Fast Segment Insertion and Incremental Construction of Constrained Delaunay Triangulations

Jonathan Richard Shewchuk Brielin C. Brown

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

Abstract

The most commonly implemented method of constructing a constrained Delaunay triangulation (CDT) in the plane is to first construct a Delaunay triangulation, then incrementally insert the input segments one by one. For typical implementations of segment insertion, this method has a $\Theta(kn^2)$ worst-case running time, where n is the number of input vertices and k is the number of input segments. We give a randomized algorithm for inserting a segment into a CDT in expected time linear in the number of edges the segment crosses, and demonstrate with a performance comparison that it is faster than gift-wrapping for segments that cross many edges. A result of Agarwal, Arge, and Yi implies that randomized incremental construction of CDTs by our segment insertion algorithm takes expected $O(n \log n + n \log^2 k)$ time. We show that this bound is tight by deriving a matching lower bound. Although there are CDT construction algorithms guaranteed to run in $O(n \log n)$ time, incremental CDT construction is easier to program and competitive in practice. Moreover, the ability to incrementally update a CDT by inserting a segment is useful in itself.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

Keywords: constrained Delaunay triangulation; ϵ-net; randomized incremental construction; computational geometry

1 Introduction

The constrained Delaunay triangulation (CDT) in the plane, proposed by Lee and Lin [17], is a variant of the well-known Delaunay triangulation in which specified edges, sometimes called *segments*, are constrained to appear. The CDT is as close to being Delaunay as possible subject to those constraints. In particular, every edge of the CDT either is an input segment or is *locally Delaunay*; the latter means that if we consider only the three or four vertices of the one or two triangles that include the edge, the Delaunay triangulation of those vertices contains the edge, as illustrated at left in Figure 1.

Supported in part by the National Science Foundation under Awards CCF-0635381 and IIS-0915462, in part by the University of California Lab Fees Research Program under Grant 09-LR-01-118889-OBRJ, and in part by an Alfred P. Sloan Research Fellowship.

SoCG'13, June 17–20, 2013, Rio de Janeiro, Brazil.

The constraints imposed by CDTs have many uses, including representing boundaries of nonconvex objects, permitting better interpolation of discontinuous functions, and aiding the enforcement of boundary conditions in finite element meshes.

The input to a CDT construction algorithm is a *planar straight line graph* (PSLG), illustrated in Figure 2. A PSLG $\mathcal{X}$ is a set of vertices and segments (constraining edges) that satisfy two restrictions: both endpoints of every segment in $\mathcal{X}$ are members of $\mathcal{X}$, and a segment in $\mathcal{X}$ may intersect other segments and vertices in $\mathcal{X}$ only at its endpoints. We seek a triangulation of the vertices in $\mathcal{X}$ that includes every segment in $\mathcal{X}$.

We assume that the reader is familiar with Delaunay triangulations [10, 6]. Throughout this paper, every triangulation is understood to be a simplicial complex; thus it is a set containing vertices, edges, and triangles. A CDT of $\mathcal{X}$ is a triangulation of the vertices in $\mathcal{X}$ in which every triangle is *constrained Delaunay*. A triangle is constrained Delaunay if its interior intersects no segment in $\mathcal{X}$ and its circumcircle encloses no vertex in $\mathcal{X}$ that is *visible* from any point in the triangle's interior, as illustrated at right in Figure 1. Two points are visible to each other if the open line segment joining them does not intersect a segment in $\mathcal{X}$. By the Delaunay Lemma [10, 17], a triangulation of a vertex set is a CDT (has all its triangles constrained Delaunay) if and only if it contains every segment and all its other (non-segment) edges are locally Delaunay.

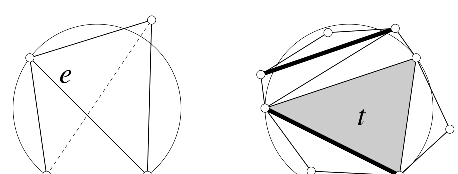

Figure 1. Left: the bold edge e is locally Delaunay. The dashed edge that crosses it is not. Right: The triangle t is constrained Delaunay, despite having two vertices inside its circumcircle. Bold lines represent segments.

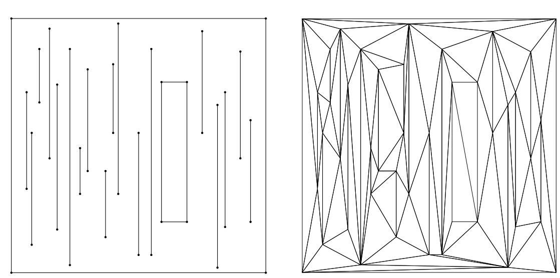

Figure 2. A planar straight line graph and its constrained Delaunay triangulation.

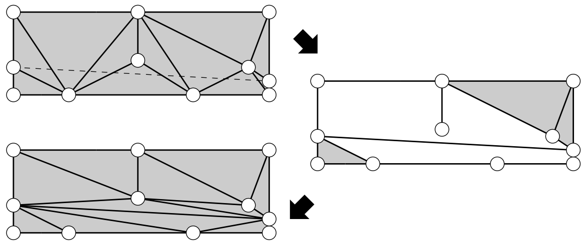

Figure 3. Inserting a segment into a constrained Delaunay triangulation.

Two algorithms are known that construct the CDT of a PSLG with n vertices in $O(n \log n)$ time, which is optimal in the decision-tree model of computation. One is a divide-and-conquer algorithm by Chew [7]. Its lineage stretches back to the first Delaunay triangulation algorithm to run in $O(n \log n)$ time, the 1975 divide-and-conquer algorithm of Shamos and Hoey [22], which was subsequently simplified and elaborated by Lee and Schachter [18] and Guibas and Stolfi [13]. The other is a sweepline algorithm by Seidel [20], which generalizes a Delaunay triangulation algorithm of Fortune [12] to CDTs.

Both algorithms are rarely implemented, perhaps because they are complicated. In practice, the only widely used CDT construction algorithm begins by constructing an ordinary Delaunay triangulation first, then inserts the segments into the triangulation one by one. To *insert a segment* is to delete all the edges and triangles that intersect its relative interior, create the new segment, and retriangulate the two polygonal cavities thus created (one on each side of the segment) with constrained Delaunay triangles, as illustrated in Figure 3. The cavities might not be simple polygons, because they might have edges dangling in their interiors, as shown.

In many implementations, each segment is inserted by a naive algorithm that takes $O(m^2)$ time where m is the number of triangles whose interiors intersect the segment, yielding a CDT construction algorithm that takes $\Theta(kn^2)$ time for some PSLGs with n vertices and k segments. See Anglada [3] for a typical segment insertion algorithm that usually takes $\Theta(m^2)$ time, though it can achieve $\Theta(m \log m)$ best-case time when it has good luck with evenly subdividing the cavities. (Anglada's algorithm is a variant of well-known gift-wrapping algorithms; it gift-wraps from the new segment out.) Chew's and Seidel's algorithms can insert a segment in $O(m \log m)$ time, but algorithms like Anglada's are much easier to implement.

Despite its asymptotic time disadvantage, the incremental segment insertion algorithm is popular for good reasons: it leverages the best existing implementations of (unconstrained) Delaunay triangulation algorithms; it is easier to implement than other CDT algorithms; its speed is tolerated in practice because many real-world inputs have few or no segments that cross many edges; and the ability to dynamically update a CDT by inserting a new segment is itself useful—for instance, in applications that support interactive geometric modeling. Moreover, Agarwal, Arge, and Yi [2] show that if the k segments are inserted in random order, the expected number of edges and triangles deleted, summed over all segment insertions, is in $O(n \log^2 k)$, compared to a deterministic worst case of $\Theta(kn)$.

This paper presents a randomized algorithm for inserting a segment into a CDT in expected $O(m)$ time, where m is the number of triangles whose interiors intersect the segment. The algorithm is simple enough to compete with naive algorithms like gift-wrapping for ease of programming. We provide pseudocode, which we turned into working C code in five hours.

We also show a matching $\Omega(n \log^2 k)$ lower bound on the number of structural changes, which resolves the long-standing question of the expected complexity of uniformly randomized incremental segment insertion on worst-case PSLGs.

