On the Efficiency of a Local Iterative Algorithm to Compute Delaunay Realizations

Kevin M. Lillis[1,2] and Sriram V. Pemmaraju[1]

[1] Department of Computer Science,
University of Iowa, Iowa City, IA 52242-1419, U.S.A.
[lillis,sriram]@cs.uiowa.edu
[2] Computer and Information Science,
St. Ambrose University, Davenport, IA 52803, U.S.A.
LillisKevinM@sau.edu

Abstract. Greedy routing protocols for wireless sensor networks (WSNs) are fast and efficient but in general cannot guarantee message delivery. Hence researchers are interested in the problem of embedding WSNs in low dimensional space (e.g., $\mathbb{R}^2$) in a way that guarantees message delivery with greedy routing. It is well known that Delaunay triangulations are such embeddings. We present the algorithm `FindAngles`, which is a fast, simple, local distributed algorithm that computes a Delaunay triangulation from any given combinatorial graph that is Delaunay realizable. Our algorithm is based on a characterization of Delaunay realizability due to Hiroshima et al. (IEICE 2000). When compared to the `PowerDiagram` algorithm of Chen et al. (SoCG 2007) that embeds triangulations in the plane so as to permit successful greedy routing, our algorithm requires on average $1/6^{th}$ the number of iterations. `FindAngles` also scales linearly to larger networks and has a much faster distributed implementation than `PowerDiagram`, requiring just a single round of communication in each iteration. The `PowerDiagram` algorithm was proposed as an improvement on another algorithm due to Thurston (unpublished, 1988). Our experiments show that on average the `PowerDiagram` algorithm uses about 19% fewer iterations than the Thurston algorithm, whereas our algorithm uses about 89% fewer iterations. Experimentally, `FindAngles` exhibits well behaved convergence. Theoretically, we prove that with certain initial conditions the error term decreases monotonically. Taken together, these suggest our algorithm may have polynomial time convergence for certain classes of graphs. We note that our algorithm runs only on Delaunay realizable triangulations. This is not a significant concern because Hiroshima et al. (IEICE 2000) indicate that most combinatorial triangulations are indeed Delaunay realizable, which we have also observed experimentally: out of 5000 randomly generated combinatorial triangulations on 100 vertices, only one was not Delaunay realizable.

1 Introduction

A wireless sensor network (WSN) consists of battery-powered nodes that can communicate with one another when within radio broadcast range and can perform local computations. Because there is no centralized control and because each node has only a small amount of memory, WSNs often employ *memoryless routing*, in which each node decides to whom a message should be forwarded based solely on the source of the message, its destination, and information gathered from nearby nodes. *Geographic routing protocols* are memoryless routing protocols that use geographic information such as the coordinates of the source, the destination, and nearby nodes. In the geographic routing protocol known as *greedy routing* each node forwards a message by choosing a neighbor that is closest to the destination. While this scheme is simple and efficient, it does not guarantee message delivery. This is because a routed message may become trapped in a cycle [6] or it may reach a *local minimum*; this is a node that is closer to the destination than any of its neighbors [11].

Given a point set $P \subseteq \mathbb{R}^2$, a *Delaunay triangulation* of P is commonly defined as a triangulation of P satisfying the property that the circumcircle of each inner face (triangle) contains no point of P in its interior. This is known as the *empty circle property*. A Delaunay triangulation of P is also well known as the planar dual of the *Voronoi diagram* of P. It is this characterization that one can use to see that greedy geographic routing will always succeed on a Delaunay triangulation. Consider a node s that currently possesses the message M, intended for a destination t. Let $v_1, v_2, \ldots, v_m$ be points in P whose Voronoi cells are adjacent to the Voronoi cell of s. Then for some v_j, $1 \leq j \leq m$, the half plane consisting of all points closer to v_j than to s will contain t. Since the Delaunay triangulation is the planar dual of the Voronoi diagram, each v_i, $1 \leq i \leq m$, is a neighbor of s in the Delaunay triangulation. Specifically, the point v_j is a neighbor of s that is closer to t than s.

The fact that greedy geographic routing is always successful on a Delaunay triangulation motivates the question of whether a given WSN can be embedded in the plane as a Delaunay triangulation. In cases where the WSN is not a

triangulation to start with, standard topology control protocols [17, 20] can be employed to extract a planar spanning subgraph of the WSN which can then easily be converted into a triangulation. Define a *combinatorial triangulation* as a planar graph, all of whose inner faces are 3-cycles. Suppose that we are given a WSN $G = (V, E)$ that is a combinatorial triangulation. We seek a one-one mapping $\Phi : V \to \mathbb{R}^2$ such that if each vertex $v \in V$ is placed on the plane at $\Phi(v)$ and each edge $\{u, v\} \in E$ is represented by a straight line segment with endpoints $\Phi(u)$ and $\Phi(v)$, then the set of points $\Phi(V) = \{\Phi(v) \mid v \in V\}$ and the set of line segments $\Phi(E) = \{\{\Phi(u), \Phi(v)\} \mid \{u, v\} \in E\}$ defines a Delaunay triangulation. The problem of finding the mapping Φ is called the *Delaunay realization problem*, the mapping Φ, if it exists, is called a *Delaunay realization*, and combinatorial triangulations G for which a Delaunay realization exists are called *Delaunay realizable* graphs. The problem of determining whether or not a combinatorial triangulation is Delaunay realizable can be solved in polynomial time; for example by checking if a certain linear system of inequalities defined by Hiroshima et al. [10] has a feasible solution. However, as far as we know, the problem of actually finding a Delaunay realization does not have a polynomial time solution and seems rather difficult. In this paper we present a simple, iterative algorithm, called `FindAngles`, that finds a Delaunay realization of a Delaunay realizable graph. We do not prove polynomial time convergence of this algorithm, but we do present substantial experimental evidence indicating that `FindAngles` converges rapidly. The algorithm is inherently local and has an obvious, distributed implementation in which each node updates some local geometric information based on such information at neighboring nodes.

Recently two other approaches have been considered for the problem of finding embeddings that permit successful greedy geographic routing. The first approach, due to Papadimitriou and Ratajczak [16], seeks *greedy embeddings*. Let a *distance decreasing* path in an embedding of a graph be a path $s = v_1, v_2, \ldots, v_k = t$ such that $\|v_i - t\| < \|v_{i-1} - t\|$, $2 \leq i \leq k$. Here $\| \ldots \|$ denotes Euclidean distance. A *greedy embedding* of a graph is an embedding into the Euclidean plane such that there exists a distance decreasing path between every pair of vertices. A Delaunay realization of a combinatorial triangulation is clearly a greedy embedding, but there may be greedy embeddings that are triangulations, but not Delaunay triangulations. In fact, Papadimitriou and Ratajczak [16] conjecture that every combinatorial triangulation has a greedy embedding[3]. This conjecture was very recently proved by Dhandapani [8]. The proof essentially depends on the Knaster-Kuratowski-Mazurkiewicz Theorem [12] that is known to be equivalent to the Brouwer Fixed Point Thorem. Due to this dependency, Dhandapani's proof does not seem to immediately lead to a polynomial time algorithm. He does mention an iterative algorithm at the end of his paper, but without any theoretical bounds or experimental results.

A second approach to the problem of finding embeddings that permit successful greedy geographic routing uses *power diagrams* and is due to Chen at al. [5]. Let $P \subseteq \mathbb{R}^2$ be a planar set of points, with each point $p \in P$ having an associated disk $D(p)$ with center p and radius $r(p) \geq 0$. The *power distance* from any point $q \in \mathbb{R}^2$ to p, denoted $power(q, p)$ is $\|p - q\|^2 - r(p)^2$. The *power diagram* of P is the cell complex in $\mathbb{R}^2$ that associates to each $p \in P$, the convex domain

$$cell(p) = \{q \in \mathbb{R}^2 \mid power(q, p) < power(q, p') \text{ for all } p' \in P - \{p\}\}.$$

The properties and applications of power diagrams, as well as algorithms for constructing them, have been studied by Aurenhammer [3]. The Voronoi diagram is a special case of a power diagram, obtained by setting $r(p) = 0$ for all $p \in P$. Just as Delaunay triangulations are duals of Voronoi diagrams, a more general class of triangulations (that include Delaunay triangulations) are the planar duals of power diagrams. These are called *regular triangulations* by Edelsbrunner and Shah [9]. This motivates the question of whether for a given combinatorial triangulation $G = (V, E)$, we can find an embedding $\Phi : V \to \mathbb{R}^2$ and a radius assignment $r : V \to \mathbb{R}^+$, whose power diagram has, as its planar dual, the graph G. If we can find such an embedding, then greedy geographic routing can be used with power distance in lieu of Euclidean distance. The argument that greedy routing with power distance will always be successful on regular triangulations is identical to the argument provided above that shows that greedy geographic routing always works on Delaunay triangulations. For any combinatorial triangulation G (and in fact for any planar graph), it is known that there exists an embedding $\Phi : V \to \mathbb{R}^2$ and a radius assignment $r : V \to \mathbb{R}^+$ that yields a power diagram whose dual is G. This follows from the celebrated Koebe Representation Theorem [1, 2, 13, 22]. None of the known proofs of Koebe's theorem lead to efficient algorithms. For example, the proof of this theorem in the book by Pach and Agarwal [15] (which was originally due to Thurston [22]) critically uses a fixed point theorem. Again, we have a situation in which the existence of appropriate embeddings are well known, but the proof of existence does not give a clear indication of how to efficiently compute these embeddings. It should be noted that there are polynomial time algorithms to compute a Koebe representation [14, 21], but these use the ellipsoid method and are therefore

