a face — even traveling the entire circumference of the graph — before reaching their destinations.[13]

3. **Long time to detect unreachable nodes.** To detect an unreachable node, the message must completely circle a face. If the unreachable node is outside the graph, the message must completely traverse the circumference of the graph before determining that the destination is unreachable.

4. **The planar routing techniques do not work in higher dimensions.** The right-hand rule does not work in three dimensions or higher, where there is no equivalent to the counter-clockwise of two-dimensional graphs.

In this paper, I present a new algorithm for computing the Delaunay graph in any number of dimensions. (The Delaunay triangulation is the Delaunay graph in two dimensions.) This new algorithm does not use locks and does not maintain a global property and, therefore, does not have the problems of existing distributed Delaunay algorithms.

I also prove that the algorithm is self-stabilizing, that is, from any initial state, the algorithm will always reach the Delaunay graph. This means that the algorithm can recover from any transient error that throws the algorithm into an unexpected state. It also means that the algorithm will be able to handle the changing conditions caused by moving nodes.

Following those proofs, I describe the current state of an ad-hoc routing protocol that implements the algorithm. The completed protocol should solve the four listed problems of the current state-of-the-art solution, planar subgraph recovery. Preliminary results for the protocol are presented.

The layout of the paper is as follows. The next section describes some basic terms and notations. Section 3 defines the Voronoi diagram and Delaunay graph and lists some of their properties that are used in this paper's proofs. Section 4 contains a theorem that is used to prove the algorithm correct. Section 5 presents the algorithm, a proof of its correctness, and a proof that the algorithm is self-stabilizing. Section 6 describes the current state of an mobile ad-hoc routing protocol based on the algorithm. Section 7 covers preliminary results from simulations of the protocol. Finally, Section 8 summarizes the results.

2 Definitions

A ***point*** p is a ordered $\mathcal{D}$-tuple of reals $(x_p, y_p, z_p, \ldots)$. A total ordering of points is achieved using lexigraphical ordering, such that the x-coordinate is the most significant coordinate. That is, for two points p and q, $p > q$ is true if:

- $x_p > x_q$
- **or** $x_p = x_q$ **and** $y_p > y_q$
- **or** $x_p = x_q$ **and** $y_p = y_q$ **and** $z_p > z_q$
- $\cdots$

For a set of points V, $\max(V)$ is the point in the set with the greatest coordinates. Distance between points is calculated using the Euclidean distance metric. Formally, $\mathrm{d}(p,q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2 + (z_p - z_q)^2 + \cdots}$.

A graph $G = (V, E)$, consists of a set of points, V, called ***vertices***, and a set of directed edges, E. An ***edge*** is an ordered pair of vertices. An edge from vertex v to vertex w is denoted by $\overrightarrow{vw}$. If edges exist in both directions between v and w, $\overline{vw}$ is used to denote both edges.

The term ***neighborhood*** refers to the vertices directly reachable from a vertex. Formally, the neighborhood of v is $\{w | \overrightarrow{vw} \in E\}$. Note that only outbound edges contribute to the neighborhood. A vertex in the neighborhood of v is said to be a ***neighbor of*** v.

An edge ***occupies*** the points on the line segment connecting its endpoints. Formally, $\overrightarrow{vw}$ occupies $(x_v + r \cdot (x_w - x_v), y_v + r \cdot (y_w - y_v), z_v + r \cdot (z_w - z_v), \ldots)$, where $0.0 \leq r \leq 1.0$. A region of a space is ***convex*** if, for any two points in the region, an edge between the points would only occupy points inside the region.

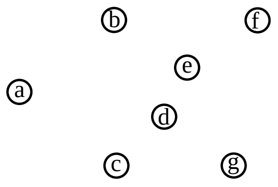

Figure 2: Point set, V.

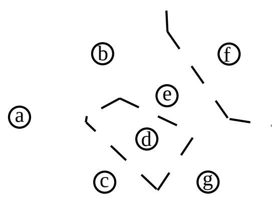

Figure 3: Voronoi regions for d and f.

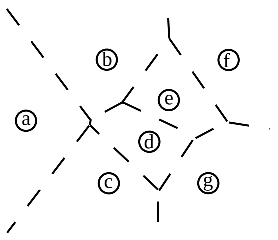

Figure 4: Voronoi diagram.

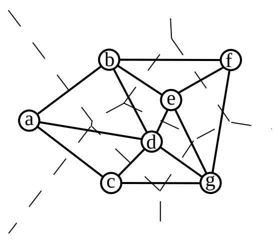

Figure 5: Delaunay graph.

3 Delaunay Graphs

In this section, I define the Voronoi Region, the Voronoi Diagram, and the Delaunay Graph, and describe their relationship to each other. I also state some of their properties that will be used as part of the proofs in this paper.

The ***Voronoi region*** of a vertex v in a set V is the set of all points in the space that are at least as close to v as any other point in V. Formally, $\mathrm{VR}(v, V) = \{p \in \Re^{\mathcal{D}} | \forall w \in V, \mathrm{d}(v,p) \leq \mathrm{d}(w,p)\}$. Figures 2 and 3 show a set of points in two dimensions and the Voronoi regions of two of the points.

Voronoi regions are convex polyhedrons (in two dimensions, convex polygons). For any region $\mathrm{VR}(v, V)$, the surfaces making up the boundary of the region are equidistant from v and another vertex in V. When a surface is equidistant from v and a vertex w, I will say that the surface is ***caused by*** the vertex w. I define the ***rightward projection*** of a point p onto $\mathrm{VR}(v, V)$ to be the point in $\mathrm{VR}(v, V)$ that has the same coordinates as p except that the x-coordinate is maximized.

If the surfaces of all the Voronoi regions of a set are unioned, the result is a graph that forms an optimal polygonal partition of the plane. This graph is called the ***Voronoi diagram***. Figure 4 shows the Voronoi diagram for the points set from Figure 2. The partition is optimal in the sense that, for any vertex v, all points where v is the closest vertex are members of $\mathrm{VR}(v, V)$.

The ***Delaunay graph*** is the dual of the Voronoi Diagram. Thus, two vertices v and w are connected by an edge in the Delaunay graph if and only if $\mathrm{VR}(v, V)$ and $\mathrm{VR}(w, V)$ share a $\mathcal{D} - 1$ dimensional surface. In two dimensions, these $\mathcal{D} - 1$ dimensional surfaces are line segments. (See Figure 5 for an example of the Delaunay graph.) The function $\mathrm{DG}(V)$ denotes the edges in the Delaunay graph of a set of vertices V.

All the proofs in this paper assume that the vertices in V are in general position. In ***general position***:

- No two vertices have the same coordinates.
- No three vertices lie on the same line.
- If $\mathcal{D} \geq 2$, no four vertices lie on the same circle.
- If $\mathcal{D} \geq 3$, no five vertices lie on the same sphere.
- If $\mathcal{D} \geq 4$, no six vertices lie on the same hypersphere.
- $\cdots$

The assumption of general position simplifies the logic in this paper. With the assumption, if the Voronoi regions of two vertices contain the same point, then the vertices are connected by an edge in the Delaunay graph. (It is easier to show that two regions share a point than to show that they share a $\mathcal{D} - 1$ dimensional surface.) A family of techniques known as "Simulation of Simplicity" or "Symbolic Perturbation" can be used to logically transform sets of vertices that are not in general position into general position.[2][5]

The proofs in this paper use the following properties of the Delaunay graph:

[2]Most papers on simulation of simplicity assume that the set of points is fixed. The author has written libraries in Java that work for a dynamic set of points. The code is available by emailing the author.

- ***Subset property***: For a set $W \subseteq V$, if $v, w \in W$ and $\overline{v\,w} \notin \mathrm{DG}(W)$ then $\overline{v\,w} \notin \mathrm{DG}(V)$.

  Proof: Adding more vertices to a vertex set can only decrease the size of the Voronoi region of each of the original vertices. Thus, $\mathrm{VR}(v, V) \subseteq \mathrm{VR}(v, W)$ and $\mathrm{VR}(w, V) \subseteq \mathrm{VR}(w, W)$. If no surface is shared by $\mathrm{VR}(v, W)$ and $\mathrm{VR}(v, W)$, then no surface can be shared by subsets of them. $\square$

- ***Equidistant property***: The edge $\overline{v\,w} \in \mathrm{DG}(V)$ if and only if there exists a point p, equidistant from v and w such that $p \in \mathrm{VR}(v, V)$.