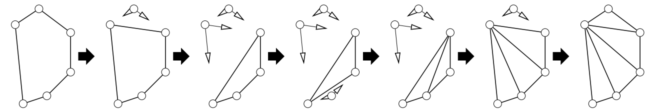

Figure 4. Chew's algorithm for computing the Delaunay triangulation of a convex polygon deletes vertices from the polygon in a random order to precompute the information needed for point location, then inserts the vertices in the opposite order.

Our third contribution is to analyze a simple algorithm for *segment location*—finding the first triangle deleted when a segment is inserted—and to show that it is fast enough.

With linear-time segment insertion, the randomized incremental segment insertion algorithm constructs an n-vertex, k-segment CDT in expected $O(n \log n + n \log^2 k)$ time. Although this running time falls short of optimality by a factor of $\log^2 k / \log n$, experience with incremental CDT construction software shows that segment insertion does not dominate the cost of constructing the initial Delaunay triangulation unless segments that intersect many edges are inserted by quadratic-time algorithms. Incremental segment insertion is likely to remain the most used CDT construction algorithm long into the future, so we think it is important to provide an understanding of its performance and how to make it run fast.

2 Chew's Delaunay Vertex Deletion Algorithm

Our segment insertion algorithm is closely related to an algorithm of Paul Chew [8] for deleting a vertex from a Delaunay triangulation in expected $O(m)$ time, where m is the degree of the deleted vertex. The latter algorithm is a good preparation for understanding the former, more complicated algorithm. The former also uses the latter as a subroutine.

Vertex deletion is an operation that updates a Delaunay triangulation so it has one less vertex and is still Delaunay. Chew's algorithm can delete vertices from CDTs as well.

For simplicity, consider the problem of constructing the Delaunay triangulation of a convex polygon. Chew's algorithm is a random incremental insertion algorithm that inserts one vertex at a time into the Delaunay triangulation. The same algorithm, with no changes, can also retriangulate the cavity evacuated when a vertex is deleted from a Delaunay triangulation, even though the cavity might not be convex. We will not justify that claim here, except to point out that Chew's algorithm is a disguised algorithm for deleting a vertex from a three-dimensional convex hull [16], which is related by the lifting map [5, 19, 11] to deleting a vertex from a two-dimensional Delaunay triangulation.

Let V be a sequence listing the m vertices of a convex polygon in counterclockwise order. The algorithm begins by generating a random permutation of V that dictates the order in which the vertices will be inserted. It constructs a triangle from the first three vertices of the permutation, then inserts the remaining vertices one by one.

Just before a vertex u is inserted, it lies outside the growing triangulation, but only one triangulation edge vw separates u from the triangulation's interior. *Point location* is the task of identifying the edge vw. Next, the algorithm inserts u by first identifying and deleting all the triangles whose circumcircles enclose u, which can be found quickly by a depth-first search from vw. Then, by extending new edges from u, it retriangulates the cavity formed by taking the union of the deleted triangles and $\triangle uvw$, as illustrated in the right half of Figure 4. This is essentially the *Bowyer–Watson algorithm* [4, 15, 24] for inserting a vertex into a Delaunay triangulation.

The cleverest aspect of Chew's algorithm is how it performs point location. It does all point location in advance, before constructing any triangles, by imagining the incremental insertion algorithm running backward in time. Specifically, imagine taking the input polygon and removing vertices one by one, reversing the random permutation of V, yielding a shrinking sequence of convex polygons as illustrated in the left half of Figure 4. Removing a vertex u has the effect of joining its neighbors v and w with an edge vw, which is the edge that later will be sought for point location.

The algorithm maintains a circularly-, doubly-linked list of vertices representing the polygon. It walks through a random permutation of V in backward order, removing vertices from the circularly-linked list until only three remain. The algorithm constructs a triangle from the three surviving vertices, then inserts the other vertices in the permutation of V in forward order.

Theorem 1. *Given an m-vertex polygon, Chew's algorithm runs in expected $O(m)$ time.*

Proof. The point location stage runs in deterministic $O(m)$ time. The expected running time of the vertex insertion stage is determined by *backward analysis*, an analysis technique that Seidel [21] summarizes thus: "*Analyze an algorithm as if it was running backwards in time, from output to input.*"

Every triangulation of an m-vertex polygon has $2m - 3$ edges, each with two endpoints. Imagine deleting a vertex chosen uniformly at random; in expectation it adjoins fewer than four edges.

With the algorithm running forward in time, the cost of inserting the last vertex is proportional to the number of edges that adjoin it after it is inserted. The expected number of those edges is less than four. The same reasoning holds for the other vertices. Summing this cost over all the vertices yields an expected linear running time. □

3 Inserting a Segment into a CDT

To "insert a segment into a CDT" is to take as input a CDT of a PSLG $\mathcal{X}$ and a new segment s to insert, and produce a CDT of $\mathcal{X} \cup \{s\}$. It is only meaningful if $\mathcal{X} \cup \{s\}$ is a valid PSLG—that is, $\mathcal{X}$ already contains the endpoints of s (otherwise, they must be inserted first), and the relative interior of s intersects no segment or vertex in $\mathcal{X}$. This section presents a segment insertion algorithm similar to Chew's algorithm. Its expected running time is linear in the number of edges the segment crosses.

Let $\mathcal{T}$ be a CDT of $\mathcal{X}$. If $s \in \mathcal{T}$, then $\mathcal{T}$ is also the CDT of $\mathcal{X} \cup \{s\}$. Otherwise, the algorithm begins by performing *segment location*: identifying a triangle in $\mathcal{T}$ that adjoins an endpoint of s and whose interior intersects s. This can be done with a simple rotary traversal of the triangles adjoining the endpoint of s with lesser degree. In Section 7, we show that this method never increases the asymptotic running time of CDT construction.

Once one triangle whose interior intersects s is found, the others can be identified by a simple walk in time linear in their number. The algorithm deletes these triangles from $\mathcal{T}$. All the other triangles in $\mathcal{T}$ remain constrained Delaunay after s is inserted. Next, the algorithm adds s to the triangulation and retriangulates the two polygonal cavities on each side of s with constrained Delaunay triangles, as illustrated in Figure 3.

Let P and $\hat{P}$ be the two polygonal cavities; their edges include s. The random incremental insertion algorithm CavityCDT in Figure 5 retriangulates P, and a second call to CavityCDT retriangulates $\hat{P}$. Be forewarned that CavityCDT *cannot* compute the CDT of an arbitrary polygon; it depends upon the special nature of the cavities evacuated by segment insertion for its correctness.

```
CavityCDT(V)
{ V = ⟨v_0, v_1, ..., v_{m−1}⟩ is a sequence of vertices in counterclock- }
{ wise order around a cavity evacuated when the segment v_0 v_{m−1} }
{ is inserted. Some vertices in the sequence may be duplicated if }
{ there are dangling segments in the cavity. }
1  for i ← 1 to m − 2
2      next[i] ← i + 1
3      prev[i] ← i − 1
       { Proportional to the distance from v_i to line v_0 v_{m−1} }
4      distance[i] ← Orient(v_0, v_i, v_{m−1})
       { π[1 ... m − 2] will always be a permutation of 1 ... m − 2 }
5      π[i] ← i
6  distance[0] ← 0; distance[m − 1] ← 0
7  for i ← m − 2 downto 2
       { Don't delete a vertex closer to line v_0 v_{m−1} than both neighbors }
8      repeat
9          j ← a random integer in [1, i]
10     while distance[π[j]] < distance[prev[π[j]]] and
               distance[π[j]] < distance[next[π[j]]]
11     Swap π[i] with π[j]
       { Point location: take vertex π[i] out of the linked list }
12     next[prev[π[i]]] ← next[π[i]]
13     prev[next[π[i]]] ← prev[π[i]]
14 CreateTriangle(v_0, v_{π[1]}, v_{m−1})      { Create the first triangle }
15 for i ← 2 to m − 2
16     InsertVertex(v_{π[i]}, v_{next[π[i]]}, v_{prev[π[i]]})
17     if v_{π[i]} has been marked
18         Use Chew's algorithm to retriangulate the fan of
                 triangles that have all three vertices marked
19         Unmark all the marked vertices

InsertVertex(u, v, w)
{ u is a new vertex. Is the triangle △uvw constrained Delaunay? }
20 x ← Adjacent(w, v)      { △wvx on other side of vw from u }
21 if x = ∅ or (InCircle(u, v, w, x) ≤ 0 and Orient(u, v, w) > 0)
22     CreateTriangle(u, v, w)      { △uvw is cons. Delaunay }
23 else      { △uvw and △wvx are not constrained Delaunay }
24     DeleteTriangle(w, v, x)      { Flip vw → ux }
25     InsertVertex(u, v, x)
26     InsertVertex(u, x, w)
27     if InCircle(u, v, w, x) ≤ 0      { Reuse Line 21 computation }
28         Mark vertices u, v, w, and x to be retriangulated later
```