[3] Actually, the Papadimitriou-Ratajczak conjecture is more general and states that every 3-connected planar graph has a greedy embedding.

not practical and are essentially useless for obtaining a fast, distributed algorithm. Given this situation, Chen et al. [5] use a simple iterative algorithm to produce a Koebe representation. This algorithm is not guaranteed to run in polynomial time and is obtained in a straightforward way from Thurston's proof of Koebe's theorem. We will call this the `Thurston` algorithm. Chen et al. [5] also present an algorithm that improves on the performance of the `Thurston` algorithm by using fewer iterations to terminate with a power diagram. We call this the `PowerDiagram` algorithm. Both of these algorithms are described in some detail in Section 2.2. Chen et al. [5] present a small number of experimental results on small sized graphs comparing the performance of the `PowerDiagram` algorithm with that of the `Thurston` algorithm. In Section 4, we present strong experimental evidence indicating that our `FindAngles` algorithm is orders of magnitude faster than the `PowerDiagram` algorithm, which in turn is a bit faster than the `Thurston` algorithm.

Main results. We present a simple, iterative, local distributed algorithm, called `FindAngles`, for computing a Delaunay realization of any given combinatorial triangulation that is Delaunay realizable. Our algorithm is based on a characterization of Delaunay realizability due to Hiroshima et al. [10]. Our algorithm uses far fewer iterations as compared to the `PowerDiagram` algorithm of Chen et al. [5]. In experiments we ran, on average, the number of iterations of our algorithm was about $1/6^{th}$ of the number of iterations of the `PowerDiagram` algorithm (see Table 2). Using the `PowerDiagram` algorithm in a distributed setting is problematic because in each iteration, it requires a sweep of the entire graph in order to update coordinates. In a distributed setting, such operations are costly and each iteration of the `PowerDiagram` essentially corresponds to n communication rounds, where n is the size of the graph. In contrast, each iteration of `FindAngles` corresponds to exactly one round of communication. In addition, the number of iterations required by our algorithm scales linearly with n. The `PowerDiagram` algorithm was proposed as an improvement on the `Thurston` algorithm and in our experimental results, on average, the `PowerDiagram` algorithm uses about 19% fewer iterations than the `Thurston` algorithm, whereas our algorithm uses about 89% fewer iterations. Experimentally, `FindAngles` exhibits very well-behaved convergence; the error term falls rapidly in the beginning and then falls more slowly as the algorithm gets close to a valid Delaunay realization. The error term rarely increases and we are able to prove that it strictly decreases under certain initial conditions. This experimentally observed behavior, along with our analysis, provides some hope that our algorithm may have polynomial time convergence, at least for special classes of combinatorial triangulations (e.g., degree-bounded combinatorial triangulations). It should be noted that whereas the `PowerDiagram` algorithm runs on all combinatorial triangulations, `FindAngles` runs only on Delaunay realizable triangulations. This may not be much of a problem because the experimental results of Hiroshima et al. [10] indicate that most combinatorial triangulations are indeed Delaunay realizable. We also perform experiments whose outcomes strongly support these results. We randomly generated 5000 combinatorial triangulations on 100 vertices; of these only one was not Delaunay realizable.

2 Technical Background

Our algorithm is based on a characterization of Delaunay realizable graphs due to Hiroshima, Miyamoto, and Sugihara [10]. We call this the *HMS test* and describe it in Section 2.1. In our experiments, we compare `FindAngles` with the `Thurston` algorithm and the `PowerDiagram` algorithm. Both of these are based on the Koebe Representation Theorem and we describe that and sketch the `Thurston` algorithm and the `PowerDiagram` algorithm in Section 2.2.

2.1 The HMS test for Delaunay Realizability

Rivin [18, 19] presents a polynomial time test for Delaunay realizability that is based on ideas in hyperbolic geometry. In contrast, the HMS test is based on elementary Euclidean geometric ideas. We describe the HMS test in some detail here because our algorithm (described in Section 3) is based on the proof of correctness of the HMS test.

Let $G = (V, E)$ be a combinatorial triangulation with internal faces $f_1, f_2, \ldots, f_p$. Each face f_i is represented by a sequence of 3 vertices that bound the face in counter clockwise order. For each face f_i, three variables x_{3i+1}, x_{3i+2} and x_{3i+3} are defined, respectively corresponding to the three vertices in f_i. Thus there are a total of $3p$ variables, each of these is called an *angle variable*. These variables should be interpreted as angles at the corners of triangle f_i. See Figure 1(a). A vertex of G is called an *outer vertex* if it is on the boundary of the outer face of G; otherwise it is called an *inner vertex*. Similarly, an edge is called an *outer edge* if it is on the boundary of the outer face and is called an *inner edge* otherwise. Let $\{u, v\}$ be an inner edge and let (u, v, w_1) and (w_2, v, u) be the two faces containing edge $\{u, v\}$. Then the angle variables at w_1 associated with face (u, v, w_1) and the angle variable at w_2 associated with face (w_2, v, u) are the two *facing angle variables* associated with edge $\{u, v\}$. For example, in Figure 1(a) the facing

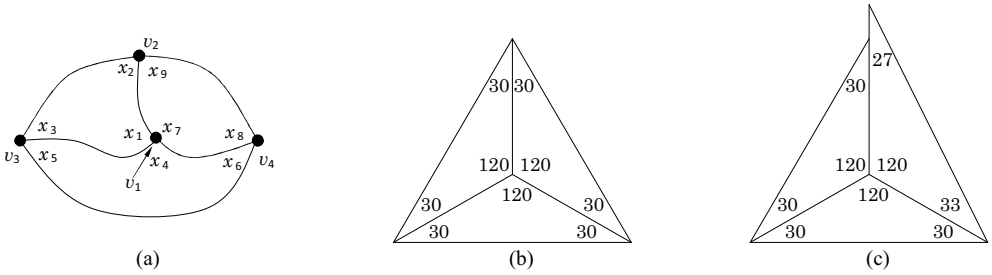

Fig. 1. This example is taken from [10] (a) The connectivity for a complete graph on 4 vertices. By condition (C1) $x_1+x_2+x_3 = 180$, $x_4+x_5+x_6 = 180$, $x_7+x_8+x_9 = 180$. By condition (C2) $x_1+x_4+x_7 = 360$. By condition (C3) $x_2+x_9 < 180$, $x_3+x_5 < 180$, $x_6 + x_8 < 180$. By condition (C4) $x_2 + x_6 < 180$, $x_3 + x_8 < 180$, $x_5 + x_9 < 180$. By condition (C5) $x_i > 0$ for $1 \leq i \leq 9$. (b) A solution to the linear constraints of that yield a Delaunay triangulation. (c) A solution to the linear constraints of that do not yield a Delaunay triangulation.

angle variables of edge $\{v_1, v_3\}$ are x_2 and x_6. Hiroshima et al. [10] state 5 simple linear constraints that the angle variables must satisfy in any Delaunay realization:

(C1) For each face f_i, the sum of the three associated angle variables x_{3i+1}, x_{3i+2} and x_{3i+3} is equal to 180.
(C2) For each inner vertex, the sum of the associated angle variables is equal to 360.
(C3) For each outer vertex the sum of the associated angle variables is less than or equal to 180.
(C4) For each inner edge, the sum of the associated pair of facing angle variables is less than or equal to 180.
(C5) Each angle variable is positive.

In the above list (C4) is particular to Delaunay triangulations and follows easily from the empty circle property. The remaining constraints are satisfied by every triangulation. Hiroshima et al. [10] point out that even though every Delaunay realization satisfies conditions (C1)-(C5), these conditions are not sufficient because they do not guarantee that the triangles incident on a vertex v can be consistently "glued" together around v (see Figure 1(c)). Interestingly, Hiroshima et al. [10] were able to show that for any solution that does satisfy (C1)-(C5), there exists a transformation that modifies this solution into one that additionally satisfies a "consistent gluing" condition around each vertex. Thus Hiroshima et al. [10] were able to reduce the problem of testing if a combinatorial triangulation is Delaunay realizable into a problem of testing the feasibility of a linear system of equations and inequalities. Note that the size of this system (i.e., number of variables, number of constraints) is linear in the number of vertices in G. The result of Hiroshima et al. [10] can be summarized as follows.