  Proof: If p is in $\mathrm{VR}(v, V)$, the closest vertices to p must be distance $\mathrm{d}(p, v)$ away. Since $\mathrm{d}(p, v) = \mathrm{d}(p, w)$, w must also be one of the closest vertices to p and, thus, $p \in \mathrm{VR}(w, V)$. Since p is in both $\mathrm{VR}(v, V)$ and $\mathrm{VR}(w, V)$, $\overline{v\,w} \in \mathrm{DG}(V)$. The "only if" portion is obvious from the definition of DG() and VR(). $\square$

- ***Neighbor-is-closer property***[3]: For a vertex v and a point p, if $p \notin \mathrm{VR}(v, V)$, then there exists a vertex w such that $\overline{v\,w} \in \mathrm{DG}(V)$ and w is closer to p than v.

  Proof: Proof by contradiction. Let $W = \{u | \overline{v\,u} \in \mathrm{DG}(V)\}$ and assume there does not exist a vertex w in W such that w is closer to p than v. Restated, $p \in \mathrm{VR}(v, \{v\} \cup W)$. Let $U = V \setminus (\{v\} \cup W)$. Since U does not contain a vertex that has an edge to v in $\mathrm{DG}(V)$, none of the vertices in U cause a surface in the boundary of $\mathrm{VR}(v, V)$, so removing them from the set will not change the boundary of v's Voronoi region. Thus, $\mathrm{VR}(v, V) = \mathrm{VR}(v, V \setminus U)$. By the definition of U, $V \setminus U = \{v\} \cup W$. Since $p \notin \mathrm{VR}(v, V)$, $p \notin \mathrm{VR}(v, \{v\} \cup W)$. This contradicts our earlier proof that $p \in \mathrm{VR}(v, \{v\} \cup W)$. $\square$

- ***Right-linked property***: For a vertex v, if there exists a vertex in V with an x-coordinate greater than x_v, then there exists an edge $\overline{v\,w} \in \mathrm{DG}(V)$ such that $x_w > x_v$.

  Proof: Let the point p have the same coordinates as v except that x_p is arbitrarily large. As x_p goes to infinity, v cannot be the closest vertex in V to p, because there exist vertices in V with x-coordinates greater than x_v. Thus, $p \notin \mathrm{VR}(v, V)$. By the neighbor-is-closer property, we know that there exists a vertex w such that $\overline{v\,w} \in \mathrm{DG}(V)$ and w is closer to p than v. Since only vertices with x-coordinates greater than x_v can be closer to p, $x_w > x_v$. $\square$

[3]The neighbor-is-closer property can be used to show that greedy routing always succeeds on a Delaunay graph.

4 A Theorem

Before presenting my algorithm for computing the Delaunay graph, I will first present a proof that if a graph has four specific properties then the graph is a Delaunay graph. In Section 5, this theorem is used to prove the algorithm correct.

The four properties for the graph (V, E) are listed below. In them, and in the rest of this section, the notation N_v represents the neighborhood of a vertex v in the graph (V, E). Thus, $N_v = \{w | \overrightarrow{v\,w} \in E\}$.

- ***Bidirectional***: If $\overrightarrow{v\,w} \in E$ then $\overrightarrow{w\,v} \in E$
- ***Greater-linked***[4]: For all vertices $v \in V$ such that $v \neq \max(V)$, there exists an edge $\overrightarrow{v\,w} \in E$ such that $w > v$.
- ***Delaunay-triangle-closed***: For all $\overrightarrow{v\,w}, \overrightarrow{w\,u} \in E$, if $\overline{v\,u} \in \mathrm{DG}(\{v\} \cup N_v \cup \{u\})$, then $\overrightarrow{v\,u} \in E$.
- ***Locally-Delaunay***: For all v, for all $w \in N_v$, $\overline{v\,w} \in \mathrm{DG}(\{v\} \cup N_v)$.

Theorem 1. *If a graph (V, E) is bidirectional, greater-linked, Delaunay-triangle-closed, and locally-Delaunay, then the graph is a Delaunay graph.*

4.1 Outline

To prove that $E = \mathrm{DG}(V)$, the proof first shows that every edge in $\mathrm{DG}(V)$ is in E and then shows that every edge that is not in $\mathrm{DG}(V)$ is not in E.

That every edge in $\mathrm{DG}(V)$ is in E is shown by contradiction. We assume that there exists at least one edge in $\mathrm{DG}(V)$ that is missing from E. Since, by its definition, $\mathrm{DG}(V)$ contains only bidirectional edges and, by the bidirectional property, E contains only bidirectional edges, we can assume all missing edges are bidirectional. Of all missing edges, select an edge $\overline{a\,b}$ such that $a < b$ and a is maximized. That is, that a is the greatest vertex such that an edge to a greater vertex is missing. Lemma 1 shows that E must contain at least one edge that is in $\mathrm{DG}(V)$ and connects a to a vertex greater than a. Then, Lemma 2 shows that if E has one such edge, it must have all edges in $\mathrm{DG}(V)$ that connect a to vertices greater than a. Since all edges in E are bidirectional, all edges going to or from a to vertices greater than a must be present in E. This conclusion contradicts the assumption on a and proves that all edges in $\mathrm{DG}(V)$ are in E.

Lemma 3 shows that if all edges of $\mathrm{DG}(V)$ are in E, then all edges not in $\mathrm{DG}(V)$ are not in E.

[4]The greater-linked property is different from the right-linked property. The greater-linked property can be satisfied by an edge between two vertices with the same x-coordinate. The right-linked property cannot.

4.2 Lemmas

In the three lemmas, let W be all vertices in V that are greater than a. (Thus, $b \in W$.) Because of the way a is selected, if there exist $v, w \in W$ such that $\overrightarrow{v\,w} \in \mathrm{DG}(V)$, then $\overrightarrow{v\,w} \in E$.

Lemma 1. *There exists a vertex c such that $c > a$, $\overrightarrow{a\,c} \in DG(V)$ and $\overrightarrow{a\,c} \in E$.*

Proof: This proof is broken down into three cases:

1. The dimension $\mathcal{D} = 1$.
2. The dimension $\mathcal{D} \geq 2$ and the x-coordinates of all vertices in W are the same as that of a.
3. The dimension $\mathcal{D} \geq 2$ and there exists at least one vertex in W that has a greater x-coordinate than that of a.

Case 1: The dimension $\mathcal{D} = 1$.

Since the dimension is one, all the vertices in V must lie on a line. General position states that no more than two points can be on a line. Thus, $V = \{a, b\}$ with $b > a$. By the greater-linked property, E contains an edge from a to a vertex greater than a. The only vertex that matches that description is b. Thus, $\overrightarrow{a\,b} \in E$. The Delaunay graph of two vertices is a bidirectional edge between the two vertices. Thus, $\overrightarrow{a\,b} \in DG(V)$. Thus, E contains an edge from a to a vertex greater than a and that edge is also in $\mathrm{DG}(V)$.

Case 2: The dimension $\mathcal{D} \geq 2$ and the x-coordinates of all vertices in W are the same as that of a.

This case can be reduced to a case with a smaller dimension. This reduction is done by ignoring the x-dimension for all points concerned. Thus, if $\mathcal{D} = 4$ and all x-coordinates are identical, the case can be treated as a $\mathcal{D} = 3$ case by ignoring the x-coordinates and treating the y-coordinates as the most-significant coordinates.

Case 3: The dimension $\mathcal{D} \geq 2$ and there exists at least one vertex in W that has a greater x-coordinate than that of a.

To begin, the vertex a must have a neighbor in E with an x-coordinate greater than x_a. This is shown by contradiction. By the greater-linked property, E must contain an edge from a to a vertex greater than a. Let v be such a vertex. Since we have assumed that a has no neighbors with a greater x-coordinate, v must have the same x-coordinate as a. By the case assumption, there exist vertices in W, and therefore in V, that have x-coordinates that are greater than that of a and v. By the right-linked property, $\mathrm{DG}(V)$ must contain an edge from v to some vertex w such that $x_w > x_v$. Since w is greater than v and v is greater than a, both v and w are elements of W. Since both endpoints of $\overrightarrow{v\,w}$ are in W and the edge is present in $\mathrm{DG}(V)$, $\overrightarrow{v\,w} \in E$. So, in E, a has an edge to v and v has an edge to w and w has a greater x-coordinate than a. From the right-linked property, we know that $\overrightarrow{a\,w} \in \mathrm{DG}(\{a\} \cup N_a \cup \{w\})$, and, using the Delaunay triangles-closed property, conclude that $w \in N_a$. However, this contradicts our assumption that a has no neighbor in E with an x-coordinate greater than x_a. Thus, a must have a neighbor in E with an x-coordinate greater than x_a.

Let p be the rightward projection of a onto $\mathrm{VR}(a, \{a\} \cup N_a)$, that is, the point in $\mathrm{VR}(a, \{a\} \cup N_a)$ that has the same coordinates as a except that the x-coordinate is maximized. The value x_p could only be infinite if and only if a has no a neighbor with an x-coordinate greater than a's. Since we have proven that a has a neighbor with greater coordinates, p must have finite coordinates.