Figure 5. Expected linear-time algorithm for retriangulating a cavity evacuated by inserting a segment into a constrained Delaunay triangulation. The 2×2 determinant $\text{Orient}(u, v, w) = \det[u - w \;\; v - w]$ is positive if u, v, and w occur in counterclockwise order. The 3×3 determinant $\text{InCircle}(u, v, w, x) = \det\begin{bmatrix} u-x & v-x & w-x \\ |u-x|^2 & |v-x|^2 & |w-x|^2 \end{bmatrix}$ is positive if x is enclosed by the circle passing through the positively oriented vertices u, v, and w. A triangulation data structure permits $\text{Adjacent}(w, v)$ to look up the third vertex of the triangle adjoining an oriented edge wv in $O(1)$ expected time. The operations CreateTriangle and DeleteTriangle add and remove positively oriented triangles in expected $O(1)$ time. These running times can be achieved with a hash table that maps edges to triangles; our implementation achieves them with a tree data structure.

CavityCDT differs from Chew's algorithm in several ways to account for the fact that P is not always convex. First, the vertices of the segment s are inserted first. Second, P might have segments dangling in its interior, like the segment connecting vertices 5 and 6 in Figure 6. In this case, imagine an ant walking a counterclockwise circuit of P's interior without crossing any edges; it will visit one or more vertices of P twice. Split each such vertex into two copies and pretend they are two separate vertices, like vertices 5 and 7 in the figure. (In rare circumstances, there may be three or more copies.)

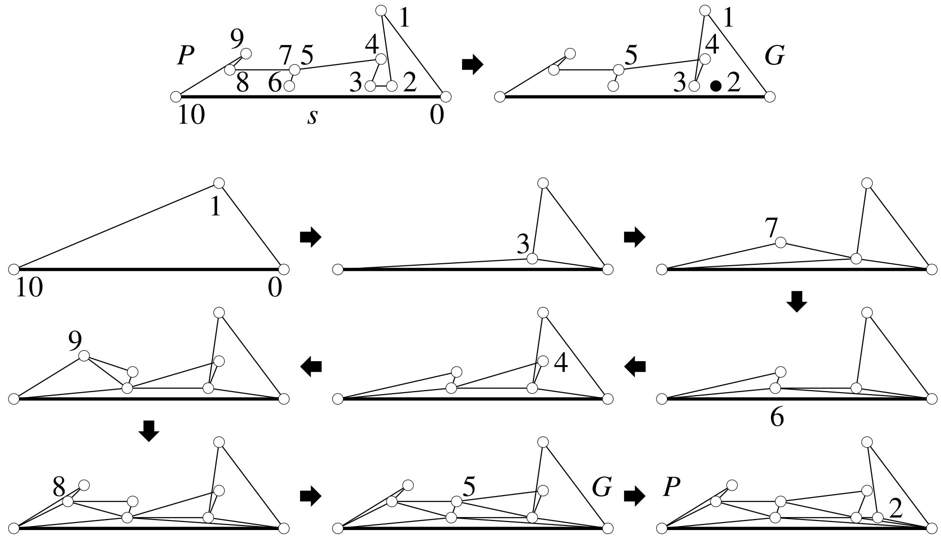

Figure 6. Computing the constrained Delaunay triangulation of a cavity obtained by inserting a segment s. The cavity has a repeated vertex, numbered 5 and 7, because of the dangling segment adjoining it. The deletion of vertex 2 creates a self-intersection, but the algorithm works correctly anyway.

Third, CavityCDT maintains the invariant that after each vertex insertion, the computed triangulation is the CDT of the polygon whose boundary is the subsequence of vertices inserted so far; we call this polygon a *subpolygon*. Because CavityCDT maintains a CDT and not merely a Delaunay triangulation, a newly inserted vertex sometimes causes a triangle to be deleted not because the new vertex lies inside the triangle's circumcircle, but because the two polygon edges adjoining the new vertex cut through the triangle; for example, the insertion of vertex 8 in Figure 6 deletes a triangle whose circumcircle does not enclose vertex 8. Line 21 of InsertVertex accounts for this possibility with an orientation test.

Unlike in Chew's algorithm, the insertion of a vertex u can create new triangles that do not adjoin u, as illustrated in Figure 7. The three shaded triangles in the top triangulation must be deleted when u is inserted, but u is not inside their circumcircles. We call triangles with this property *crossed triangles*. After u's insertion, the corresponding new triangles (shaded in the bottom triangulation) may include some that do not adjoin u.

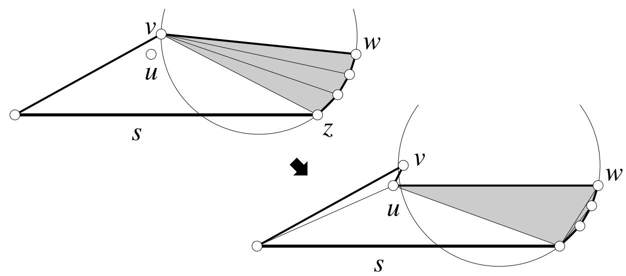

Figure 7. The shaded triangles in the top triangulation are deleted by u's insertion despite u not being inside their circumcircles. The corresponding shaded triangles in the bottom triangulation include two that do not adjoin u. The latter shaded triangles are computed by Chew's algorithm.

CavityCDT inserts u in a manner that initially connects u to all the vertices of all the deleted triangles, then subsequently uses Chew's original algorithm to correctly retriangulate the shaded region (in expected linear time). We observe few crossed triangles in practice, so the overhead of occasionally invoking Chew's algorithm is unlikely to have much influence on the running time. An alternative, easier to implement, is for CavityCDT to call itself recursively with uw serving as the segment, but it is unknown whether this option preserves the expected linear running time in theory.

Fourth, a subpolygon might be *self-intersecting*. Observe in Figure 6 that deleting vertex 2 from the cavity P creates a subpolygon G in which the edge connecting vertices 1 and 3 crosses the edge connecting vertices 4 and 5, and the subpolygon's interior angle at vertex 3 exceeds 360°. By some senses, G is not actually a polygon, although it is a polygon in the conventional sense of a looped chain of edges; and its triangulation in Figure 6 (bottom center) is not a simplicial complex, because it has triangles that overlap each other. Fortunately, it is like a CDT in two respects: it has all the combinatorial properties of a triangulation of a polygon—for example, its dual graph is a tree—and every edge is locally Delaunay.

The incremental vertex insertion algorithm works correctly even when these self-intersecting subpolygons arise, subject to one caveat: it will not correctly insert a vertex at which the polygon's internal angle is 360° or greater. For example, it cannot create P in Figure 6 by inserting vertex 6 last, nor create G by inserting vertex 3 last. These vertex insertions are anticipated and averted during the algorithm's point location step, when random vertices are deleted from P one by one until the subpolygon is reduced to a triangle. Hence, the random permutation by which the vertices are inserted is not chosen uniformly from all permutations of the vertices.

For the sake of speed, CavityCDT does not compute internal angles. Instead, let $\bar{s}$ be the line that includes the segment s. It is a property of the cavities created by segment insertion that a subpolygon vertex can have an internal angle of 360° or greater only if that vertex is closer to $\bar{s}$ than both its neighbors on the subpolygon chain. (We will justify this claim shortly.) CavityCDT declines to delete from P any vertex with the latter property (see Line 10).

Line 4 of CavityCDT computes the distance of each vertex of P from $\bar{s}$. The point location step (Lines 7–13) deletes vertices from P one by one, choosing uniformly at random from all the vertices that are not endpoints of s and are not closer to $\bar{s}$ than both their neighbors.