Theorem 1. ***(Hiroshima et al. [10])*** *Whether a given combinatorial triangulation is Delaunay realizable can be tested in polynomial time.*

The transformation mentioned above, that takes a solution to (C1)-(C5) and modifies it to additionally satisfy a "consistent gluing" condition is only shown existentially by Hiroshima et al. [10]. As a result the above theorem does not lead to a polynomial time algorithm for constructing a Delaunay realization. We now review the proof of Hiroshima et al. [10] which prompted our algorithm, presented in Section 3.

Let v be an inner vertex of G and let $w_0, w_1, \ldots, w_{s-1}$ be its neighbors in counter clockwise order. Then the angles $(v, w_0, w_1), (v, w_1, w_2), \ldots, (v, w_{s-2}, w_{s-1}), (v, w_{s-1}, w_0)$ are called *cc-facing angles* about v. Denote these respectively by $\phi_0, \phi_1, \ldots, \phi_{s-1}$. Similarly, the angles $(v, w_1, w_0), (v, w_2, w_1), \ldots, (v, w_{s-1}, w_{s-2}), (v, w_0, w_{s-1})$ are called *c-facing angles* about v. Denote these respectively by $\theta_0, \theta_1, \ldots, \theta_{s-1}$ (see Figure 2). Define

$$F(v) = \frac{\sin(\phi_0)\sin(\phi_1)\cdots\sin(\phi_{s-1})}{\sin(\theta_0)\sin(\theta_1)\cdots\sin(\theta_{s-1})}$$

It is not difficult to see that to be able to "glue" the triangles incident on v consistently around v, $F(v)$ must equal 1. This is shown in Lemma 3.1 of Hiroshima et al. [10] and follows from a straightforward trigonometric argument. Note that since all ϕ_i's and θ_i's are strictly between 0 and 180, we have $F(v) > 0$. Hiroshima et al. define a condition (C6) as follows:

(C6) $\quad F(v) = 1$ for each inner vertex v.

Conditions (C1)-(C6) exactly capture Delaunay realizability; these conditions are both necessary and sufficient. Condition (C6) has to be satisfied by any triangulation, not just Delaunay triangulations. Unfortunately, (C6) is not "well behaved" like conditions (C1)-(C5) and no efficient algorithm is known for solving the system of equations and inequalities implied by conditions (C1)-(C6).

Consider any real α. Let $change(v, \alpha)$ denote the operation in which each cc-facing angle ϕ_i around v is changed to $\phi_i + \alpha$ and each c-facing angle θ_i around v is changed to $\theta_i - \alpha$. Let ϕ_{min} denote $\min_i \phi_i$ and let θ_{min} denote $\min_i \theta_i$. For any α, $-1 \times \phi_{min} < \alpha < \theta_{min}$, applying the operation $change(v, \alpha)$ keeps all the angles positive. The operation $change(v, \alpha)$, for α, $-1 \times \phi_{min} < \alpha < \theta_{min}$, has the very useful property that if conditions (C1)-(C5) are satisfied before the operation, they will continue to be satisfied after the operation. Hiroshima et al. [10] show that given an assignment of values to the angle variables satisfying (C1)-(C5), there exists a set $\{(v_1, \alpha_1), (v_2, \alpha_2), \ldots\}$, such that performing $change(v_1, \alpha_1), change(v_2, \alpha_2), \ldots$ leads to condition (C6) also being satisfied. While the existence of $\{(v_1, \alpha_1), (v_2, \alpha_2), \ldots\}$ is shown, there is no known polynomial time algorithm to find it.

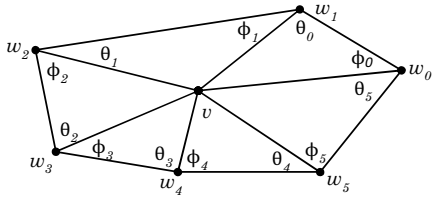

Fig. 2. A vertex v with 6 neighbors. The cc-facing angles about v are $\phi_0, \phi_1, \ldots, \phi_5$. The c-facing angles are $\theta_0, \theta_1, \ldots, \theta_5$.

2.2 The Koebe Representation Theorem

Koebe [13] in 1936 proved the following remarkable theorem.

Theorem 2. ***(Koebe [13])*** *Given any planar graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$, we can find a packing of n (not necessarily congruent)disks $C = \{C_1, C_2, \ldots, C_n\}$ in the plane with the property that C_i and C_j touch iff $\{v_i, v_j\} \in E$ for $1 \leq i, j \leq n$.*

The fact that the set C is a circle *packing* implies that the interiors of the circles are pairwise mutually disjoint.

From Thurston's proof of Koebe's theorem [22] one can extract an iterative algorithm for (approximately) computing a Koebe representation (i.e., a circle packing promised by Koebe's theorem) for a given planar graph. In the following description we assume, for simplicity, that G is a combinatorial triangulation. Let $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ be arbitrary, initial radii assigned to the vertices $v_1, v_2, \ldots, v_n$ respectively. Then $\mathbf{r}$ uniquely defines the three angles in each inner face of G. Let $\sigma_{\mathbf{r}}(v_i)$ denote the sum of the angles at v_i in all of the inner faces to which v_i belongs. The subscript "$\mathbf{r}$" here denotes the fact that the angles depend on the radii assignment $\mathbf{r}$. If $\sigma_{\mathbf{r}}(v_i) = 2\pi$ for all inner vertices v_i, we are done because we can consistently "glue" the triangular faces together. If $\sigma_{\mathbf{r}}(v_i) \neq 2\pi$ for some inner vertex v_i, then we can find a new radius r_i' for v_i such that with respect to $\mathbf{r}' = (r_1, r_2, \ldots, r_i', \ldots, r_n)$, the angle sum $\sigma_{\mathbf{r}'}(v_i) = 2\pi$. Of course this update affects the angle sums at neighbors of v_i. However, it is possible to show that this iterative process converges. Collins and Stephenson [7] suggest several improvements to this basic algorithm that reduce the number of iterations needed for convergence. This algorithm has an obvious distributed implementation in which, in each round, each node v_i for which $\sigma_r(v_i) \neq 2\pi$ updates its radius r_i. New radius values are then exchanged with neighboring nodes in one round of local communication, and the algorithm proceeds to the next round. It is this entire class of algorithms that we refer to as the `Thurston` algorithm.

A Koebe representation $C = \{C_1, C_2, \ldots, C_n\}$ of a combinatorial triangulation $G = (V, E)$ can also be viewed as a power diagram whose planar dual is G. This is the starting point of the work of Chen et al. [4, 5], who go on to point out that Koebe representations are special power diagrams in which disks corresponding to adjacent cells are mutually tangent. With this motivation, Chen et al. [4, 5] develop a *local power diagram (LPD)* test takes as input a mapping $\Phi : V \to \mathbb{R}^2$ and an assignment of disks $D(v)$ to vertices $v \in V$ and determines if $\{\Phi, D\}$ is a power diagram. It is this algorithm that we call `PowerDiagram`. This is called a "local" test because it involves checking a condition at each vertex v that is a function of Φ and D at v and its neighbors only. Thus this test can be implemented in a distributed fashion. One problem with LPD is the fact that it depends not just on the radii of the disks, but also on the coordinates of the vertices specified by Φ. This is a problem because after each radii update (as in the description of Thurston's algorithm) we have to recompute the coordinates of all the vertices by using some kind of a global sweep of the graph. Thus, after each round in which all nodes update their radii, we need $\Omega(diameter)$ rounds of communication to update the coordinates. In our view, this problem largely offsets the gains obtained by using LPD.