Since p has finite coordinates and is on the border of $\mathrm{VR}(a, \{a\} \cup N_a)$, there must exist a vertex c that is the same distance from p as a is and that $c \in N_a$. Restated, the vertex c causes the surface in $\mathrm{VR}(a, \{a\} \cup N_a)$ that contains p.

A proof by contradiction shows that the edge $\overrightarrow{a\,c} \in \mathrm{DG}(V)$. If $\overrightarrow{a\,c} \notin \mathrm{DG}(V)$, then $p \notin \mathrm{VR}(c, V)$. By the neighbor-is-closer property we know that if $p \notin \mathrm{VR}(c, V)$ then there exists a vertex in $\{w | \overline{c\,w} \in \mathrm{DG}(V)\}$ that is closer to p. Let d be such a vertex. From the way we selected p, only vertices greater than a can be closer to p. We have already assumed that the neighborhood of c in E contains all vertices in $\{w | \overline{c\,w} \in \mathrm{DG}(V)\}$ that are greater than a. Thus, if $\overrightarrow{a\,c} \notin \mathrm{DG}(V)$, c must have a neighbor d in E that is closer to p than c. Since E contains both $\overrightarrow{a\,c}$ and $\overrightarrow{c\,d}$, if it can be shown that $\overrightarrow{a\,d} \in \mathrm{DG}(\{a\} \cup N_a \cup \{d\})$ then the Delaunay-triangle-closed property can be used to prove that d is a member of N_a.

Since d is closer than a to p, we can select a point q on the line segment from p to a that is equidistant from d and a. Since p and a in $\mathrm{VR}(a, \{a\} \cup N_a)$ and Voronoi regions are convex, all points on the line segment from p to a are inside $\mathrm{VR}(a, \{a\} \cup N_a)$. Thus, $q \in \mathrm{VR}(a, \{a\} \cup N_a)$. Since d and a are equidistant from q, $q \in \mathrm{VR}(a, \{a\} \cup N_a \cup \{d\})$. By the equidistant property, we can state that $\overrightarrow{a\,d} \in \mathrm{DG}(\{a\} \cup N_a \cup \{d\})$ and, then, by the Delaunay-triangle-closed property state that $d \in N_a$.

However, if $d \in N_a$, c would not be the vertex in N_a closest to p. This contradicts the definition of c. Thus, the edge $\overrightarrow{a\,c}$ must be part of $\mathrm{DG}(V)$ and we know that E has an edge from a to at least one vertex greater than a such that the edge is in $\mathrm{DG}(V)$. $\square$

Lemma 2. *If there exists a vertex c such that $c > a$, $\overrightarrow{a\,c} \in DG(V)$ and $\overrightarrow{a\,c} \in E$, then for all v such that $v > a$ and $\overrightarrow{a\,v} \in DG(V)$, then $\overrightarrow{a\,v} \in E$.*

Note: The proof of this lemma is broken into two parts. To understand the purpose of the parts, the best analogy is that of infection. To show that a population is infected, we can show that the infection passes between adjacent people and that there exists a sequence of adjacent people from an infected person to every person in the population. In the analogy, the "population" is the set of a's neighbors in the Delaunay graph that are greater than a, a vertex is "infected" if is has an edge from a in E, and two vertices are "adjacent" when the surfaces they cause on $\mathrm{VR}(a, V)$ share a point.

Note that it is possible, although complicated, to transform this proof into a proof by induction on the length of the sequence of adjacent people.

Proof: For the case where $\mathcal{D} = 1$, Lemma 1's Case 1 is sufficient to show that all edges in $\mathrm{DG}(V)$ are present in E. The proof for cases where the dimension $\mathcal{D} \geq 2$ is broken into two parts:

1. If there exists a vertex v such that $v > a$, v causes a surface on $\mathrm{VR}(a, V)$, and $\overrightarrow{a\,v} \in E$ and there exists a vertex w such that $w > a$ and w causes a surface on $\mathrm{VR}(a, V)$ that shares a point with the surface caused by v, then $\overrightarrow{a\,w} \in E$.

2. For any vertices v and w such that both are greater than a and both cause a surface on $\mathrm{VR}(a, V)$, then there exists a sequence of surfaces on $\mathrm{VR}(a, V)$ caused only by vertices greater than a such that the first is caused by v, the last is caused by w, and each consecutive pair shares a point.

Part 1:

If there exists a vertices v and w such that they both cause a surface on $\mathrm{VR}(a, V)$ and those surfaces share a point p, then a, v and w are the closest points in V to that point p and, as a result, v and w must be connected by an edge in $\mathrm{DG}(V)$. We know that if an edge is in $\mathrm{DG}(V)$ and both endpoints are greater than a, then the edge is in E. Thus, $\overrightarrow{v\,w} \in E$.

The Delaunay-triangles-closed property states that if $\overrightarrow{a\,v} \in E$, $\overrightarrow{v\,w} \in E$ and $\overrightarrow{a\,w} \in \mathrm{DG}(\{a\} \cup N_a \cup \{w\})$, then $\overrightarrow{a\,w} \in E$. We have assumed $\overrightarrow{a\,v} \in E$ and shown $\overrightarrow{v\,w} \in E$. Since w causes a surface in $\mathrm{VR}(a, V)$, there must exists points, like p, such that no vertex in V is closer than a and w. Since $\{a\} \cup N_a \cup \{w\}$ is a subset of V, $\{a\} \cup N_a \cup \{w\}$ cannot contain a vertex closer to p, and, thus, $\overrightarrow{a\,w} \in \mathrm{DG}(\{a\} \cup N_a \cup \{w\})$. As a result, we conclude that $\overrightarrow{a\,w} \in E$.

Part 2:

Now, we consider two vertices v and w such that v and w are both greater than a and both cause a surface in $\mathrm{VR}(a, V)$. I need to show that there exists a sequence of surfaces on $\mathrm{VR}(a, V)$ caused only by vertices greater than a such that the first is the one caused by v, the last is caused by w, and each consecutive pair shares a point.

Assume that there exists a vertex in W with an x-coordinates greater than the x_a. If this is not the case, the problem can be treated as one of smaller dimension, as done in Lemma 1's Case 2. Let p be the rightward projection of a onto $\mathrm{VR}(a, V)$. The value x_p could only be infinite if and only if V contains no vertex with an x-coordinate greater than a's. Since there exists a vertex in W and, therefore, in V with an x-coordinate larger than x_a, p must have finite coordinates.

Let q and r be any two points that lie on the surface of $\mathrm{VR}(a, V)$ that are caused by v and w, respectively. The line segments $\overline{q\,p}$ and $\overline{p\,r}$ must occupy points only inside $\mathrm{VR}(a, V)$, because all three points lie inside $\mathrm{VR}(a, V)$ and Voronoi regions are convex.

Consider the rightward projection of the points occupied by the line segments $\overline{q\,p}$ and $\overline{p\,r}$ onto $\mathrm{VR}(a, V)$. This projection lies across the surfaces caused by vertices that have edges to a in $\mathrm{DG}(V)$. As the projection crosses between surfaces, the surfaces share a point. What remains to be proven is that the rightward projections of p and q are in the surfaces caused by v and w, respectively, and that every point of the projection lies in a surface caused by a vertex greater than a.

Since v is greater than a, x_v is either greater than or the same as x_a. If $x_v > x_a$, then q is equal to the rightward projection of q onto $\mathrm{VR}(a, V)$. If the rightward projection had a higher x-coordinate than q, it would be closer to v than to a and, therefore, outside of $\mathrm{VR}(a, V)$, contradicting the definition of an rightward projection. For the case where $x_v = x_a$, any change in an x-coordinate does not change which of a or v is closer. Since q and the rightward projection of q only differ in the x-coordinate, the rightward projection of q onto $\mathrm{VR}(a, V)$ is still equidistant from a and v and, therefore, on the surface of $\mathrm{VR}(a, V)$ caused by v. The proof that rightward projection of r is in the surface of $\mathrm{VR}(a, V)$ caused by w is similar.

Now, I must show that each point of the rightward projection of the points in the line segments $\overline{q\,p}$ and $\overline{p\,r}$ onto $\mathrm{VR}(a, V)$ is contained in a surfaces caused by a vertex greater than a. Let s be any point on the rightward projection. Since s has the maximum x-coordinate that is still in $\mathrm{VR}(a, V)$, there must exist a vertex d which, if s had any higher x-coordinate, it would lie closer to d than a and, therefore, be outside of $\mathrm{VR}(a, V)$. Since s is equidistant from a and d, d must cause the surface on $\mathrm{VR}(a, V)$ that contains d. Because a point with s's coordinates except a higher x-coordinate would be closer to d than a, d must have a higher x-coordinate than x_a. If $x_d > x_a$, then $d > a$. So, all points of the projection lie in surfaces caused by vertices greater than a.