Be forewarned that CavityCDT cannot use the same triangulation data structure as the triangulation in which the segment s is being inserted, because CavityCDT sometimes temporarily creates triangles that conflict with those outside the cavity. For example, in Figure 6 the triangulation outside the cavity probably includes an

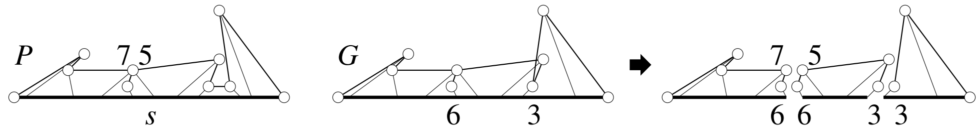

Figure 8. At left, each vertex has a sightline. Note that vertices 5 and 7 have different sightlines, although they are really a single repeated vertex. A self-intersecting polygon (center) can be interpreted topologically as several polygons glued together along their sightlines (right).

edge connecting vertices 7 and 9 and two adjoining triangles. CavityCDT temporarily creates a third triangle with this edge when it first inserts vertex 9. To avoid corrupting the data structure, CavityCDT requires the use of a separate, initially empty triangulation data structure, and the final triangles must subsequently be copied to the main triangulation.

4 The Speed and Correctness of CavityCDT

Theorem 2. *Given an m-vertex cavity,* CavityCDT *runs in expected $O(m)$ time.*

Proof. The expected cost of InsertVertex is proportional to the number of edges adjoining the newly inserted vertex u plus the number of newly created triangles that do not adjoin u. We bound these numbers separately, both by backward analysis.

Every triangulation of an m-vertex polygon has $m-2$ triangles and $m-3$ interior edges. At least $(m-1)/2$ vertices are eligible to be the first vertex deleted during point location and the last vertex inserted. One of those vertices is chosen uniformly at random; in expectation it adjoins at most $2(m-3)/((m-1)/2) < 4$ interior edges, thus fewer than 6 edges total.

When the insertion of a new vertex u causes the creation of a triangle t that does not adjoin u, as illustrated in Figure 7, it happens because a new subpolygon edge adjoining u entirely crosses t's circumcircle and hides one or more vertices inside t's circumcircle. Consider CavityCDT running backward in time; what is the probability that a triangle t will be deleted during the deletion of a randomly chosen vertex not adjoining t? There are at most three subpolygon edges—one for each edge of t—whose deletion could expose t to a vertex, deleting t. (In fact there is at most one such edge, but we do not prove it here.) The probability that the vertex chosen for deletion is one of the endpoints of one of those three edges is at most $12/(m-1)$. Therefore, the expected number of deleted triangles not adjoining u is less than 12.

Thus, forward in time, the expected cost of inserting each vertex is constant; summing this cost over all the vertices yields an expected linear running time. □

Before proving the correctness of the algorithm CavityCDT, we must make sense of self-intersecting polygons and their triangulations. The original cavity P has a special property that permits CavityCDT to work despite the possibility of self-intersecting subpolygons: P is included in a union of triangles that cross the segment s. Therefore, each vertex v_i has a *sightline* b_i: a line segment connecting v_i to s strictly through P's interior, depicted in Figure 8. We choose each sightline to be a subset of an edge that was deleted to make way for s; these sightlines do not intersect each other.

A self-intersecting subpolygon G can be understood as a topological space defined by gluing non-self-intersecting quadrilaterals and triangles together along the sightlines, as illustrated. We call these constructions *multisheet polygons* because they can be assembled from multiple sheets of paper taped together. In such a multisheet polygon, two points with the same Euclidean coordinates are not necessarily the same point; they may be in different, overlapping quadrilaterals or triangles. We triangulate a multisheet polygon by subdividing it into triangles that form the topological space G when they are glued together along shared edges. We call this triangulation a CDT if all its edges are locally Delaunay.

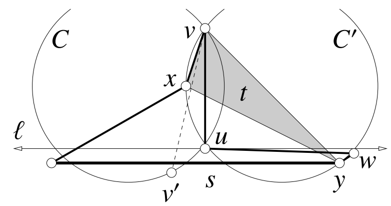

Figure 9. An impossible configuration: upon the insertion of u, a crossed triangle t has a vertex x above ℓ that is not u's neighbor on the subpolygon chain.

One consequence of G's sightlines is that for any vertex v of G at which the internal angle is 360° or greater (e.g. vertex 3 or 6 in Figure 8), both its neighbors on G's boundary chain are further from the affine hull of s than v—each because the other neighbor has a valid sightline. This observations justifies the algorithm's use of the latter property to screen out vertices with the former property in Lines 8–10.

Recall that when the insertion of a vertex u causes the deletion of crossed triangles whose circumcircles do not enclose u, we use Chew's algorithm to fix part of the triangulation around u. The correctness of this procedure follows from the fact that the crossed triangles always form a fan adjoining a single vertex v, as illustrated in Figure 7 (see Lemma 3). If we imagine adding the triangle $\triangle vuz$ to the fan, the edge vz is locally Delaunay. The key precondition for Chew's algorithm to correctly retriangulate a cavity is that there exists a point (here, v) such that, if it is connected by edges to the cavity's vertices, all these edges are locally Delaunay (on the lifting map, v is lifted to the apex of a convex cone). Chew's algorithm produces the triangles shaded in the bottom half of the figure.

To see that the crossed triangles always form this fan configuration, suppose without loss of generality that s is horizontal, as in Figure 9, with the cavity above s and the vertex indices increasing from 0 at the right endpoint of s to $m-1$ at the left endpoint. Let ℓ be the horizontal line through u (parallel to s). By design, when u is inserted, its two neighbors (v and w) on the subpolygon G cannot both be above ℓ. Suppose without loss of generality that u's neighbor w, whose index is less than u's index, is on or below ℓ. Because there is a crossed triangle, u is a concave vertex of G, so u's other neighbor v (with greater index) must be strictly above ℓ.

Lemma 3. *Given the suppositions stated above, the crossed triangles form a fan of triangles sharing vertex v, and their other vertices have indices less than u's index.*

Proof. Let t be a crossed triangle. It has at least one vertex x whose index is greater than u's and one vertex y whose index is less. Suppose for the sake of contradiction that $x \neq v$, as Figure 9 shows. Either $y = w$ or the triangle edge xy crosses the subpolygon edge uw, so y lies on or below ℓ. As xy also crosses the subpolygon edge uv, x lies strictly above ℓ.

Before s was inserted, v was connected by an edge of the CDT to a vertex v' on the other side of s, so there existed a circle C through v and v' that enclosed no vertex visible from the interior of the edge vv'. The portion of vv' terminating at s is a sightline for v in the original cavity P and in every subpolygon of P. Both u and x are visible within P from the point $vv' \cap s$, so neither u nor x is inside C. The edge vv' separates x from u and y, so there is a point z where vv' intersects the interior of t, and v is visible from z. As t is constrained Delaunay (just before u is inserted), v is not inside t's circumcircle C'. As t is a crossed triangle, u is not inside C' either.

The vertices v and x are above ℓ, and v' and y are below or on ℓ, so both circles C and C' (circumscribing vv' and xy) intersect ℓ, as illustrated. Neither circle encloses u; C intersects ℓ to the left of u, and C' intersects ℓ to the right of u (both possibly touching u). The edges vv' and xy cross each other above ℓ, but C cannot enclose x (as it does in the figure), and C' cannot enclose v. This is impossible; the result follows by contradiction. □

Our correctness proof for CavityCDT relies on a constrained version of the famous *Delaunay Lemma*. In 1934, Boris Delaunay [10] showed that a triangulation of a point set is Delaunay (has every triangle Delaunay) if and only if every edge is locally Delaunay. In 1986, Lee and Lin [17] showed that a triangulation of a PSLG is constrained Delaunay (has every triangle constrained Delaunay) if and only if every edge is locally Delaunay except perhaps the PSLG segments.

This Constrained Delaunay Lemma extends to a multisheet polygon G as follows. Two points $p, q \in G$ are said to be *visible* to each other if there is a path from p to q in G that is a straight line segment. Observe that two distinct points in G can lie at the same position in Euclidean space yet not be visible to each other, because the shortest path connecting them in G goes around a corner and through a sightline. With this definition of "visible," the definition of "constrained Delaunay triangle" in the Introduction applies to triangulations of multisheet polygons. It is easy to extend the proof of Lee and Lin to show that a triangulation of a multisheet polygon has every triangle constrained Delaunay if and only if every edge is locally Delaunay. Our proof uses the constrained Delaunay property of the triangles before each vertex insertion as a precondition to guarantee the locally Delaunay property of the edges after each vertex insertion. The latter property implies that the former holds at the beginning of the next vertex insertion.