3 The FindAngles Algorithm

The FindAngles algorithm first obtains an initial angle assignment by finding a feasible solution of the linear program defined by constraints (C1)-(C5). The algorithm terminates if no feasible solution it found. Once an initial angle is found, the algorithm repeatedly adjusts the angle values. It is this angle adjustment process that is the focus of our work. Angles are adjusted through an iterative process that repeatedly applies the $change(v, \alpha)$ operation introduced in Section 2.1. This process consists of a sequence of *rounds*. In each round, we scan through all the inner vertices in an arbitrary order. At each inner vertex v we check if $F(v) = 1$. If this is not the case, we solve for an α^* such that after applying $change(v, \alpha^*)$, $F(v)$ equal 1. We then apply $change(v, \alpha^*)$. To see that it is possible to efficiently solve for such an α^*, define the following function $g : \mathbb{R} \to \mathbb{R}$:

$$g(\alpha) = \frac{\sin(\phi_0 + \alpha)\sin(\phi_1 + \alpha)\cdots\sin(\phi_{s-1} + \alpha)}{\sin(\theta_0 - \alpha)\sin(\theta_1 - \alpha)\cdots\sin(\theta_{s-1} - \alpha)}. \quad (1)$$

Now suppose that $F(v) < 1$. Then $g(0) = F(v) < 1$ and $\lim_{\alpha\to\theta_{min}} g(\alpha) = +\infty$ (recall that $\theta_{min} = \min_i \theta_i$). Since $g(\alpha)$ is a continuous function, it is guaranteed that for some $\alpha^* \in (0, \theta_{min})$, $g(\alpha^*) = 1$. In fact $g(\alpha)$ is monotonically increasing as α increases from 0 to θ_{min}. To see this observe that the ratio $\sin(\phi_i + \alpha)/sin(\theta_i - \alpha)$ is monotonically increasing for $0 < \alpha < \theta_i$ provided $\phi_i + \theta_i \leq 180$. This is clearly so since ϕ_i and θ_i are required to be positive (condition (C5)) and the sum of ϕ_i, θ_i, and a third angle equals 180 (condition (C1)). Thus the equation $g(\alpha) = 1$ can be easily solved using a standard numerical root finding technique such as Newton's method. We have a symmetric situation if $F(v) > 1$, where we solve for an $\alpha^* < 1$ such that $g(\alpha^*) = 1$. After each iteration, we check if a global termination condition is satisfied. This termination condition depends on a fixed parameter $\varepsilon > 0$ and is defined as either:

Max-termination condition: $|1 - F(v)| < \varepsilon$ for all inner vertices v

or

Sum-termination condition: $\sum_{v\in I} |1 - F(v)| < \varepsilon$ where I is the set of inner vertices in G.

We have separately implemented both termination conditions and report results for both, however we focus on the first termination condition since it is local, i.e., each node can decide for itself, based on local information, if it needs to participate in the current iteration. Once the iterations terminate we perform a global BFS sweep of the graph to fix planar coordinates of all vertices in G. We Start with an arbitrary pair of adjacent vertices u and v and place them arbitrarily, at distinct points on the plane. The sweep guarantees that for every subsequent vertex w that is processed, there is a triangle of G, (a, b, w) such that a and b are vertices that have already been placed in $\mathbb{R}^2$. Given the fact that the three angles of the triangle (a, b, w) are known, there is a unique location in $\mathbb{R}^2$ for w. Upon termination of our algorithm, each $F(v)$ may differ from 1 by ε. Hence the angle variables themselves yield an *approximate* Delaunay realization (see Figure 3). We therefore geometrically evaluate each approximate Delaunay realization to see if it is indeed a Delaunay triangulation. As expected, as ε becomes smaller, the fraction of approximate Delaunay realizations that are actually Delaunay triangulations, increases and reaches 1 (see Table 3). While FindAngles may not run in polynomial time, experimental evidence suggests that it is very fast (see Section 4), leading us to believe that some variant of this algorithm may indeed be shown to have polynomial time convergence. The FindAngles algorithm is summarized in the below pseudocode. The algorithm was implemented in *Mathematica 6.0*, using in-built functions for linear programming and root finding. Details of the implementation appear in the appendix.

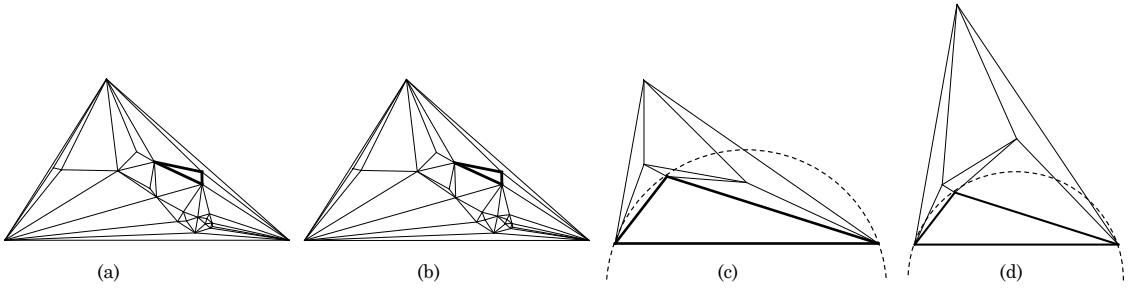

Fig. 3. (a) and (b) were produced from a single input with $\varepsilon = 10^{-2}$ and $\varepsilon = 10^{-6}$ respectively. (a) is not a Delaunay triangulation (the highlighted triangle violates the empty circle property) while (b) is Delaunay. For illustrative purpose only, (c) and (d) were produced from a single input with $\varepsilon = 20$ and $\varepsilon = 12$ respectively. (c) is not Delaunay while (d) is.

Algorithm `FindAngles`
Input: G: A combinatorial triangulation, $\varepsilon \in \mathbb{R}^+$
Output: An embedding of G in $\mathbb{R}^2$ that is a Delaunay triangulation, or report G is not Delaunay realizable.

1. Find a feasible solution to the LP defined by constraints (C1) - (C5). This is the initial angle assignment.
2. **if** (no feasible solution exists)
3. Report that G is not Delaunay realizable
4. Stop
5. **end if**
6. **while** ($\exists$ inner vertex $v : \mid 1 - F(v) \mid \geq \varepsilon$)
7. **for**(each inner vertex u)
8. **if** ($\mid 1 - F(u) \mid \geq \varepsilon$)
9. Compute α^* such that $F(u) = 1$ following $change(u, \alpha^*)$
10. Apply $change(u, \alpha^*)$
11. **end if**
12. **end for**
13. **end while**
14. Arbitrarily select adjacent vertices u and v and embed them at distinct points in $\mathbb{R}^2$
15. Perform a BFS traversal starting with u or v. At each vertex w perform the following:
16. Pick a triangle (a, b, w) such that a and b have been embedded
17. Embed w in $\mathbb{R}^2$ using the three angle assignments associated with triangle (a, b, w)

4 Experimental Results

The basic experiments described in this section were run on a total of 65 input graphs, 13 each of order $n \in \{23, 43, 63, 83, 103\}$. To determine how well our algorithm scales we used 50 larger graphs up to order $n = 1003$. For a given input graph, we are interested in finding a realization that is the combinatorial dual of a power diagram. For simplicity, when we succeed in finding such a realization, we say we have found a "Power Diagram".

4.1 Relative Performance of Algorithm `FindAngles`

In this subsection we compare the performance of the `FindAngles` algorithm with that of algorithms `Thurston` and `PowerDiagram`. The comparison is along three dimensions: number of iterations, fraction of realizations that are power diagrams, and running time. Along all three dimensions, `FindAngles` outperforms both of the other algorithms.

We start by looking at the number of iterations and the processing time required by each algorithm to produce 100% power diagrams (see Table 1 and Figure 4). Recall that the `PowerDiagram` algorithm always terminates with a valid power diagram. So to perform a meaningful comparison, for each n we pick a largest ε for which all realizations generated by `FindAngles` are Delaunay triangulations and report the average number of iterations and time required using this ε. Note different ε may be selected for different n. The results for the `Thurston` algorithm are similarly selected and reported. Chen et al. [5] have proposed the `PowerDiagram` algorithm as an improvement over the `Thurston` algorithm. However, it is clear from Table 1 that the improvement of the `PowerDiagram` algorithm is marginal, relative to the improvement obtained by using our algorithm. For example, for $n = 103$, the `Thurston` algorithm needs 1664 iterations, the `PowerDiagram` algorithm needs 1330 iterations, whereas Algorithm `FindAngles` needs only 122 iterations! The last three columns of the table show the times (in seconds) for the three algorithms. Again, our algorithm is significantly faster than both the `Thurston` algorithm and the `PowerDiagram` algorithm. In our implementation, the `PowerDiagram` algorithm takes significantly more time than the `Thurston` algorithm, despite using fewer iterations. This is due to fact that we implemented the local power diagram test by solving a non-linear system of equations using the Mathematica function `FindInstance`. It is possible that an alternate, purely geometric, implementation of the local power diagram test would speed up the `PowerDiagram` algorithm.

Next we compare `FindAngles` with the `Thurston` algorithm (see Table 2 and Figure 5). The reported results are for $\varepsilon = 10^{-5}$. Similar tables for other values of ε appear in the appendix. Again, the comparison is along three dimensions and again, `FindAngles` significantly out performs the `Thurston` algorithm.

As mentioned in Section 3, our algorithm produces an approximate Delaunay realization, where the approximation depends on ε: as $\varepsilon \to 0$, the realization produced gets "closer" to a Delaunay triangulation. Table 3 shows the increase in the fraction of actual Delaunay triangulations produced by `FindAngles` and the `Thurston` algorithm as $\varepsilon \to 0$. For example, for $\varepsilon \leq 1 \times 10^{-5}$, all realizations produced by Algorithm `FindAngles` are Delaunay triangulations.