Thus, there exists a sequence of surfaces that contain points of the rightward projections of the points occupied by line segments $\overline{qp}$ and $\overline{pr}$ onto $\mathrm{VR}(a, V)$ that constitute a sequence of surfaces on $\mathrm{VR}(a, V)$ such that the first surface is caused by v, the last surface is caused by w, all are caused by vertices greater than a, and each consecutive pair in the sequence share a point. $\square$

Lemma 3. *If $E \supseteq DG(V)$ then $E = DG(V)$.*

Proof: Assume there exists an edge $\overrightarrow{vw}$ that is in E, but not in $\mathrm{DG}(V)$.

By its definition, the Delaunay graph contains an edge between two vertices only if the Voronoi regions of the two vertices share a border. Since $E \supseteq \mathrm{DG}(V)$, $\mathrm{VR}(v, \{v\} \cup N_v) = \mathrm{VR}(v, V)$ because N_v contains all the points that cause the border of the Voronoi region of v in V.

Because Delaunay graphs have the equidistant property and $\overrightarrow{vw} \notin \mathrm{DG}(V)$, we know that the set of points equidistant from v and w does not intersect $\mathrm{VR}(v, V)$. Since $\mathrm{VR}(v, \{v\} \cup N_v) = \mathrm{VR}(v, V)$, we know that the set of points equidistant from v and w does not intersect $\mathrm{VR}(v, \{v\} \cup N_v)$. From this, using the equidistant property, we can conclude that $\overrightarrow{vw} \notin \mathrm{DG}(\{v\} \cup N_v)$.

By Property 4, we know that if $\overrightarrow{vw} \notin \mathrm{DG}(\{v\} \cup N_v)$ then $w \notin N_v$. This contradicts our assumption that $\overrightarrow{vw} \in E$ and we can conclude that $E = \mathrm{DG}(V)$. $\square$

5 Algorithm

In this section, I present a new distributed algorithm for computing the Delaunay graph and prove that it is both correct and self-stabilizing. The algorithm is described as running on a shared-memory machine, rather than a message passing machine, in order to simplify the notations in the proofs. The algorithm also works on a simpler problem than ad-hoc routing. In it, nodes do not join or leave the graph and nodes do not move. In the next section, I will address how to make the algorithm run in a message-passing environment where nodes join, leave, and change location.

I begin the section by defining self stabilization. That is followed by a description of the model of distributed computer and the program notation. Then I present the algorithm and prove that it is correct — that it always halts at the Delaunay graph. The section ends with the proof that the algorithm is self stabilizing.

5.1 Self Stabilization

A distributed system consists of a set of processors whose local states, combined together, make the global state of the system. A distributed algorithm coordinates changes in the local states in order to control the global state of the system. At any time, we can divide the set of all global states into two classes — "safe" and "unsafe" — based on whether or not an algorithm is fulfilling its requirements at that time. A ***self-stabilizing*** algorithm is one where, if the global state is ever unsafe, it is guaranteed to return to a safe state within a finite amount of time and, once in a safe state, remain in a safe state forever.[15]

The value in using a self-stabilizing algorithm is that the algorithm can handle transient faults. If data gets garbled or if a processor resets, the system may enter an unsafe state for a period of time, but it is guaranteed to return to a safe state and correct operation within a finite amount of time.

For the algorithm presented here, the only safe state is the Delaunay graph. All other states are unsafe.

5.2 Computational Model

The distributed computational model used in the presentation of this algorithm and its proof of self-stabilization is a fully-connected shared-memory computer. Each processor has its own local memory. Processors are able to read any other processor's memory, but can only write to their own memory. Section 6 will discuss how to transform this shared-memory algorithm into a message-based one.

5.3 Algorithm

I will begin by describing the local state of each processor and how the Delaunay graph is represented in the global state. Following that, I will discuss the language in which the algorithm is written. Lastly, I will present the algorithm.

The algorithm's input is a set of vertices V. For each of the n vertices in V there is a processor. The processors are labeled $P_1, P_2, \ldots, P_n$. Each processor's state is initialized with the coordinates of one vertex. The vertex associated with processor P_i is denoted by $P_i.\gamma$. The value of γ is immutable — it does not change during the execution of the algorithm. Section 6 will discuss the effects of letting γ change, which is equivalent to letting nodes move in an ad-hoc system.

The second piece of state that a processor stores is a vertex $P_i.m$ that is greater than the vertex $P_i.\gamma$, unless $P_i.\gamma = \max(V)$. In the simplest case, $P_i.m = \max(V)$ for all i. The value of $P_i.m$ is immutable and is initialized before execution begins. In practice, the value for $P_i.m$ may be calculated by another algorithm and used as an input to this algorithm. Section 6 will describe a simple algorithm to compute $P_i.m$.

The third and final piece of state that each processor stores is its vertex's neighborhood in a local variable N, denoted either $P_i.N$ or, if it has been shown that $v = P_i.\gamma$,

N_v. The neighborhoods are variable and change during the execution of the algorithm. However, the neighborhoods have an invariant. The ***neighborhood invariant*** states that if $v \in P_i.N$ then $\overrightarrow{P_i.\gamma\, v} \in \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N)$. Neighborhoods are initialized to any set of vertices that satisfy the neighborhood invariant (e.g., the empty set, which always satisfies the neighborhood invariant). I will say that a neighborhood is ***corrupt*** if it contains a point not in V. Neighborhoods will not become corrupt during the normal execution of the program, but may as a result of an error.

The global state is denoted as set of edges E that is defined in terms of the Ns. That is, $\overrightarrow{v\,w} \in E$ if and only if there exists a processor P_i such that $P_i.\gamma = v$ and $w \in P_i.N$. **Note, that this is the opposite of Section 4 where the set E was given and the Ns were derived from E's edges.**

The goal of the algorithm is to have the graph (V, E) be a Delaunay graph.

From Section 4, we know if a graph is bidirectional, greater-linked, Delaunay-triangle-closed, and locally-Delaunay then it is a Delaunay graph. The algorithm is based on this theorem. The neighborhood invariant ensures that (V,E) is always locally-Delaunay. The other three properties are enforced by the operations of the algorithm. A high-level view of the algorithm would be:

```
If graph is not bidirectional,
    then add edges to make the graph bidirectional.
If the graph is not greater-linked,
    then add edges to make the graph greater-linked.
If the graph has two sides of a Delaunay-triangle,
    then add the edge to make the Delaunay-triangle.
```

This view is somewhat inaccurate. If the algorithm only added edges to a graph, it would likely result in a superset of the Delaunay graph, not the Delaunay graph. Also, after a new edge is added, the neighborhood invariant may not hold. Thus, after any edge is added to the graph, the algorithm has to remove edges that are not in $\mathrm{DG}(V)$ and make sure the neighborhood invariant still holds.

The algorithm removes edges not in $\mathrm{DG}(V)$ by using the subset property of Delaunay graphs. This property states that if an edge is not present in the Delaunay graph of a subset of vertices, then the edge is not present in the Delaunay graph of the set. Consider the case of the algorithm attempting to add the edge $\overrightarrow{v\,w}$ to the graph, which means that a processor P_i where $P_i.\gamma = v$ is adding w to its neighborhood. The algorithm removes all edges not in the Delaunay graph of V by removing from the neighborhood all vertices u such that $\overrightarrow{v\,u} \notin \mathrm{DG}(\{v\} \cup P_i.N \cup \{w\})$. From the subset property, we know that the edges not in the Delaunay graph of $\{v\} \cup P_i.N \cup \{w\}$ are not in $\mathrm{DG}(V)$. In addition to removing edges not in the Delaunay graph of V, this computaion always generates a neighborhood that obeys the neighborhood invariant.

Before presenting a formal version of the algorithm, I introduce the syntax and semantics used in presentation of the formal version. The notation uses a guarded loop written in this fashion:

```
do { forever }
        $G_1 \to A_1$
[]      $G_2 \to A_2$
[]      $G_3 \to A_3$
⋮       ⋮
od
```

The term G_i is a boolean function called a ***guard***. The corresponding A_i is an operation, called an ***action***. An action will only executed if its corresponding guard evaluates to true. (This is similar to *if . . . then . . .* semantics in most languages.) The ***do . . . od*** structure loops forever and, for every iteration, may execute an action if its guard evaluates to true. The *do . . . od* structure will always execute an action if one or more guards evaluate to true. However, if multiple guards evaluate to true, the semantics does not specify which of the actions is executed.