Theorem 4. CavityCDT *correctly constructs the CDT of a cavity evacuated by inserting a segment into a CDT.*

Proof. Let $\mathcal{T}'$ and $\mathcal{T}$ be the triangulation before and after a top-level call to the recursive procedure InsertVertex inserts a new vertex u and Line 18 of CavityCDT fixes the marked triangles (if necessary). Let G' and G be the corresponding subpolygons without and with u. We will show that if $\mathcal{T}'$ is a CDT of G', then $\mathcal{T}$ is a CDT of G. The result follows by induction on the sequence of vertex insertions.

It is straightforward to check that a top-level call to InsertVertex is equivalent to gluing a new triangle $\triangle uvw$ onto an edge vw of $\mathcal{T}'$, then repeatedly checking each edge that is opposite the new vertex u in some triangle (initially the edge vw, later other edges that are exposed by flips), and flipping any such edge that is not locally Delaunay or forms a *fold* in the triangulation (Line 21 of InsertVertex). Every edge created by a flip has the vertex u, and every vertex of every triangle deleted by a flip gets connected to u.

Each InsertVertex call maintains the invariant that the set of stored triangles forms a combinatorial triangulation of a polygon—specifically, the dual graph of the triangulation is a tree whose leaves correspond to the edges of the polygon in circular order. Gluing on a new triangle is equivalent to replacing a leaf of the dual tree with a new degree-three node and two new leaves. Edge flips correspond to tree rotations.

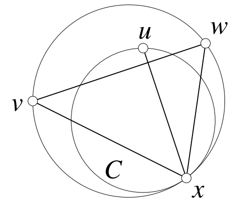

Figure 10. As $\triangle wvx$ was constrained Delaunay before u was inserted, ux is locally Delaunay.

If gluing $\triangle uvw$ onto $\mathcal{T}'$ does not create a fold at vw, then the modified triangulation is immediately a triangulation of G. Otherwise, pretend that the algorithm flips only fold edges until none survive, then flips edges that are not locally Delaunay. (CavityCDT actually flips the edges in a different order, but this does not change the final product.) Observe that u is safely confined between the sightlines of v and w; no other vertex lies between these sightlines, so no vertex can lie inside $\triangle uvw$. It is straightforward to show that the flips of the fold edges effectively remove $\triangle uvw$ from G' (and that there is only one fold edge at any given time), yielding G. Subsequent flips do not change the underlying topological space of the triangulation, so it remains a triangulation of G.

To show that the updated triangulation is constrained Delaunay, we show that all its edges are locally Delaunay. Let ux be an edge created because the InCircle test in Line 21 found that the new vertex u is inside the circumcircle of a triangle $\triangle wvx \in \mathcal{T}'$. We wish to show that ux is locally Delaunay in $\mathcal{T}$; suppose for the sake of contradiction it is not. Let C be the circle that passes through u and x and is tangent to the circumcircle of $\triangle wvx$ at x, as illustrated in Figure 10. Observe that the circumcircle entirely encloses C except at x. Because ux is not locally Delaunay in $\mathcal{T}$, $\mathcal{T}$ contains a triangle uxy or xuy that has the edge ux and a third vertex y enclosed by C. But the presence of $\triangle uxy$ in $\mathcal{T}$ implies that y is visible from some point in the relative interior of $ux \cap \triangle wvx$, and therefore $\mathcal{T}'$ was not constrained Delaunay, a contradiction.

Thus ux is locally Delaunay in $\mathcal{T}$. More generally, every edge adjoining u because of the Incircle test in Line 21 is locally Delaunay. Every edge that adjoins the same two triangles in $\mathcal{T}$ it adjoined in $\mathcal{T}'$ is locally Delaunay because it was locally Delaunay in $\mathcal{T}'$. Every edge that adjoins one old triangle surviving from $\mathcal{T}'$ and one triangle new to $\mathcal{T}$ is locally Delaunay either because the Incircle test in Line 21 chose not to flip it or because the new triangle replaces a crossing triangle after a vertex stopped being visible. Every edge on the boundary of G is locally Delaunay because it is an edge of only one triangle. The only edges that fit none of these categories are the new edges created by Line 18 of CavityCDT, which are locally Delaunay by the correctness of Chew's algorithm. Therefore, every edge in $\mathcal{T}$ is locally Delaunay, and by the Constrained Delaunay Lemma, $\mathcal{T}$ is a CDT. □

5 A Comparison of Two Cavity Retriangulation Implementations

We have implemented CavityCDT and Anglada's gift-wrapping algorithm [3] so that we could compare their speeds on cavities of different sizes. Both implementations use Shewchuk's floating-point Orient and InCircle predicates [23], which are fast despite being robust. We timed two kinds of cavity, illustrated in Figure 11:

	vertices	Anglada time (μs)	Anglada InCircle tests	CavityCDT time (μs)	CavityCDT InCircle tests	CavityCDT Orient tests (Line 21 only)
Cavities in	10	0.41	16	1.29	13.02	7.78
which most	30	4.20	196	4.20	67.07	43.01
vertices are	100	47.27	2,401	14.14	271.00	178.14
collinear	1,000	4,748.62	249,001	152.65	2,959.48	1,968.93
	10	0.57	14.41	1.53	11.03	7.17
Jittered	85	16.26	620.04	16.26	161.39	106.04
cavities	100	20.68	804.34	19.10	191.48	125.84
	1,000	719.67	32,184.86	180.63	1,920.00	1,268.00

Table 1. Timings for CavityCDT and Anglada's gift-wrapping algorithm compiled by "gcc -O3" on a MacBook Pro with a 3.06 GHz Intel dual core, 8 GB memory (1.07 GHz DDR3 SDRAM), and a 6 MB level-two cache. Each number is an average over 10,000 runs with different jittering for each run.

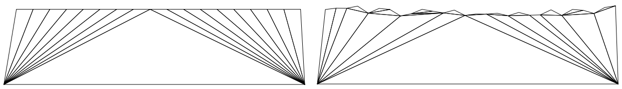

Figure 11. Constrained Delaunay triangulations of our input cavities, collinear or jittered.

cavities in which all the vertices are collinear except the segment endpoints, and perturbed versions of those cavities in which the vertices are jittered by random vertical movements proportional to the distances between the vertices. Although these examples do not represent the variety of cavities that can come up in practice, they *are* representative of the most common ways that the cavity geometry influences the running times. Collinear points, which occur frequently in practice, bring out the worst of gift-wrapping's quadratic running time. Jittered vertices hide the quadratic growth until the number of vertices is greater, because the gift-wrapping algorithm has more luck finding balanced subdivisions of small recursive subproblems.

Table 1 tabulates average running times and numbers of predicate calls for the two algorithms. Calls to Orient in Line 4 of CavityCDT are not counted because we optimized Line 4 by taking advantage of subexpressions that are reused every time it iterates and by not using exact arithmetic. We optimized Line 21 to reduce the number of Orient tests by taking advantage of the fact that if a call to InsertVertex finds that its input triangle $\triangle uvw$ has positive orientation, all its recursive calls to InsertVertex will also have positively oriented input triangles that do not require testing.

Given the cavities with collinear vertices as input, CavityCDT becomes faster than gift-wrapping for 30 vertices or more, and the advantage grows rapidly. Given the jittered cavities, CavityCDT is faster for 85 vertices or more. We observe that CavityCDT performs fewer InCircle calls on average for 8 or more vertices. In a higher-precision implementation with expensive InCircle calls, the balance would shift further in favor of CavityCDT.

Because both algorithms are easy to implement, a CDT construction program can choose between them according to the cavity size.

6 The Cost of Randomized Incremental Segment Insertion

Agarwal, Arge, and Yi [2] show that when k segments are inserted into an n-vertex CDT in random order, the total expected number of structural changes (triangles and edges created and deleted) is in $O(n \log^2 k)$. For completeness, we reprise their analysis, filling in many missing details. Then we exhibit a PSLG for which this bound is tight. Thus, if an $O(n \log k)$-time incremental segment insertion algorithm exists, it will require a smarter insertion order, not just a better analysis.

Let $\mathcal{X}$ be a PSLG with n vertices and k segments. Let S be the set of segments in $\mathcal{X}$. We use the theory of ϵ-nets to bound the expected maximum number of segments in $\mathcal{X}$ that a line segment in the plane can cross without crossing any segment in $\mathcal{X}$ that has already been inserted. Different line segments in the plane can intersect different subsets of S, but unless k is small, not all 2^k subsets are possible. The plane imposes a structure such that the number of possible subsets is polynomial in k.