	Average Iterations for 100% Power Diagrams			Average Time (in sec.) for 100% Power Diagrams		
n	`Thurston`	`PowerDiagram`	`FindAngles`	`Thurston`	`PowerDiagram`	`FindAngles`
23	256	175	60	9	66	2
43	536	442	49	71	389	4
63	879	758	165	256	1115	23
83	1137	961	134	583	2123	24
103	1664	1330	122	1321	3922	27

Table 1. For each n, we report the average number of iterations and time (in seconds) for the `PowerDiagram` algorithm to terminate. For each of the other two algorithms, for each n, we find the largest ε, for which all realizations produced by the algorithm are power diagrams.

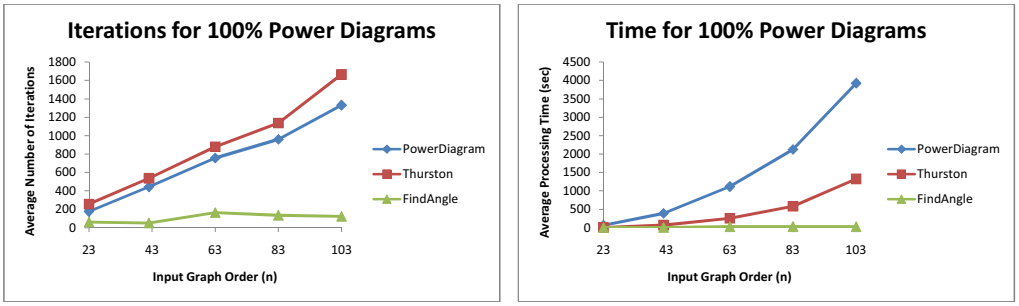

Fig. 4. These charts represent the data from Table 1.

Input Graph	`Thurston`			`FindAngles`		
Order n	Average Iterations	Average Time (sec)	Percent Power Diag	Average Iterations	Average Time (sec)	Percent Power Diag
23	309	11	100	83	3	100
43	591	78	100	104	9	100
63	879	256	100	165	23	100
83	1101	565	92	171	32	100
103	1341	1063	54	170	40	100

Table 2. This summarizes the results of the execution of the `Thurston` algorithm and Algorithm `FindAngles`. The results are for runs with $\varepsilon = 10^{-5}$. Each row represents the average results for 13 different input graphs of the order shown.

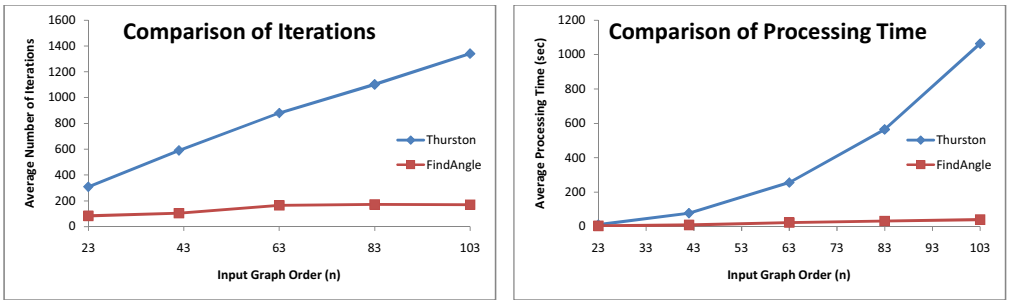

Fig. 5. These charts show the data form Table 2

	$\times 10^{-3}$					$\times 10^{-4}$					$\times 10^{-5}$					$\times 10^{-6}$				
ε	8	6	4	2	1	8	6	4	2	1	8	6	4	2	1	8	6	4	2	1
`Thurston`	0	0	0	0	2	3	6	9	17	25	26	32	40	69	89	95	97	97	100	100
`FindAngles`	11	12	14	20	34	40	45	55	74	86	92	95	98	98	100	100	100	100	100	100

Table 3. This table shows the increase in the fraction of actual Delaunay triangulations produced `FindAngles` and the `Thurston` algorithm as $\varepsilon \to 0$.

As mentioned in Section 3, we have also used sum-termination condition as an alternate termination condition for `FindAngles`. Being a bit more flexible than the max-termination condition, we expect termination to be reached in fewer iterations using sum-termination. This is confirmed in Table 4. For example, for $n = 103$ and $\varepsilon = 10^{-4}$, it takes on average, 200 iterations for `FindAngles` to terminate with the sum-termination condition. The sum-termination condition requires the sum of the "errors" at all inner vertices to be at most 10^{-4}. We interpret this constraint as being roughly equivalent to the "error" at each vertex being bounded by $10^{-4}/n = 10^{-6}$. We can then compare the 200 iterations needed to terminate with $\varepsilon = 10^{-4}$ with the sum-termination condition with the 243 iterations it takes to terminate with $\varepsilon = 10^{-6}$ under the max-termination condition. Similar comparisons can be made for each of the other values of n and we see that one can ensure roughly similar average error using slightly fewer iterations with the sum-termination condition. The salient point of this comparison is to show that convergence of our algorithm seems fairly robust and is not greatly affected by using alternate termination conditions.

	Average Number of Iterations		
n	Sum-termination $\varepsilon = 10^{-4}$	Max-termination $\varepsilon = 10^{-5}$	Max-termination $\varepsilon = 10^{-6}$
23	73	83	106
43	103	104	138
63	176	165	220
83	191	171	232
103	200	170	243

Table 4. The last three entries of each row respectively represent the average number of iterations for $\varepsilon = 10^{-4}$ under sum-termination, $\varepsilon = 10^{-5}$ under max-termination, and $\varepsilon = 10^{-6}$ under max-termination.

4.2 Scaling to Graphs of Order $n = 1000$

Our simulations demonstrate that `FindAngles` is efficient for input graphs with up to 103 vertices. To determine whether it scales to larger graphs, `FindAngles` was run on five graphs each of order $n \in \{103, 203, 303, 403, 503, 603, 703, 803, 903, 1003\}$, using $\varepsilon = 10^{-5}$. As shown in Table 5 and in Figure 6, the number of iterations grows linearly with the input size. The processing time grows a bit more rapidly and consists mainly of time spent adjusting angles. The time needed to solve the linear program is just a small fraction of the overall processing time.

Scaling to Large Inputs ($\varepsilon = 10^{-5}$)										
n	103	203	303	403	503	603	703	803	903	1003
Average Iterations	157	273	399	340	505	579	637	786	834	951
Average Time to Solve LP	0.02	0.05	0.08	0.11	0.14	0.17	0.20	0.23	0.26	0.28
Average Time to Adjust Angles	38	139	341	396	836	1171	1588	2426	3285	3711

Table 5. This table shows how our algorithm scales to inputs of size 1003. These average are taken over five inputs of each size shown.

4.3 Most triangulations are Delaunay Realizable

As mentioned in Section 1, one of the main motivations of our work is the observation by Hiroshima et al. [10] that most triangulations are Delaunay realizable. In that work the authors experimentally verified their observation for small size triangulations (with up to 12 vertices) [10]. In an effort to corroborate those findings, we randomly constructed 5000 pairwise non-isomorphic combinatorial triangulations of order $n = 103$ and used the linear program corresponding to constraints (C1)-(C5) to see how many were Delaunay realizable. The triangulations were generated by first dropping points randomly in the plane, constructing a Delaunay triangulation, "flipping" randomly chosen diagonals repeatedly, and finally extracting a combinatorial representation of the resulting triangulation. The number of diagonals flipped is determined by the parameter m. If the Markov Chain defined by this process is rapidly mixing, then for m not too large, the 5000 combinatorial triangulations would form a sample picked uniformly at random

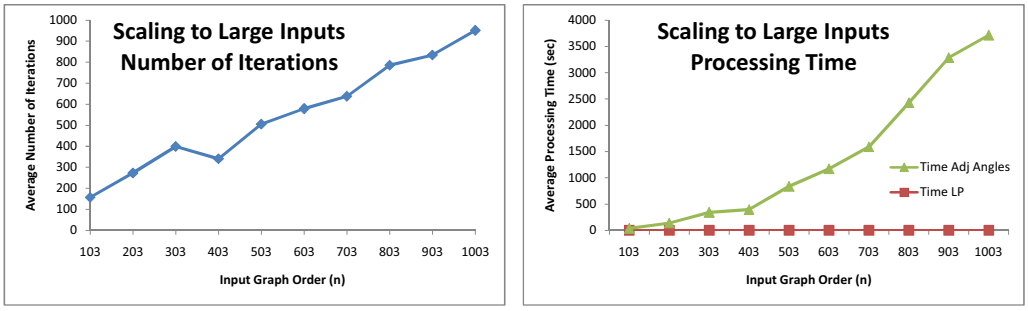

Fig. 6. These charts represent the data from Table 5.

from the set of all combinatorial triangulations on n points. In the future, we will investigate the mixing properties of this process. The result was quite satisfying and lends support to the observation of Hiroshima et al. [10]: *of the 5000 combinatorial triangulations generated, only 1 is not Delaunay realizable.*

5 Understanding the Convergence

The angle adjustment process of the `FindAngles` algorithm starts with an initial angle assignment (a feasible solution to (C1)-(C5)) and modifies the angles iteratively in an effort to satisfy constraint (C6) as well. This process can can be viewed as moving from an inconsistent angle assignment in which the triangles cannot be "glued" together, toward a valid Delaunay triangulation. As a measure of how close to a valid Delaunay triangulation the current angle assignment is after an iteration, we define *error* as $\sum_{v \in I} |1 - F(v)|$, where I is the set of all inner vertices and $F(v)$ is as defined in Section 2.1. We are interested in the behavior of this error term as the iterations progress. The faster the error term converges to zero, the sooner the iterative process terminates. Figure 7 shows that the error falls rapidly initially and then converges more slowly as the algorithm gets closer to a valid Delaunay triangulation. While the decrease in error is largely monotonic, for a small percentage of iterations the error increases slightly. In the Theorem 3 we prove that under certain initial conditions the error term strictly decreases. For any inner vertex v and real α, let $change(v, \alpha)$ be as described in Section 2.1.