Table 1 contains the code for the algorithm. In the code, the following short hand is used:

- $\alpha \equiv \exists v \in P_i.N$ such that $v \notin V$
- $\beta \equiv \exists v \in P_i.N$ such that $\overrightarrow{P_i.\gamma\, v} \notin \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N)$

The term α is true when the neighborhood has been corrupted to contain a point not in V. The term β is true when the neighborhood invariant does not hold.

For the proofs that the algorithm halts and is correct, I will assume that the neighborhood invariant always holds and that the state is not corrupt. Thus, both α and β will be false.

5.4 Algorithm Halts

This section begins with the definition of a state transition. This is followed by an outline of the proof that the algorithm, starting at a non-corrupt state where the neighbor invariant holds, will always halt. Following the outline are the lemmas comprising the proof.

A ***state transition*** occurs whenever any processor or any set of processors synchronously execute an action. I will use E^0 to denote the initial state of the system and E^j to denote the global state of the system after j state transitions. Similarly, I will use $P_i.N^0$ to denote the initial neighborhood at processor P_i and $P_i.N^j$ to denote it after j state transitions. Thus, $P_i.N^j \equiv \{v | \overrightarrow{P_i.\gamma\, v} \in E^j\}$.

The proof that the algorithm halts is broken down into the following lemmas:

```
do { forever at $P_i$ }
    { $G_1 \rightarrow A_1$: make edges bidirectional }
    $\neg\alpha$ and $\neg\beta$
    and $\exists j$ such that $P_i.\gamma \in P_j.N$
    and $P_j.\gamma \notin P_i.N$
    and $\overrightarrow{P_i.\gamma\, P_j.\gamma} \in \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N \cup \{P_j.\gamma\})$
        $\rightarrow P_i.N := \{w | \overrightarrow{P_i.\gamma\, w} \in \mathrm{DG}(W)$
            where $W = \{P_i.\gamma\} \cup P_i.N \cup \{P_j.\gamma\}\}$

    { $G_2 \rightarrow A_2$: make the graph greater-linked }
[]  $\neg\alpha$ and $\neg\beta$
    and $\nexists v \in P_i.N$ such that $v > P_i.\gamma$
    and $P_i.m > P_i.\gamma$
        $\rightarrow P_i.N := \{w | \overrightarrow{P_i.\gamma\, w} \in \mathrm{DG}(W)$
            where $W = \{P_i.\gamma\} \cup P_i.N \cup \{P_i.m\}\}$

    { $G_3 \rightarrow A_3$: make Delaunay triangles }
[]  $\neg\alpha$ and $\neg\beta$
    and $\exists j, k$ such that $P_j.\gamma \in P_i.N$
    and $P_k.\gamma \notin P_i.N$
    and $\overrightarrow{P_i.\gamma\, P_k.\gamma} \in \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N \cup \{P_k.\gamma\})$
        $\rightarrow P_i.N := \{w | \overrightarrow{P_i.\gamma\, w} \in \mathrm{DG}(W)$
            where $W = \{P_i.\gamma\} \cup P_i.N \cup \{P_k.\gamma\}\}$

    { $G_4 \rightarrow A_4$: remove corruption, restore invariant}
[]  $\alpha$ or $\beta$
        $\rightarrow P_i.N := \emptyset$
od
```

Table 1: The algorithm. See Section 5.3 for definitions of α and β.

4. If guard G_2 is true at processor P_i in state E^j, then $\overline{P_i.\gamma\, P_i.m} \in DG(\{P_i.\gamma\} \cup P_i.N^j \cup \{P_i.m\})$.

5. If action A_1, A_2 or A_3 is executed at processor P_i in state E^j, then there exists a vertex $w \in P_i.N^{j+1}$ such that $w \notin P_i.N^j$.

6. At any processor P_i, for all vertices w and all states E^j, if $w \in P_i.N^j$ and $w \notin P_i.N^{j+1}$ then, for all states k where $k > j$, $w \notin P_i.N^k$.

The first lemma, Lemma 4, is used in the proof of the last two lemmas. Lemma 5 is used to show that, every time an action is executed, a vertex is added to a neighborhood. In Lemma 6, we see that once a vertex is removed from a neighborhood, it can never be added again. This means that each processor can only execute n actions. Since each of the n processors can only execute n actions, the algorithm must halt within n^2 state transitions. Note that the proof of n actions holds for any processor where the neighborhood invariants hold and no state is corrupt.

Lemma 4. *If guard G_2 is true at processor P_i in state E^j, then $\overline{P_i.\gamma\, P_i.m} \in DG(\{P_i.\gamma\} \cup P_i.N^j \cup \{P_i.m\})$.*

Proof: Let the point p be $(r^{\mathcal{D}}, r^{\mathcal{D}-1}, r^{\mathcal{D}-2}, \ldots)$ where r is arbitrarily large. From an expansion of the distance frunction d() it is easy to see for two vertices a and b that if $a < b$ then $\mathrm{d}(a,p) < \mathrm{d}(b,p)$ as r goes to infinity.

If guard G_2 is true, $P_i.\gamma$ is greater than every vertex in $P_i.N^j$. Thus, we know that $P_i.\gamma$ closest to p and that $p \in \mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j)$. Since $P_i.m > P_i.\gamma$, we know that p is closer to $P_i.m$ than $P_i.\gamma$. Since p is closer to $P_i.m$, we can select a point q on the line segment from p to $P_i.\gamma$ that is equidistant from $P_i.m$ and $P_i.\gamma$. Because Voronoi regions are convex, all points on the line segment from p to v are inside $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j)$. Since q is on the line segment, we know that q is inside $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j)$ and that $P_i.\gamma$ is at least as close to q as any vertex in $P_i.N^j$. Since $P_i.m$ is the same distance as $P_i.\gamma$ from q, we know that q is in $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{P_i.m\})$. Since q is equidistant from $P_i.m$ and $P_i.\gamma$, we can use the equidistant property to conclude that $\overline{P_i.\gamma\, P_i.m} \in \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^j \cup \{P_i.m\})$. □

Lemma 5. *If action A_1, A_2 or A_3 is executed at processor P_i in state E^j, then there exists a vertex $w \in P_i.N^{j+1}$ such that $w \notin P_i.N^j$.*

Proof: In order for action A_1 to be executed at processor P_i, guard G_1 must be true at that processor. From the statement of guard, it is obvious that the vertex which is described in the guard as "$P_j.\gamma$" is not in $P_i.N^j$ and will be included in $P_i.N^{j+1}$.

In order for action A_2 to be executed at processor P_i, guard G_2 must be true at that processor. From the guard, we can conclude that $P_i.m \notin P_i.N^j$ because $P_i.N^j$ contains no vertices greater than $P_i.\gamma$. From Lemma 4 we can conclude that if action A_2 is executed, then $P_i.m$ will be included in $P_i.N^{j+1}$.

In order for action A_3 to be executed at processor P_i, guard G_3 must be true at that processor. From the statement of guard, it is obvious that the vertex described in the guard as "$P_k.\gamma$" is not in $P_i.N^j$ and will be included in $P_i.N^{j+1}$. □

Lemma 6. *At any processor P_i, for all vertices w and all states E^j, if $w \in P_i.N^j$ and $w \notin P_i.N^{j+1}$ then, for all states k where $k > j$, $w \notin P_i.N^k$.*

Proof: For this proof, let $v = P_i.\gamma$.

We begin by proving that if $w \in P_i.N^k$ where $k > j$, then there must exist a point that is equidistant from

$P_i.\gamma$ and w that is inside $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{k-1} \cup \{w\})$. Assume that w is added to $P_i.N$ in state E^k where $k > j$. This could only happen if an action executed that added w to $P_i.N$. For the action to execute, the guard must have been true. For the guard to be true, $\overrightarrow{P_i.\gamma\, w}$ must be an element of $\mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^{k-1} \cup \{w\})$. (This is obvious for guards G_1 and G_3 and was shown in Lemma 4 to be a consequence of guard G_2.) By the equidistant property, we know that for $\overrightarrow{P_i.\gamma\, w}$ to be an element of $\mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^{k-1} \cup \{w\})$, there must exist a point that is equidistant from $P_i.\gamma$ and w and is inside $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{k-1} \cup \{w\})$.

Since w is only added if there exists a point that is equidistant from $P_i.\gamma$ and w that is inside $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{k-1} \cup \{w\})$, I can prove the lemma by showing that $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{k})$ (which is a superset of $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{k} \cup \{w\})$) does not contain any point equidistant from $P_i.\gamma$ and w for $k > j$. This is proved by induction. The base case is that $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{j+1})$ does not contain any points equidistant from $P_i.\gamma$ and w. The inductive case is that any point not in $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{l})$ is not in $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{l+1})$ for any l. (The inductive case is more general than is necessary, but is included because it provides an intuitive understanding of why the algorithm halts.)