Lemma 5. *Let S be a set of k non-crossing segments in the plane. Consider the sequence of segments in S whose relative interiors intersect a fixed line, written in the order of the intersection points. Let $\mathcal{Q}$ be the set of all such sequences, for all lines in the plane, with the proviso that a sequence and its reverse are considered equivalent and are not counted as distinct members of $\mathcal{Q}$. The cardinality of $\mathcal{Q}$ is at most* $8k^2 + 1$.

Proof. We use the standard planar geometric duality by which a point $p = (p_x, p_y)$ dualizes to a line p^* with the formula $y = p_x x - p_y$, and vice versa. Planar duality preserves incidences between points and lines, so if a primal point p lies on a primal line ℓ, then the dual point ℓ^* lies on the dual line p^*. Let U be the set of vertices of the segments in S; then $|U| \leq 2k$. Vertical lines do not have duals in this formulation, so assume without loss of generality that the coordinate system is rotated so that no two vertices in U have the same x-coordinate. Hence, every vertical line can be tilted slightly so that it is not precisely vertical but it intersects the same segment interiors as before.

Let $\mathcal{A}$ be the arrangement (expressed as a polygonal complex) formed by the lines dual to the vertices in U. The total number of faces of all dimensions in $\mathcal{A}$—vertices, edges, and 2-faces—is at most $2(2k)^2 + 1 = 8k^2 + 1$. If two primal lines ℓ_1 and ℓ_2 dualize to points in the same face of $\mathcal{A}$, it is possible to translate and rotate ℓ_1 to ℓ_2 without causing it to pass over a vertex in U or change its incidences with the vertices in U; it follows that ℓ_1 and ℓ_2 intersect the same segments in S in the same order. Therefore, for every sequence of segments $Q \in \mathcal{Q}$, there is a face of $\mathcal{A}$ whose interior points all dualize to lines that generate the sequence Q. It follows that the number of sequences in $\mathcal{Q}$ is less than or equal to the number of faces of $\mathcal{A}$. □

Lemma 6. *Let S be a set of k non-crossing segments in the plane. Consider the subset of segments in S whose relative interiors intersect a fixed line segment; let M be the set of all such subsets, for all line segments in the plane. The cardinality of M is at most $f(k)$, where $f(k) = k(k+1)(8k^2+1)/2 + 1$.*

Proof. The sequence of segments in S whose relative interiors intersect a line segment is an interval taken from the sequence of segments whose relative interiors intersect a line. By Lemma 5, there are at most $8k^2 + 1$ such sequences, each having length at most k. A sequence of length k has $k(k+1)/2$ nonempty intervals. We add one for the empty set. □

For our purposes, the exact bound given by Lemma 6 is not important; it suffices that the bound is polynomial, rather than exponential, in k.

Next, we reprise a famous result of Haussler and Welzl [14] on ϵ-nets. Let S_1 and S_2 be two random sequences obtained by taking i independent samples from S with replacement (hence segments may be repeated). For some $\epsilon \in (0, 1)$, let E be the event that there exists a line segment e that does not intersect any segment in S_1, but e intersects at least ϵk segments in S, where $k = |S|$. If E does not occur, S_1 is said to be an *ϵ-net* for S. We want to show that E is very unlikely when i is sufficiently large, thus S_1 is very likely to be an ϵ-net. Let E_+ be the event that there exists a line segment e that does not intersect any segment in S_1, but intersects at least ϵk segments in S and at least $\epsilon i/2$ segments in S_2. (Repeated segments in S_2 may be counted multiple times.) The following lemma shows that event E usually entails event E_+.

Lemma 7 (Haussler and Welzl [14]). *If $i \geq 8(1-\epsilon)/\epsilon$, then* $\Pr[E] \leq 2\Pr[E_+]$.

Proof. Suppose that event E occurs; then we will show that E_+ also occurs with probability at least $1/2$. Event E implies that there exists a line segment e that intersects at least ϵk of the k segments in S. Let γ be the number of segments in S_2 that e intersects; if $\gamma \geq \epsilon i/2$, then event E_+ also occurs. Because S_2 is chosen by randomly choosing i segments from S with replacement, γ is a binomial random variable with expectation ϵi and variance $\epsilon(1-\epsilon)i$. By Chebyshev's inequality, the probability that event E_+ does not also occur is

$$\Pr\left[\gamma < \frac{\epsilon i}{2}\right] \leq \Pr\left[|\gamma - \epsilon i| > \frac{\epsilon i}{2}\right] \leq 4\frac{1-\epsilon}{\epsilon i},$$

which is less than or equal to $1/2$ by assumption. Thus $\Pr[E_+] = \Pr[E]{\cdot}\Pr[E_+|E] \geq \Pr[E]/2$. ☐

Let E_* be the event that there exists a line segment e that intersects no segment in S_1, but intersects at least $\epsilon i/2$ segments in S_2.

Lemma 8 (Haussler and Welzl [14]). $\Pr[E_+] \leq \Pr[E_*] \leq f(2i)\,2^{-\epsilon i/2}$, *where* $f(k) = k(k+1)(8k^2+1)/2 + 1$.

Proof. The first inequality follows because event E_+ implies event E_*. For the second inequality, imagine sampling $2i$ segments from S with replacement to form a sequence S_{12}, then randomly choosing i of those segments to form S_1; the remainder form S_2.

Let e be a fixed line segment in the plane, and let c be the number of segments that intersect e among the $2i$ segments in S_{12}. The line segment e can trigger the event E_* only if none of those c segments are chosen for S_1, which happens with probability

$$\binom{2i-c}{i}\bigg/\binom{2i}{i} = \frac{i}{2i}\cdot\frac{i-1}{2i-1}\cdots\frac{i-c+1}{2i-c+1} \leq 2^{-c}.$$

Moreover, e can trigger E_* only if $c \geq \epsilon i/2$, so the probability of that is at most $2^{-\epsilon i/2}$. (Observe that this upper bound is independent of how many segments in S intersect e.)

Now, consider the probability that *any* line segment in the plane triggers E_*. By Lemma 6, there are at most $f(2i)$ subsets of S_{12} that a line segment e can pick out. If two line segments intersect exactly the same segments in S_{12}, then either both of them trigger E_*, or neither do. Therefore, the probability of event E_* is at most $f(2i)\,2^{-\epsilon i/2}$. ☐

Let $\Pi = \langle s_1, s_2, \ldots, s_k\rangle$ be a permutation of the k segments in $\mathcal{X}$, chosen uniformly at random from the set of all such permutations. Let $\mathcal{T}_0$ be the Delaunay triangulation of the n vertices in $\mathcal{X}$, ignoring the segments. For $i \in [0, k]$, let $\mathcal{T}_i$ be the CDT constructed by inserting the first i segments in Π.

A *conflict* is a segment-edge pair (s, e) consisting of an edge $e \in \mathcal{T}_i$ and a segment $s \in \mathcal{X}$ that crosses e. When the segment s_{i+1} is inserted into the triangulation $\mathcal{T}_i$, it deletes every edge in $\mathcal{T}_i$ it conflicts with. An edge e is said to have c conflicts if it crosses c segments in $\mathcal{X}$.

Theorem 9. *The expected number of edges deleted over the duration of the randomized incremental segment insertion algorithm is in $O(n\log^2 k)$.*

Proof. By Lemmas 7 and 8, the probability $\Pr[E]$ that there exists a line segment that intersects at least ϵk segments in $\mathcal{X}$ but intersects no segment among i segments sampled randomly from $\mathcal{X}$ with replacement satisfies $\Pr[E] \leq 2\,f(2i)\,2^{-\epsilon i/2}$. This probability is not increased by sampling without replacement, as the incremental algorithm does. Setting $\epsilon = 10\log_2 k/i$ yields $\Pr[E] \leq 2\,f(2i)/k^5 \in O(1/k)$. (Observe that this choice of ϵ satisfies the precondition of Lemma 7 for $k \geq 2$.) Therefore, the first i randomly chosen segments are likely to be a $(10\log_2 k/i)$-net for the segments in $\mathcal{X}$.