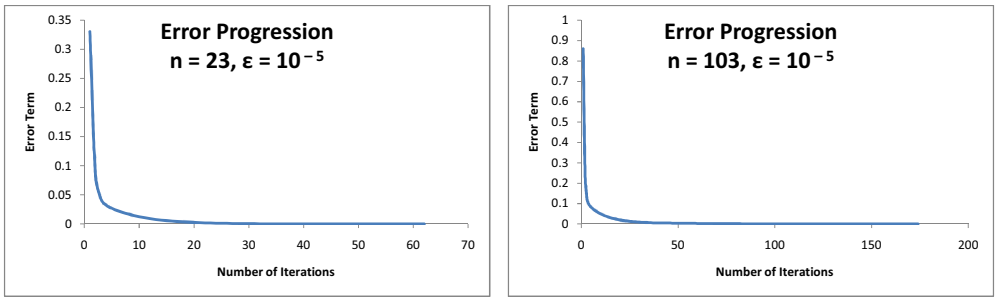

Fig. 7. This figure shows how the error term falls rapidly when the `FindAngles` algorithm is run with $\varepsilon = 10^{-5}$ on two separate input graphs, one of order $n = 23$ and one of order $n = 103$.

Theorem 3. *Let v be an inner vertex of degree s and suppose that $F(v) \neq 1$. Further suppose that $F(v)$ is not smaller than the F-values of neighbors of v. Let α^* be such that performing $change(v, \alpha^*)$ makes $F(v) = 1$. Then performing $change(v, \alpha^*)$ decreases the error.*

Proof. There are two cases, depending on whether or not $F(v) < 1$. We first assume that $F(v) < 1$. Let ϕ_i (θ_i), $0 \leq i \leq s-1$, be the cc-Facing (c-facing) angles about v (recall Figure 2). Group the terms in $F(v)$ as follows

$$F(v) = \left(\frac{\sin(\phi_0)}{\sin(\theta_1)}\right) \cdot \left(\frac{\sin(\phi_1)}{\sin(\theta_2)}\right) \cdots \left(\frac{\sin(\phi_{s-2})}{\sin(\theta_{s-1})}\right) \cdot \left(\frac{\sin(\phi_{s-1})}{\sin(\theta_0)}\right),$$

calling them $t_0, t_1, \ldots, t_{s-1}$ respectively. Note that the two angles involved in term t_j, namely ϕ_j and θ_{j+1}, are "facing" angles and therefore according to condition (C4) satisfy $\phi_j + \theta_{j+1} < 180$. Here (and in the remainder of the proof) we use modulo s arithmetic, so if $j = s-1$, then $j+1 \equiv 0$ and if $j = 0$, then $j-1 \equiv s-1$. Now let us view the quantity

$$\left(\frac{\sin(\phi_j + \alpha)}{\sin(\theta_{j+1} - \alpha)}\right)$$

as a function of α and denote it by $t_j(\alpha)$. So $t_j(0)$ is just t_j. Since $\phi_j + \theta_{j+1} < 180$, using Lemma 4.2 from Hiroshima et al. [10] we see that $t_j(\alpha)$ is monotonically increasing in the range $\alpha \in [0, \theta_{j+1})$. Therefore the product $\prod_{j=0}^{s-1} t_j(\alpha)$ is also monotonically increasing starting at

$$\prod_{j=0}^{s-1} t_j(0) = \prod_{j=1}^{s-1} t_j = F(v) < 1$$

and increasing to $+\infty$ as $\alpha \to \min\{\theta_0, \theta_1, \ldots, \theta_{s-1}\}$. Therefore there is some $0 < \alpha^* < \min\{\theta_0, \theta_1, \ldots, \theta_{s-1}\}$ such that $\prod_{j=0}^{s-1} t_j(\alpha^*) = 1$. In other words, performing the operation $change(v, \alpha^*)$ makes $F(v) = 1$.

Now fix an α, $0 < \alpha \leq \alpha^*$. Since $t_j(\cdot)$ is a monotonically increasing function in the range $[0, \alpha^*]$, each $t_j(\alpha)$ can be expressed as $t_j + \Delta_j$ for some $\Delta_j > 0$. Performing the operation $change(v, \alpha)$ results in $F(v)$ increasing from $\prod_{j=0}^{s-1} t_j$ to $\prod_{j=0}^{s-1}(t_j + \Delta_j)$. As $F(v)$ increases towards 1, we see a decrease in the contribution of v to the error by

$$\prod_{j=0}^{s-1}(t_j + \Delta_j) - \prod_{j=0}^{s-1} t_j$$

We can lower bound this quantity as follows:

$$\prod_{j=0}^{s-1}(t_j + \Delta_j) - \prod_{j=0}^{s-1} t_j > \Delta_0 \cdot \prod_{j \neq 0} t_j + \Delta_1 \prod_{j \neq 1} t_j + \cdots + \Delta_{s-1} \prod_{j \neq s-1} t_j = \sum_{j=0}^{s-1} \frac{\Delta_j}{t_j} F(v)$$

Thus the decrease in the error term due to v is greater than

$$\sum_{j=0}^{s-1} \frac{\Delta_j}{t_j} F(v). \tag{2}$$

The operation $change(v, \alpha)$ also affects $F(w_j)$ for each neighbor w_j, $0 \leq j \leq s-1$ of v. To understand the precise effect, note that each $F(w_j)$ contains the term $\frac{\sin(\theta_j)}{\sin(\phi_{j-1})}$ times other terms that do not involve any of the changed angles. Note that ϕ_{j-1}, which is a cc-facing angle about v, is a c-facing angle around w_j and similarly, θ_j, which is a c-facing angle about v, is a cc-facing angle round w_j. So $F(w_j)$ can be written as $\frac{1}{t_{j-1}} \times rest_j$, where $rest_j$ denotes the product of the other terms in $F(w_j)$. Performing the operation $change(v, \alpha)$ decreases $F(w_j)$ to

$$\frac{1}{(t_{j-1} + \Delta_{j-1})} \cdot rest_j$$

Therefore the change in $F(w_j)$ is

$$\frac{1}{t_{j-1}} \cdot rest_j - \frac{1}{(t_{j-1} + \Delta_{j-1})} \cdot rest_j = \frac{\Delta_{j-1}}{t_{j-1}(t_{j-1} + \Delta_{j-1})} \cdot rest_j = \frac{\Delta_{j-1}}{(t_{j-1} + \Delta_{j-1})} \cdot F(w_j)$$

The above change is an increase in the contribution of $F(w_j)$ to the error.

Now for each j, $0 \leq j \leq s-1$, we can "charge" the increase in the contribution to the error by $F(w_j)$ to the term $\frac{\Delta_{j-1}}{t_{j-1}} F(v)$ in Equation (2). This is because $F(v) \geq F(w_j)$, since v was chosen to maximize $F(v)$ and so we get:

$$\frac{\Delta_{j-1}}{t_{j-1}} \cdot F(v) \geq \frac{\Delta_{j-1}}{t_{j-1}} \cdot F(w_j) \geq \frac{\Delta_{j-1}}{(t_{j-1} + \Delta_{j-1})} \cdot F(w_j)$$

Thus the increase in the error due to w_j is less than the decrease in the error due to the term $\frac{\Delta_{j-1}}{t_{j-1}} F(v)$ in Equation (2). Summing this over all j, we obtain that the total increase in error due to the neighbors of v is less than the decrease in the error due to v. All of this is true for any $\alpha \in (0, \alpha^*]$ and in particular for $\alpha = \alpha^*$ as well.