Base case: There does not exist a point p such that $\mathrm{d}(P_i.\gamma, p) = \mathrm{d}(w, p)$ and $p \in \mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{j+1})$.

Since $\overrightarrow{P_i.\gamma\, w} \in E^j$, we know that $w \in P_i.N^j$ and, since $\overrightarrow{P_i.\gamma\, w} \notin E^{j+1}$, we know that $w \notin P_i.N^{j+1}$. Every action contains the same operation to generate $P_i.N^{j+1}$, so we know that $P_i.N^{j+1} = \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$, where $\overrightarrow{P_i.\gamma\, a}$ is the edge added by the action. Since $w \notin P_i.N^{j+1}$, we know that $\overrightarrow{P_i.\gamma\, w} \notin \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$, and can conclude that there do not exist any points equidistant from $P_i.\gamma$ and w in $VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$.

I will now show that $VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\}) = VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{j+1})$. Let B be the set of all vertices in $P_i.N^j$, that are not in $P_i.N^{j+1}$. (Note that B contains w.) By the definition of N^{j+1}, there is no edge from $P_i.\gamma$ to any vertex in B in $\mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$. Since there is no edge from $P_i.\gamma$ to any vertex in B, the boundary of $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$ is made of surfaces caused by vertices not in B. As a result, if all the vertices in B are removed from the set, the Voronoi region will remain the same. Thus, $VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\}) = VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\} \backslash B)$, and from the definition of B it is easy to conclude that $P_i.N^{j+1} = P_i.N^j \cup \{a\} \backslash B$.

Since there are no points equidistant from $P_i.\gamma$ and w in $VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\})$ and $VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^j \cup \{a\}) = VR(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{j+1})$, we can conclude that the base case is true.

Inductive case: For any integer l, $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{l+1}) \subseteq \{P_i.\gamma\} \cup P_i.N^{l})$.

First, if no vertex is added to $P_i.N$ in the state transition from l to $l+1$, then $P_i.N^l = P_i.N^{l+1}$ and the case is true. The rest of this proof will concentrate on the case where a vertex a is added to $P_i.N$. Thus, $P_i.N^{l+1} = \mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^l \cup \{a\})$.

It is obvious from the definition of a Voronoi region that adding vertices to the set can only make the Voronoi region smaller. Thus, we know that $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^l \cup \{a\}) \subseteq \mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^l)$. Let B be the set of vertices in $P_i.N^l$ that are not present in $P_i.N^{l+1}$. By the definition of N^{l+1}, there is no edge from $P_i.\gamma$ to any vertex in B in $\mathrm{DG}(\{P_i.\gamma\} \cup P_i.N^l \cup \{a\})$. Since there is no edge from $P_i.\gamma$ to any vertex in B, the boundary of $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^l \cup \{a\})$ is made of surfaces caused by vertices not in B. As a result, if all the vertices in B are removed, the Voronoi region will remain the same. Thus, $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{l+1}) = \mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^l \cup \{a\})$, and we can conclude that $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^{l+1})$ is a subset of $\mathrm{VR}(P_i.\gamma, \{P_i.\gamma\} \cup P_i.N^l)$. $\square$

5.5 Algorithm is Correct

The proof that the algorithm always halts at the Delaunay graph is based on Theorem 1 from Section 4. The theorem states that if the graph is bidirectional, greater-linked, Delaunay-triangle-closed, and locally-Delaunay, then the graph is a Delaunay graph.

The proof that the algorithm is correct, that is, always halts at the Delaunay graph, is broken into these lemmas:

7. If the graph (V, E^j) is not bidirectional, then there exists a processor P_i, where a guard evaluates to true in state E^j.

8. If the graph (V, E^j) is not greater-linked, then there exists a processor P_i, where a guard evaluates to true in state E^j.

9. If the graph (V, E^j) is not Delaunay-triangle-closed, then there exists a processor P_i, where a guard evaluates to true in state E^j.

The neighborhood invariant ensures that the graph is always locally-Delaunay. If, in any state, any of the other properties does not hold, then one of the lemmas must apply and a guard must evaluate to true at (at least) one processor. If a guard evaluates to true, the algorithm cannot have halted. Thus, if the graph is not the Delaunay graph, the algorithm cannot have halted. Since we know that the algorithm halts within n^2 steps, when it halts the state must be the Delaunay graph. Note that this proof holds for any initial state where the neighborhood invariants hold and no state is corrupt.

Lemma 7. *If the graph (V, E^j) is not bidirectional, then there exists a processor P_i, where a guard evaluates to true in state E^j.*

Proof: If the graph is not bidirectional, there exists two vertices v and w such that $\overrightarrow{vw} \in E^j$ and $\overrightarrow{wv} \notin E^j$.

This proof is broken into two cases.

Case 1: $\overline{vw} \in \mathrm{DG}(\{w\} \cup N_w^j \cup \{v\})$

In this case, guard G_1 is true at the processor P_i where $P_i.\gamma = w$.

Case 2: $\overline{vw} \notin \mathrm{DG}(\{w\} \cup N_w^j \cup \{v\})$

Since by the neighborhood invariant we know that $\overline{vw} \in \mathrm{DG}(\{v\} \cup N_v^j)$, we know from the equidistant property that there must exist at least one point p such that p is equidistant from v and w and inside $\mathrm{VR}(v, \{v\} \cup N_v^j)$. However, by the case assumption, we know that $\mathrm{VR}(w, \{w\} \cup N_w^j \cup \{v\})$ does not contain any point that is equidistant from v and w and, therefore, does not contain p. Since $p \notin \mathrm{VR}(w, \{w\} \cup N_w^j \cup \{v\})$, by the neighbor-is-closer property there must exist some vertex in N_w^j that is closer to p than v or w. Let a be such a vertex.

Let q be a point that is occupied by a line segment from p to v such that q lies equidistant from a and v. Because Voronoi regions are convex, $q \in \mathrm{VR}(v, \{v\} \cup N_v^j)$ and thus v is closer to q than any vertex in N_v^j. From the facts that q is equidistant from a and v and inside $\mathrm{VR}(v, \{v\} \cup N_v^j \cup \{a\})$, using the equidistant property, we can conclude that $\overrightarrow{va} \in \mathrm{DG}(\{v\} \cup N_v^j \cup \{a\})$.

Because, in E^j, w is a neighbor of v and a is a neighbor of w and $\overrightarrow{va} \in \mathrm{DG}(\{v\} \cup N_v^j \cup \{a\})$, we know that guard G_3 is true at processor P_i where $P_i.\gamma = v$. □

Lemma 8. *If the graph (V, E^j) is not greater-linked, then there exists a processor P_i, where a guard evaluates to true in state E^j.*

Proof: If the graph is not greater-linked, then there must exist a vertex (not $\max(V)$) that does not have a connection to a vertex with greater coordinates. At this vertex, guard G_2 must be true. □

Lemma 9. *If the graph (V, E^j) is not Delaunay-triangle-closed, then there exists a processor P_i, where a guard evaluates to true in state E^j.*

Proof: Any three vertices that cause the Delaunay-triangle-closed property not to hold, will also cause Guard G_3 to be true at one of them. □

5.6 Algorithm is Self-Stabilizing

Having shown that the algorithm always halts at the Delaunay graph for any initial state that is not corrupt and satisfies the neighborhood invariant, I will now show that the algorithm is self stabilizing.

To prove that an algorithm is self-stabilizing, two things must be proved: convergence and closure.[15] ***Convergence*** is the property that, from an arbitrary starting state, the algorithm will reach a safe state within a finite amount of time. ***Closure*** is the property that once the algorithm enters a safe state, it will not enter an unsafe state.

The algorithm converges. It should be obvious from the construction of the algorithm that if a processor's state is ever corrupted or if the invariant no longer holds at a processor, then the processor's neighborhood is reset to the empty set. Until the neighborhood is reset to the empty set, the processor will perform no other actions. From the proof of Section 5.4, we know that a processor which has a neighborhood that is not corrupt and obeys the neighborhood invariant will execute at most n actions. We can now state that any processor with a neighborhood that is corrupt or does not obey the invariant executes at most $n + 1$ actions: one to reset the neighborhood to the empty set and n "normal" transitions. Since the algorithm cannot halt with a corrupt state or a neighborhood that violates the invariant, we conclude that the algorithm will always return to the Delaunay graph within $n(n + 1)$ state transitions. Thus, the full algorithm does show convergence.

The algorithm also shows closure. Once the graph is in a safe state — the Delaunay graph — the algorithm does not change state. We know this because we have seen that the algorithm must halt and, in every state that is not the Delaunay graph, the algorithm has not halted. Thus, once the state has reached the Delaunay graph, it will not make any transitions.