Let e be an edge with c conflicts in the triangulation $\mathcal{T}_i$. When a randomly chosen segment s_{i+1} is inserted, the probability that e is deleted is $c/(k-i)$. The probability that any edge has more than $10k\log_2 k/i$ conflicts is at most $\Pr[E]$. As $\mathcal{T}_i$ has fewer than $2n$ edges, the expected number of edges deleted over the duration of the algorithm is less than

$$2n + 2n\sum_{i=1}^{k-1}\left(\frac{10k\log_2 k}{i(k-i)} + \Pr[E]\right) \in O(n\log^2 k). \qquad ☐$$

Therefore, the total expected cost of all calls to CavityCDT is in $O(n\log^2 k)$. With the cost of constructing the initial Delaunay triangulation $\mathcal{T}_0$ and performing segment location for each segment prior to inserting it (see Section 7), the incremental CDT construction algorithm runs in expected $O(n\log n + n\log^2 k)$ time.

There is a matching $\Omega(n\log^2 k)$ lower bound on the number of structural changes. To see this, consider the PSLG in Figure 12, which is similar to an example Agarwal, Arge, and Yi [2] use to establish an $\Omega(n\log k)$ lower bound. The example uses two sequences of nearly cocircular vertices. On the bottom half of the circle is a sequence of $\Theta(n)$ *pushing vertices* that lie precisely on the circle. On the top half is a smaller sequence of m *pulling vertices* $v_1, v_2, \ldots, v_m$ that are perturbed so they lie just inside the circle. Each pulling vertex v_j is concealed by j segments, all of whose endpoints lie outside the circle. The total number of segments is $k = m(m+1)/2$. No segment conceals more than one pulling vertex. The pulling vertices are placed so that edges of the CDT connect every pushing vertex to the pulling vertex with highest index that is not concealed, which we call the *dominant* pulling vertex. Every edge connecting a pushing vertex to a pulling vertex v_j has j conflicts. Say that v_j is *alive* if no segment that conceals it has been inserted yet, and call the segments that conceal v_j its *conflicts*. When a newly inserted segment conflicts with the dominant pulling vertex, $\Theta(n)$ edges are deleted and $\Theta(n)$ new edges are created adjoining the new dominant pulling vertex.

We analyze the longevity of a pulling vertex v_c with c conflicts with a method developed by Clarkson [9]. After i segments have been inserted, the likelihood that v_c is still alive is

$$\begin{aligned}\Pr[v_c \text{ alive}] &= \binom{k-c}{i}\bigg/\binom{k}{i} = \frac{k-i}{k}\cdot\frac{k-i-1}{k-1}\cdots\frac{k-i-c+1}{k-c+1}\\ &> \left(\frac{k-i-c}{k-c}\right)^c.\end{aligned}$$

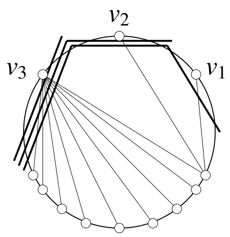

Figure 12. The pulling vertices v_1, v_2, and v_3 conflict with one, two, and three segments (thick lines), respectively. The triangulation (thin lines) connects the $\Theta(n)$ pushing vertices to v_3 until a segment conceals it.

Consider the $(i+1)$th segment insertion where

$$i \in \left[\frac{\alpha k \ln k}{m}, \min\left\{\frac{\alpha}{2} k^{1-2\alpha} \ln k, \frac{k}{2} - m\right\}\right]$$

for some constant $\alpha > 0$ we will choose shortly. For every pulling vertex v_c that has $c \leq \frac{\alpha k}{i} \ln k$ conflicts, the range of i implies that $c \leq m$ and $i \leq k/2 - c$, so

$$\begin{aligned} \Pr[v_c \text{ alive}] &> \left(\frac{k-i-c}{k-c}\right)^{\frac{\alpha k}{i} \ln k} = \left(1 + \frac{i}{k-i-c}\right)^{-\frac{\alpha k}{i} \ln k} \\ &\geq e^{-\alpha k \ln k/(k-i-c)} = k^{-\alpha k/(k-i-c)} \geq k^{-2\alpha}. \end{aligned}$$

The middle inequality follows because $1+x \leq e^x$ for all x. Consider the $\alpha k \ln k/(2i)$ vertices in the range from $v_{\alpha k \ln k/(2i)}$ to $v_{\alpha k \ln k/i}$. The probability that at least one of them is alive is

$$\begin{aligned} \Pr[\text{one of } v_{\alpha k \ln k/(2i)}, \ldots, v_{\alpha k \ln k/i} \text{ alive}] &> 1 - \left(1 - k^{-2\alpha}\right)^{\frac{\alpha k \ln k}{2i}} \\ &\geq 1 - e^{-\alpha k^{1-2\alpha} \ln k/(2i)}. \end{aligned}$$

We assume that $i \leq \frac{\alpha}{2} k^{1-2\alpha} \ln k$, so the probability is at least 0.63 that there is a live vertex with at least $\alpha k \ln k/(2i)$ conflicts. Edges connect the exposed vertex with the most conflicts to the $\Theta(n)$ pushing vertices, and the $(i+1)$th segment insertion deletes these edges with probability at least $\alpha k \ln k/(2i(k-i))$. We choose α to be a positive constant less than $1/4$. Therefore, the expected number of edges deleted during the duration of the algorithm is at least

$$\sum_{i=\frac{\alpha k \ln k}{m} \in \Theta(\sqrt{k} \ln k)}^{\min\left\{\frac{\alpha}{2} k^{1-2\alpha} \ln k, k/2 - m\right\}} 0.63 \frac{\alpha k \ln k}{2i(k-i)} \Theta(n) = \Theta(n \log^2 k).$$

The cogent observation is that there is a range of values for i spanning a factor of $k^{0.5-2\alpha}$ in which the dominant live vertex has many conflicts. Although this range is narrow (and hidden—it took us a long time to realize it existed), it suffices to allow the summation $\sum 1/i$ to contribute a $\Theta(\log k)$ factor. In this range, each doubling of i contributes $\Theta(n \log k)$ structural changes, and i is doubled $\Theta(\log k)$ times.

Our lower bound example is related to the coupon collector's problem. Imagine that the pulling vertices represent a set of m types of coupons you wish to collect, and that when you buy a coupon (choose a random segment), it is of type v_i with probability p_i. The probability of collecting every coupon $v_{i+1}, v_{i+2}, \ldots, v_m$ before collecting v_i is [1]

$$\Pr[v_i \text{ after } v_{i+1}, \ldots, v_m] = \int_0^\infty p_i e^{-p_i t} \prod_{j>i} (1 - e^{-p_j t}) \, \mathrm{d}t.$$

Our analysis is equivalent to asking, if $p_i = i/k$ where $k = \sum_{i=1}^m i$, how often can you expect to collect a coupon with a lesser index than every previous coupon? We have not been able to find a published asymptotic bound for this problem in terms of m, but we have shown that the expected answer is in $\Theta(\log^2 k) = \Theta(\log^2 m)$. By comparison, if every coupon arrives with equal probability, the expected answer is in $\Theta(\log m)$.

7 The Cost of Segment Location

Recall that for an uninserted segment s, segment location is the act of identifying a triangle in the current CDT that adjoins an endpoint of s and whose interior intersects s. We perform segment location with a simple rotary traversal of the triangles adjoining the endpoint of s with lesser degree, taking time proportional to that endpoint's degree. One way to identify the endpoint of lesser degree is to record the vertex degrees and update them with every change to the triangulation. A simpler alternative is to perform simultaneous rotary searches around both endpoints of s, interleaving steps around one endpoint with steps around the other; a suitable triangle will be found in time proportional to the lesser degree.

In practice, most segment location operations take $O(1)$ time. But what if both endpoints of many segments have large degrees? Here, we show that the total cost of this segment location method is at worst proportional to the number of vertices plus the number of structural changes that occur during segment insertion. Therefore, segment location never compromises the asymptotic running time of incremental CDT construction.

Consider incrementally constructing the CDT of a PSLG $\mathcal{X}$ with n vertices and k segments. Let l be the total number of edges that are created during all the segment insertion operations, including edges that are subsequently deleted. We have seen in Section 6 that in expectation, $l \in O(n \log^2 k)$ when the segment insertion order is randomized; but often l is much smaller.

Theorem 10. *During incremental construction of the CDT of $\mathcal{X}$ (whether randomized or not), the total time for segment location as described above is in $O(n + l)$.*

Proof. For $i \in [0, k]$, let $\mathcal{T}_i$ be the CDT after the first i segments have been inserted. For $j \in [1, n]$, let d_j be the maximum degree of the vertex with index j over all the triangulations $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k$. It follows from Euler's formula that the sum of vertex degrees in $\mathcal{T}_0$ is at most $6n - 12$, so $\sum_{j=1}^n d_j \leq 6n + 2l - 12$. The following argument shows that segment location takes $O(\sum_{j=1}^n d_j)$ time, and therefore takes $O(n + l)$ time.