The case in which $F(v) > 1$ is quite similar. Here we need to decrease $F(v)$ so we pick a negative α and perform a $change(v, \alpha)$. This results in an increase in $F(w_j)$-values, but as in the above proof the decrease in $F(v)$ decreases the error by more than total increase due to all of the $F(w_j)$'s. $\square$

6 Future Work

While the `FindAngles` algorithm presented in this paper is a significant improvement over both the `PowerDiagram` and `Thurston` algorithms, there are, nevertheless, further optimizations and extensions that should be pursued. As an initial preprocessing step `FindAngles` looks for a feasible solution to the global linear constraints (C1)-(C5) set forth by Hiroshima et al. [10]. We currently use the built in Mathematica function `LinearProgramming` to solve this LP. We would like to replace this with a strictly local test for feasibility. Since (C1)-(C5) are local constraints, such a test seems reasonable. We point out in Section 4.2 that virtually all processing time is spent adjusting angles. Hence, finding a simpler way to solve for α^* (line 9 of the `FindAngles` algorithm) may greatly improve the algorithm's overall processing time. Currently vertices whose angles are adjusted are selected arbitrarily. We may be able to improve convergence by selectively picking vertices based on some threshold of their F-values. While our algorithm is efficient, it is limited to combinatorial triangulations. We would like to extend it to run on the the less restrictive class of 3-connected planar graphs. As mentioned in Section 1, Papadimitriou and Ratajczak conjecture [16] (and Dhandapani [8] has now proved) that every such graph has a greedy embedding. We show that with certain initial conditions the error term used for termination of our algorithm decreases monotonically. Motivated by the desire to find a general polynomial time algorithm for Delaunay realizability, we would like to study more closely the behavior of this error term and identify an alternate definition of error that decreases monotonically for all inputs.

References

1. E. M. Andreev. On convex polyhedra in Lobacevskii space. *Math. USSR Sbornik*, 10(3):413–440, 1970.
2. E. M. Andreev. On convex polyhedra of finite volume in Lobachevskii space. *Math. USSR Sbornik*, 12(2):255–259, 1970.
3. F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.
4. M. Ben-Chen, C. Gotsman, and Steven Gortler. Routing with guaranteed delivery on virtual coordinates. In *Proceedings of the 18th Canadian Conference on Computational Geometry (CCCG'06)*, pages 117–120, 2006.
5. M. Ben-Chen, C. Gotsman, and C. Wormser. Distributed computation of virtual coordinates. In *Proceedings of the 23rd annual symposium on Computational geometry (SoCG'07)*, pages 210–219, New York, NY, USA, 2007. ACM Press.
6. P. Bose and P. Morin. Online routing in triangulations. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC'99)*, pages 113–122, London, UK, 1999. Springer-Verlag.
7. C. R. Collins and K. Stephenson. A circle packing algorithm. *Computational Geometry: Theory and Applications*, 25(3):233–256, 2003.
8. R. Dhandapani. Greedy drawings of triangulations. In *Proceedings of the 19th annual ACM-SIAM symposium on discrete algorithms (SODA'08) (to appear)*. SIAM, 2008.
9. H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the eighth annual symposium on Computational geometry (SoCG'92)*, pages 43–52, New York, NY, USA, 1992. ACM.
10. T. Hiroshima, Y. Miyamoto, and K Sugihara. Another proof of polynomial-time recognizability of Delaunay graphs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (IEICE'00)*, 83(4):627–638, April 2000.
11. B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'00)*, pages 243–254, 2000.
12. B. Knaster, C. Kuratowski, and S. Mazurkiewicz. Ein beweis des fixpunktsatzes für n-dimensionale simplexe. *Fundamenta Mathematicae*, 14:132–137, 1929.
13. P. Koebe. Kontatkprobleme der konformen abbildung. *Berichte über die Verhandlungen d. Sächs. Akademie der Wissenschaften Leipzia*, 88:141–164, 1936.
14. Bojan Mohar. A polynomial time circle packing algorithm. *Discrete Mathematics*, 117(1-3):257–263, 1993.
15. J. Pach and P. K. Agarwal. *Combinatorial Geometry*. John Wiles & Sons, New York, NY, USA, 1995.
16. C. H. Papadimitriou and D. Ratajczak. On a conjecture related to geometric routing. *Theoretical Computer Science*, 344(1):3–14, 2005.
17. R. Rajaraman. Topology control and routing in ad hoc networks: a survey. *SIGACT News*, 33(2):60–73, 2002.
18. I. Rivin. Euclidean structures on simplicial surfaces and hyperbolic volume. *Annals of Mathematics*, 139(3):553–580, 1994.
19. I. Rivin. A characterization of ideal polyhedra in hyperbolic 3-space. *Annals of Mathematics*, 143(1):51–70, 1996.
20. P. Santi. Topology control in wireless ad hoc and sensor networks. *ACM Computing Surveys*, 37(2):164–194, 2005.
21. W. D. Smith. Accurate circle configurations and numerical conformal mapping in polynomial time. NEC Research Institute, unpublished technical memorandum, December 1991.
22. W. P. Thurston. The geometry and topology of 3-manifolds. Princeton University Notes, 1988.

Appendix

Details of Simulation Platform

Our simulations were written using the functional programming language built into the 64-bit version of Mathematica 6.0 for Microsoft Windows. The `Combinatorica` and `ComputationalGeometry` add-on packages were used for graph modeling and Delaunay triangulations, respectively. All simulations were executed on a 2.41 GHz AMD 4000+ 64-bit Athlon Dual-Core processor running Windows XP Professional x64 Edition with 2 GB of RAM.

The timing results reported in Section 4 were obtained using the Mathematica function `Timing` and include only time spent by the Mathematica kernel executing the algorithm. They do not include time spent measuring other aspects of the algorithm (such as counting the number of iterations, etc.) or in any external processes. The Mathematica function `LinearProgramming` is used to solve the LP corresponding to conditions (C1)-(C5). This is done only once to obtain an initial feasible set of angle values. From that point we start the iterative process of adjusting the angles. The Mathematica function `FindRoot` is used to solve for an α^* in line 9 of algorithm `FindAngles`.

Example of a Koebe Representation

In Section 2.2 we describe the theory of Koebe [13], which says that given a planar graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$, there exists a packing of n disks $C = \{C_1, C_2, \ldots, C_n\}$ in the plane such that C_i and C_j touch iff $\{v_i, v_j\} \in E$ for $1 \leq i, j \leq n$. Here we provide a simple example of Koebe's theorem.

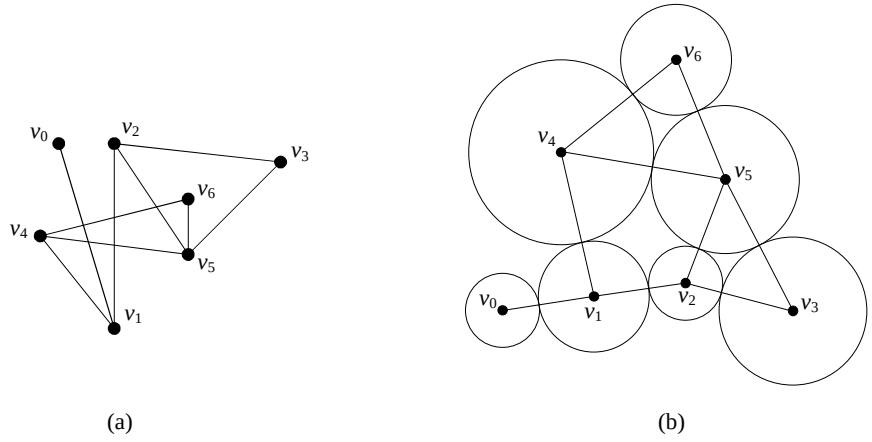

Fig. 8. (a) A planar graph $G = (V, E)$ with vertex set $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$. (b) A packing of circles $\mathcal{C} = \{C_0, C_1, C_2, C_3, C_4, C_5, C_6\}$ in the plane such that circle C_i and C_j touch each other iff edge $\{v_i, v_j\} \in E$. By Koebe's Representation Theorem [13] such a circle packing is guaranteed to exist for every planar graph.

Greedy Routing Succeeds on Delaunay Triangulations

In Section 1 we provide justification supporting the claim that greedy geographic routing is guaranteed to succeed on Delaunay triangulations. The proof of this result, which is due to Bose and Morin [6], is provided in more detail here.