6 Protocol

The major contribution of this paper is the algorithm and the proof of its correctness. I will use the rest of this paper to describe a mobile ad-hoc routing protocol based on the algorithm and some preliminary results for simulations of the protocol.

I will use ***algorithm*** to refer to the shared-memory computation presented in Section 5 and ***protocol*** to refer to the message-based computation presented in this section.

6.1 Mapping the algorithm to the protocol

The protocol has to handle a number of new issues, because the algorithm presented in Section 5 made a number of assumptions that do not hold in an ad-hoc environment. First, the protocol must use message passing, not shared memory. Second, the protocol's nodes can send messages directly to only a limited number of nodes, not to all of them. Third, the protocol's nodes are not static; they can move, join the network, and leave the network. Lastly, the protocol cannot assume away the problem of calculating $P_i.m$ for each node.

It is trivial to map the shared-memory operations onto message-passing operations. The algorithm contains read and writes of local memory and reads of remote memory. For reads and writes of local memory, nothing changes. For reads of remote memory, they are replaced by a ReadRequest message and a ReadReply message. (In the protocol, the ReadReply message may be delayed to prevent sending multiple copies of the same message within a short period of time.)

In the algorithm, we assumed every node could communicate with (i.e., read the remote memory of) every other node. In the ad-hoc environment, nodes cannot send a message directly to any other node; some are out of transmission range. The protocol solves this by using source-routing. Each node maintains an explicit hop-by-hop path to each of its neighbors. (These source-routed paths are expected to only contain a few hops.)

In the algorithm, the set of nodes was static and node positions did not change. In the ad-hoc environment, nodes can join and leave the network, as well as move about. This is actually not a problem, since the algorithm is self-stabilizing. Any out-of-date information is viewed as corrupt and will be purged by the algorithm and by the soft-state mechanisms.

Lastly, the algorithm assumed away the problem of calculating $P_i.m$. The current incarnation of the protocol does not calculate $P_i.m$ and, therefore, does not stabilize to a Delaunay triangulation in some cases. In Section 6.4, I will discuss how the protocol can be augmented to handle those cases.

6.2 State

In the protocol, each node has three pieces of state.

The first piece of state is the node's coordinates. This is assumed to be gotten from an external source, such as a GPS receiver.

The second piece of the state is the node's neighborhood. For each neighbor, the node stores the neighbor's network (MAC) address, its last known location, a source route to the neighbor, and the time that a message was last received from the neighbor. This information is soft state and if a message has not be received from the neighbor in the last $T_{timeout}$ seconds, the neighbor's entry is deleted.

The third and final piece of state kept is the candidate neighbor information. The ***candidate neighbor*** is the neighbor's neighbor that is closest. The node tries to add the lone candidate neighbor to its neighborhood every $T_{heartbeat}$ seconds. (This could be done to more nodes and more often, but limiting it to one node every $T_{heartbeat}$ seconds prevents flooding the network.) For the candidate neighbor information, the node stores the candidate's network address, its last known location, and a source route to it.

6.3 Events

The state is changed by five types of events: a timer expiring, a message arriving, or the node joining, moving, or leaving.

6.3.1 Timer expires

Every $T_{heartbeat}$ seconds, the protocol's timer expires. At that point, the node deletes stale information. That is, the node deletes every neighbor from whom a message hasn't been received in $T_{timeout}$ seconds. (For the simulations, $T_{heartbeat}$ was set to 2 seconds and $T_{timeout}$ to 10 seconds.) Next, the node broadcasts a ReadRequest message and sends source-routed ReadRequest messages to any neighbors that are more than one hop away. Lastly, the node sends a source-routed RouteRequest message to the candidate neighbor, if there is one. After that message is sent, the candidate neighbor information is cleared to make way for a new candidate neighbor.

6.3.2 Message Arrives

There are two types of messages: ReadRequest and ReadReply. They have the same contents and are treated exactly the same, except in one regard: The ReadRequest message will, in some cases, cause a ReadReply message to be sent; the ReadReply message will never cause a ReadReply to be sent.

The ReadRequest and ReadReply messages contain the sender's coordinates, a complete copy of the sender's neighborhood, and a source-route from the source to the destination. The protocol assumes that the underlying layer is bidirectional, and that a source route from node a to node b can be reversed to generate a source route from b to a.

When a message is received at its destination, the receiver checks if the sender is currently a neighbor. If the sender was a neighbor and its locations has not changed, the receiver updates the time a message was last received from the sender.

If the message's sender was not a neighbor, or if the sender was a neighbor and its location has changed, the receiver evaluates if the sender should be a neighbor. This is done the same as in the algorithm — the sender should be a neighbor if the Delaunay triangulation of the locations of the sender, the receiver, and the receiver's current neighbors contains an edge from the sender's location to the receiver's. If the sender should be a neighbor, it is added to the neighborhood and then the neighborhood invariant is restored, which may cause some neighbors to be deleted.

If the sender was not added as a neighbor and the message sent was a ReadRequest, then a ReadReply message is sent back. If the sender was added as a neighbor, then no ReadReply is sent back because, when the timer expires, a ReadRequest message will be sent to it. Thus, in essence, the reply is just being delayed and merged with the ReadRequest sent at the next timer expiration.

Lastly, for each message received, the receiver updates the candidate neighbor information. If the sender has a neighbor that should be a neighbor of the receiver and currently is not, and that neighbor of the sender is closer than the current candidate neighbor, then the sender's neighbor becomes the new candidate neighbor. The source-routed path to the candidate neighbor is gotten by reversing the path from the sender to the receiver and concatenating the path from the sender to its neighbor.

6.3.3 Node Joins

When a node joins, it broadcasts a ReadRequest and starts the timer. In time, it will receive messages from the other nodes in order to initialize its neighborhood.

6.3.4 Node Moves

When a node moves, it needs to restore the neighborhood invariant, because some nodes in the neighborhood may now violate the neighborhood invariant. The violating nodes are removed from the neighborhood.

6.3.5 Node Leaves

When a node leaves, it does nothing. Other nodes will delete it from their neighborhood when the information grows stale in $T_{timeout}$ seconds.

This concludes the description of the current running version of the protocol. I will now discuss why $P_i.m$ is not being calculated and how the protocol can be extended to match the algorithm.

6.4 Not calculating $P_i.m$

The value of $P_i.m$ at every processor P_i plays two roles in the algorithm. One of its roles is to provide *local connectivity* — each node can find at least one neighbor. The other of its roles is to provide *global connectivity* — that the set of nodes forms a single Delaunay triangulation and not multiple smaller ones.

Local connectivity is provided for in the protocol by broadcasting ReadRequest messages every $T_{heartbeat}$ seconds. Any pair of nodes within communication range can hear those broadcast messages and find a neighbor.

Global connectivity is not provided for by the current implementation of the protocol. In one sense, global connectivity is impossible to guarantee. If there is a partition of the nodes into sets A and B such that none of the nodes in A can communicate directly with any of the nodes in B, no protocol could form a single Delaunay triangulation.

But what about a lesser definition of global connectivity? If there does not exist such a partition, can this protocol guarantee that a single Delaunay triangulation is computed? No. However, I believe this algorithm does form a single triangulation where there are no obstructions or other effects that lessens the transmission radius. Currently, the simulator does not support obstructions, so implementing the calculation of $P_i.m$ has not become vital to my studies.

I am considering two possible enhancements to ensure global connectivity. The first enhancement would have each node store a source-route to what it believes to be the node with the greatest coordinates. Each node that did not have a neighbor with greater coordinates would flood through out the network a message that contained a source-route to it. Every node would forward the source route of the node with the greatest coordinates. If a node did not have a neighbor with greater coordinates and received a flooded message, it would send a ReadRequest message to the node with the greatest coordinates.

This first enhancement would ensure global connectivity, but it does so by creating messages that use very large source-routes. In dynamic environments, those source routes may not be valid for very long. Additionally, a message that must traverse many hops has a higher probability of being lost.

The second possible enhancement tries to avoid messages that use long source-routes. In this enhancement, every node that does not have a neighbor with greater coordinates floods only its location and floods it only over the Delaunay triangulation of which it is a part. Thus, if there are multiple Delaunay triangulations, the nodes in each would record different nodes as being the one with the greatest coordinates. If two nodes that can communicate (i.e., are able to transmit to each other, but are not neighbors) discover that they have different nodes as the greatest in the DT, then they know that they are part of two separate DTs and need to merge them.