Suppose that the vertices are indexed in nonincreasing order of maximum degree, so that $d_i \geq d_j$ whenever $i < j$. The time to locate a segment $s \in \mathcal{X}$ whose vertices are numbered i and j with $i < j$ is at worst proportional to $\min\{d_i, d_j\} = d_j$; call that number the *cost* of s. For all $i \in [3, n]$, let $\mathcal{X}_i \subseteq \mathcal{X}$ be the PSLG induced by taking only the first i vertices in $\mathcal{X}$ and the segments that adjoin two of those vertices.

Observe that the total time for segment location is proportional to the total cost of all the edges in $\mathcal{X} = \mathcal{X}_n$. We show that this total is less than $3 \sum_{i=1}^n d_i$. As $\mathcal{X}$ is planar, $\mathcal{X}_i$ has at most $3i - 6$ edges as a consequence of Euler's formula. Suppose without loss of generality that $\mathcal{X}_n$ has exactly $3n - 6$ edges, the maximum possible. As $\mathcal{X}_{n-1}$ has at most $3n - 9$ edges, at least three of the edges in $\mathcal{X}_n$ adjoin vertex n and have a cost no greater than d_n. Likewise, $\mathcal{X}_{n-2}$ has at most $3n - 12$ edges, so at least six of the edges in $\mathcal{X}_n$ (including the three just discussed) adjoin vertex $n-1$ or vertex n and have a cost no greater than d_{n-1}. In general, for $i \in [1, n-3]$, at least $3i$ edges in $\mathcal{X}_n$ have a cost no greater than d_{n+1-i}. Putting these costs together, we can identify $3n - 9$ edges whose total cost is at most $3 \sum_{i=4}^n d_i$. The total cost of the remaining three edges is at most $d_2 + 2d_3$, so the total cost of all the edges is less than $3 \sum_{i=1}^n d_i$. □

8 Conclusions and Open Problems

Although this paper emphasizes the complexity of randomized incremental segment insertion, we draw some conclusions useful to programmers implementing CDT construction codes. First, when some segments cross very large numbers of edges, a faster segment insertion algorithm can make enough of a difference to justify its implementation. Second, there is a very simple segment location method that never compromises incremental insertion's asymptotic running time, with or without randomization. Third, it might be worthwhile to randomize the order in which the segments are inserted; compare the expected $O(n \log^2 k)$ upper bound on the number of structural changes with the $\Theta(nk)$ structural changes that can occur with a deterministic ordering. Fourth, the $O(n \log^2 k)$ upper bound is almost always too pessimistic in practice; circumstances in which this much work is required are difficult to devise and unlikely to occur in the real world. Fifth, and most importantly, if incremental segment insertion is implemented intelligently, it is fast enough; implementing a more complicated $O(n \log n)$-time CDT construction algorithm is unlikely to repay the effort.

Given the $\Omega(n \log^2 k)$ lower bound on the expected number of structural changes that the randomized incremental segment insertion algorithm performs on some PSLGs, it is natural to wonder whether it can be made faster by biasing the order in which the segments are inserted or otherwise changing the algorithm. For example, it is possible to choose two random segments, decide which one crosses the fewest edges, and insert it in time proportional to the number of edges it crosses—without taking the time to count all the edges the other segment crosses. Does this procedure reduce the expected asymptotic number of structural changes?

Interestingly, the $O(n \log^2 k)$ upper bound on the number of structural changes does not rely on the constrained Delaunay property; the analysis applies however the cavities are retriangulated. Are there other algorithms for computing optimal triangulations that can be sped up by randomized incremental segment insertion?

Acknowledgments

We thank Kevin Yi for introducing us to his work on CDT structural changes [2], Pankaj Agarwal and Ken Clarkson for discussions of the probabilistic methods behind it [9, 14], Peter Shor for pointing out the connection to the coupon collector's problem, and James Martin for introducing us to the Poissonization of that problem. We also thank James O'Brien for the inspiration of writing "This is hopeless. You will never figure it out" on the whiteboard while we worked out the lower bound.

References

[1] Ilan Adler, Shmuel Oren, and Sheldon M. Ross. *The Coupon-Collector's Problem Revisited*. Journal of Applied Probability **40**(2):513–518, June 2003.

[2] Pankaj K. Agarwal, Lars Arge, and Ke Yi. *I/O-Efficient Construction of Constrained Delaunay Triangulations*. Unpublished manuscript, 2005. Most of this paper appears in the Proceedings of the Thirteenth European Symposium on Algorithms, pages 355–366, October 2005, but the published version omits the analysis of the number of structural changes performed by randomized incremental segment insertion.

[3] Marc Vigo Anglada. *An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations*. Computers and Graphics **21**(2):215–223, March 1997.

[4] Adrian Bowyer. *Computing Dirichlet Tessellations*. The Computer Journal **24**(2):162–166, 1981.

[5] Kevin Q. Brown. *Voronoi Diagrams from Convex Hulls*. Information Processing Letters **9**(5):223–228, December 1979.

[6] Siu-Wing Cheng, Tamal Krishna Dey, and Jonathan Richard Shewchuk. *Delaunay Mesh Generation*. CRC Press, Boca Raton, Florida, December 2012.

[7] L. Paul Chew. *Constrained Delaunay Triangulations*. Algorithmica **4**(1):97–108, 1989.

[8] ———. *Building Voronoi Diagrams for Convex Polygons in Linear Expected Time*. Technical Report PCS-TR90-147, Department of Mathematics and Computer Science, Dartmouth College, 1990.

[9] Kenneth L. Clarkson. *New Applications of Random Sampling in Computational Geometry*. Discrete & Computational Geometry **2**(1):195–222, December 1987.

[10] Boris Nikolaevich Delaunay. *Sur la Sphère Vide*. Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk **7**:793–800, 1934.

[11] Herbert Edelsbrunner and Raimund Seidel. *Voronoi Diagrams and Arrangements*. Discrete & Computational Geometry **1**:25–44, 1986.

[12] Steven Fortune. *A Sweepline Algorithm for Voronoi Diagrams*. Algorithmica **2**(2):153–174, 1987.

[13] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams*. ACM Transactions on Graphics **4**(2):74–123, April 1985.

[14] David Haussler and Emo Welzl. *ϵ-Nets and Simplex Range Queries*. Discrete & Computational Geometry **2**(1):127–151, December 1987.

[15] François Hermeline. *Triangulation Automatique d'un Polyèdre en Dimension N*. RAIRO Analyse Numérique **16**(3):211–242, 1982.

[16] Rolf Klein and Andrzej Lingas. *A Note on Generalizations of Chew's Algorithm for the Voronoi Diagram of a Convex Polygon*. Proceedings of the Fifth Canadian Conference on Computational Geometry, pages 370–374, August 1993.

[17] Der-Tsai Lee and Arthur K. Lin. *Generalized Delaunay Triangulations for Planar Graphs*. Discrete & Computational Geometry **1**:201–217, 1986.

[18] Der-Tsai Lee and Bruce J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation*. International Journal of Computer and Information Sciences **9**(3):219–242, 1980.

[19] Raimund Seidel. *Voronoi Diagrams in Higher Dimensions*. Diplomarbeit, Institut für Informationsverarbeitung, Technische Universität Graz, 1982.

[20] ———. *Constrained Delaunay Triangulations and Voronoi Diagrams with Obstacles*. 1978–1988 Ten Years IIG (H. S. Poingratz and W. Schinnerl, editors), pages 178–191. Institut für Informationsverarbeitung, Technische Universität Graz, 1988.

[21] ———. *Backwards Analysis of Randomized Geometric Algorithms*. New Trends in Discrete and Computational Geometry (János Pach, editor), Algorithms and Combinatorics, volume 10, pages 37–67. Springer-Verlag, Berlin, 1993.

[22] Michael Ian Shamos and Dan Hoey. *Closest-Point Problems*. 16th Annual Symposium on Foundations of Computer Science (Berkeley, California), pages 151–162, October 1975.

[23] Jonathan Richard Shewchuk. *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*. Discrete & Computational Geometry **18**(3):305–363, October 1997.

[24] David F. Watson. *Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes*. The Computer Journal **24**(2):167–172, 1981.