Theorem. (Bose and Morin [6]) *If G is the Delaunay Triangulation of a point set $P \subset \mathbb{R}^2$, then greedy geographic routing always succeeds on G.*

Proof. Let $t \in P$, we need to show that for all $s \in P - \{t\}$ there is a point $w \in N_G(s)$ such that $\|w - t\| < \|s - t\|$ (see Figure 9). If $t \in N_G(s)$ this is trivially true, so let $t \notin N_G(s)$. Consider the Voronoi Diagram on P, and the directed line segment $\overrightarrow{st}$. Let e be the first Voronoi edge intersected by $\overrightarrow{st}$. Notice that e is on the boundary of two Voronoi cells: C_s, the cell associated with s, and C_w, the cell associated with some other point w. The line on which e lies defines two half planes: $h_s = \{p \in \mathbb{R}^2 \mid \|s - p\| < \|w - p\|\}$ and $h_w = \{p \in \mathbb{R}^2 \mid \|w - p\| < \|s - p\|\}$. Since e is the boundary of cells C_s and C_w, edge $\{s, w\} \in G$ and since $t \in h_w$, $\|w - t\| < \|s - t\|$ $\square$.

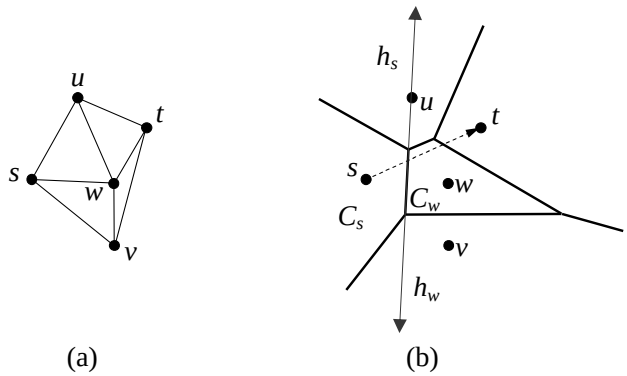

Fig. 9. (a) Input graph G, the Delaunay triangulation of the point set $P = \{s, t, u, v, w\}$. (b) The Voronoi diagram of the point set P. The directed line segment from s to t first intersects the Voronoi edge that is the boundary of Voronoi Cells C_s and C_w. The line supporting this Voronoi edge divides the plane into two half planes, h_s and h_w such that all points in h_s are closer to point s than to point w and all points in h_w are closer to point w than point s. Since $t \in h_w$, it is closer to w than to s, and since Voronoi cells C_s and C_w share a Voronoi edge, $w \in N_G(s)$, as can be seen in (a).

Tables for Other Values of ε

	$\varepsilon = .01$				$\varepsilon = .008$				$\varepsilon = .006$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	56	17	2.0	0.3	63	18	2.2	0.4	72	21	2.5	0.5
43	80	14	10	0.6	92	16	12	0.7	108	17	14	0.8
63	99	15	28	1.0	116	18	33	1.2	139	22	40	1.6
83	105	20	53	1.4	124	22	63	1.7	152	25	77	2.2
103	112	20	87	1.8	134	24	105	2.2	166	28	130	2.7
	$\varepsilon = .004$				$\varepsilon = .002$				$\varepsilon = .001$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	86	24	3.0	0.6	110	32	3.9	0.9	135	37	5	1
43	134	21	17	1.1	181	27	24	1.7	232	35	30	2
63	176	28	51	2.2	246	41	71	3.7	323	55	94	6
83	195	30	99	2.9	282	43	143	4.8	379	57	193	7
103	217	31	171	3.7	320	40	252	5.8	438	54	346	9
	$\varepsilon = .0008$				$\varepsilon = .0006$				$\varepsilon = .0004$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	144	41	5.1	1.3	154	43	5.5	1.4	170	47	6.0	1.6
43	249	40	33	2.8	271	44	36	3.1	302	49	40	3.7
63	349	61	101	6.5	382	70	111	7.6	430	79	125	9.1
83	411	62	210	8.4	455	67	232	10	516	77	264	12
103	478	59	378	9.7	531	65	420	11	608	75	481	13
	$\varepsilon = .0002$				$\varepsilon = .0001$				$\varepsilon = .00008$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	196	54	6.9	1.9	222	60	7.8	2.2	230	63	8.1	2.4
43	356	59	47	4.7	410	68	54	5.7	427	73	56	6.0
63	514	95	149	12	598	112	174	14	625	116	182	15
83	624	92	319	15	734	110	376	19	769	114	394	20
103	743	92	588	18	880	109	697	23	924	116	732	25

	$\varepsilon = .00006$				$\varepsilon = .00004$				$\varepsilon = .00002$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	241	65	8.5	2.5	256	69	9.1	2.7	282	77	10	3.0
43	450	77	59	6.4	482	83	64	7.0	536	93	71	8.1
63	660	124	192	16	709	132	206	18	794	150	231	21
83	815	124	418	22	880	134	451	24	990	151	508	28
103	982	122	778	27	1062	133	842	30	1202	153	953	35
	$\varepsilon = .00001$				$\varepsilon = .000008$				$\varepsilon = .000006$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	309	83	11	3.4	317	86	11	3.5	328	88	12	3.6
43	591	104	78	9.2	608	107	80	10	631	112	83	10
63	879	165	256	23	907	172	264	24	942	178	274	25
83	1101	171	565	32	1137	176	583	34	1183	186	607	35
103	1341	170	1063	40	1386	177	1099	42	1443	185	1145	44
	$\varepsilon = .000004$				$\varepsilon = .000002$				$\varepsilon = .000001$			
	Iterations		Time (sec)		Iterations		Time (sec)		Iterations		Time (sec)	
n	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles	Thurston	FindAngles
23	343	93	12	3.8	370	100	13	4.2	396	106	14	4.5
43	663	118	88	11	717	128	95	12	772	138	102	13
63	992	187	289	27	1077	205	314	30	1162	220	339	32
83	1248	195	640	38	1359	214	698	42	1470	232	755	46
103	1525	199	1210	47	1664	219	1321	53	1804	243	1432	60

Additional Error Term Plots

Section 5 provides plots of the decreasing error term of `FindAngles` for two simulation runs. Below are plots for runs using different size inputs and different values of ε. Because we ran over 1300 individual runs (66 input graphs with 21 different values of ε), all error term plots cannot be included here. For additional plots see `www.sau.edu/LillisKevinM/WEA2008/`.

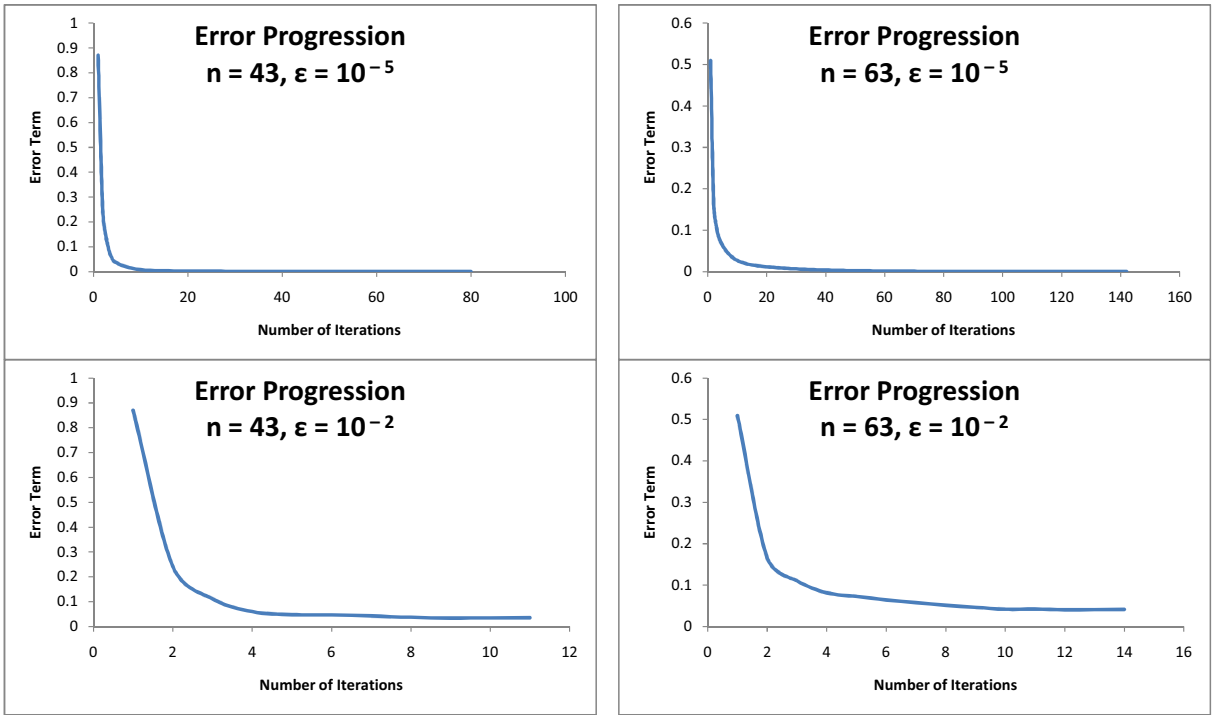

Fig. 10. Additional plots showing the decreasing error term as `FindAngles` progresses.