For the merge operation, let a and b be the nodes that can communicate but are part of separate DTs. Node a will send a message of a new type, a MergeStepOne message, to b. The MergeStepOne message contains a's coordinates. When b receives a MergeStepOne message, it sends out a MergeStepTwo message. The MergeStepTwo message contains a's coordinates and a source route to a. The MergeStepTwo message is *not* source routed, but greedy routed over b's DT. The MergeStepTwo is routed to c, the node in b's DT that is closest to a's coordinates. The node c now sends a ReadRequest message to a and begins the merging of the two DTs. The node c must eventually add a as a neighbor, because it is the node in its DT that is closest to a.

This second enhancement would use shorter source-routes than the first enhancement, because nodes a, b and c would have to be geographically close to one another. It would appear to ensure global connectivity for any set of nodes, but it does depart drastically from the $P_i.m$ feature of the proof. Restated, it is not obvious that this second enhancement is derivative of the proof and it is not obvious that the resulting protocol would have the qualities demonstrated in the proof.

7 Simulation Results

I remind the reader that the contribution of this paper is the algorithm and its proof of correctness. The results presented here are meant to be emblematic of the protocol's operation and not the rigorous study expected of a systems paper.

The simulator was written in Java and implements the 802.11 MAC protocol. The code is available by emailing the author. The simulator can be run on the web at *URL commented out.*

The study consisted of 10 simulations, each with 128 nodes placed randomly in a square. The size of the square was selected so that, on average, each node could communicate with 8 other nodes. The value 8 was selected as a small value where the physical network was connected. In 4 of the 10 simulations, there were disconnected components. In these cases, the disconnected components were removed — up to 3 nodes were deleted and not replaced.

The time to stabilize fell between 15 and 30 seconds. With $T_{heartbeat}$ set to 2 seconds, that means that runs stabilized after between 8 and 15 timer periods. The time to stabilize seemed to be loosely correlated with the maximum length of a source-route seen in the network.

When stable, nodes averaged 6 DT neighbors and none had more than 10 neighbors. (No numbers were gathered when the graph was not stable.)

Figure 6 shows a histogram of the number of hops in the source routes. 74 percent of DT edges were a direct connection between nodes and 95 percent of DT edges had 5 or fewer hops in their source route. The maximum length source-route varied from run to run with the low being 11 hops, and the high being 32 hops. Most long source routes occurred at the borders of the network, where nodes were often in range of only one other node.

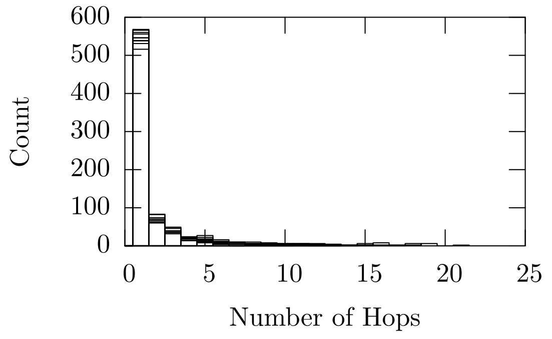

Figure 6: Histogram of hops in the source-routes.

To evaluate routes, 100 pairs of endpoints were randomly selected for each of the 10 simulation. Figure 7 plots the distance between the endpoints to the distance of the path on the DT. The length of the path on the DT is almost linear to the distance between the endpoints.

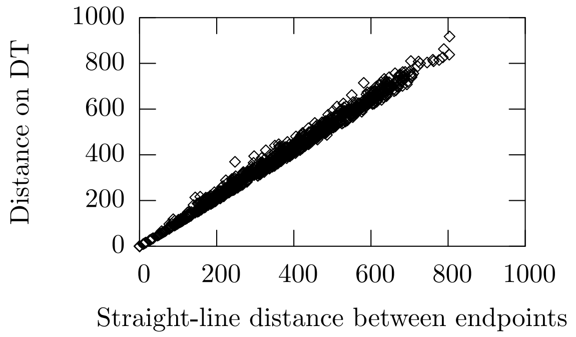

Figure 7: Distance vs. Distance on DT, 1000 routes.

Figure 8 plots the distance between the endpoints against the number of hops in the physical network. The relationship between distance and hops is non-linear for three reasons. The first reason is that some straight edges must follow a circuitous source-route around holes in the network. The second reason is that some DT edges are short, so that a single hop might not carry a route a great distance. The third reason is that a route could cross between the same pair of nodes multiple times.

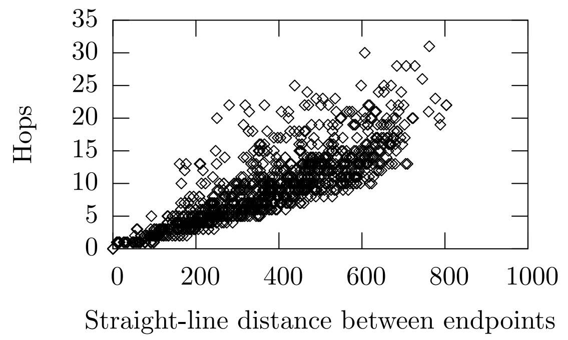

Figure 8: Distance vs. hops, 1000 routes.

8 Conclusion

In this paper, I have presented a new distributed algorithm for computing the Delaunay graph. The algorithm is self-stabilizing and is, therefore, suited for mobile adhoc applications.

I presented the design for a protocol which, with the calculation of $P_i.m$, should solve the four problems with the state-of-the-art solution, planar-subgraph recovery. First, the presented protocol is a generic routing protocol — it will work on any connected graph; it just happens to be efficient when nearby nodes can communicate. As a result of this, interference, cancellations, and obstructions can lower the efficiency of the network, but not, as in planar subgraph recovery, prevent routes that exist from being found.

Second, nodes near "holes" in the network have a choice of source-routes, either through their clockwise or couterclockwise neighbors, and selecting the lowest cost source-route should lower the chance of messages taking the "long way" around holes in the network. The presented protocol will never route a message around the complete circumference of the graph.

Third, the presented algorithm can quickly determine if a node is not present in the graph. Once the message has reached the node in the graph that is closest to the (absent) destination, that node can tell that the destination does not exist or has moved. The message will never completely circle a hole in the graph nor traverse the entire circumference of the graph in order to determine that a node is unreachable.

Lastly, the presented protocol works in three dimensions. Admittedly, most of the properties listed in the introduction have not been proven for the Delaunay graph in three dimensions and it is known that, in three dimensions, nodes can average $O(n)$ neighbors.[1] Nonetheless, this paper's protocol, to the best of my knowledge, is the first position-based routing protocol for three dimensions that guarantees delivery of all messages without flooding.

References

[1] Franz Aurenhammer. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[2] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *3rd Int. Workshop on Discrete Algorithms and methods for mobile computing and communications (DialM '99)*, 1999.

[3] Prosenjit Bose and Pat Morin. Online routing in triangulations. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC'99)*, 1999.

[4] Nikos Chrisochoides and Florian Sukup. Task parallel implementation of the Bowyer-Watson algorithm. In *Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1996.

[5] Herbert Edelsbrunner and Ernst P. Mucke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. In *Symposium on Computational Geometry*, pages 118–133, 1988.

[6] J. Gao, L. J. Guibas, J. Hershburger, L. Zhang, and A. Zhu. Geometric spanner for routing in mobile networks. In *Proceedings of the 2nd ACM Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc01)*, pages 45–55, 2001.

[7] S. Giordano, I. Stojmenovic, and L. Blazevic. Position based routing algorithms for ad hoc networks: A taxonomy. *Ad Hoc Wireless Networking*, 2003. To appear.

[8] A. L. Hinde and R. E. Miles. Monte carlo estimates of the distributions of the random polygons of the Voronoi tessellation with respect to a Poisson process. *Journal of Statistical Computer Simulations*, 10:205–223, 1980.

[9] B. Karp. Challenges in geographic routing: Sparse networks, obstacles, and traffic provisioning. In *DIMACS Workshop on Pervasive Networking*, 2001.

[10] B. Karp and H.T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *MobiCom 2000*, August 2000.

[11] J. Mark Keil and Carl A. Gutwin. The Delaunay triangulation closely approximates the complete Euclidean graph. In *Workshop on Algorithms and Data Structures 1989*, pages 47–56, 1989.

[12] Xiang-Yang Li, G. Calinescu, and Peng-Jun Wan. Distributed construction of planar spanner and routing for ad hoc networks. In *Proceedings of IEEE INFOCOM'2002*, 2002.

[13] Martin Mauve, Jörg Widmer, and Hannes Hartenstein. A survey on position-based routing in mobile ad-hoc networks. In *IEEE Network*, November 2001.

[14] E. Puppo, L. Davis, D. De Menthon, and Y. A. Teng. Parallel terrain triangulation. *Interational Journal of Geographical Information Systems*, 8(2):105–128, 1994.

[15] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.