CMSC 754: Lecture 12
Delaunay Triangulations: Incremental Construction

Reading: Chapter 9 in the 4M's. This algorithm was presented in L. J. Guibas, D. E. Knuth, and M. Sharir, *Randomized incremental construction of Delaunay and Voronoi diagrams*, Algorithmica, 7, 1992, 381–413.

Constructing the Delaunay Triangulation: We will present a simple randomized incremental algorithm for constructing the Delaunay triangulation of a set of n sites in the plane. Its expected running time is $O(n \log n)$ (worst-case over all point sets, but expected over random insertion orders). This simple algorithm had been known for many years as a practical solution, but theoreticians dismissed it as inefficient because its worst-case running time is $O(n^2)$. When the randomized analysis was developed, the algorithm was viewed much more positively.

The algorithm is remarkably similar in spirit to the randomized algorithm for the trapezoidal maps: it not only builds the triangulation but also provides a point-location data structure for the final triangulation. (Rather than building a point-location data structure, we will adopt an alternative method based on *bucketing* future sites.)

The input consists of a set $P = \{p_1, \ldots, p_n\}$ of point sites in $\mathbb{R}^2$. As with any randomized incremental algorithm, the idea is to insert sites in random order, one at a time, and update the triangulation with each new addition. The analysis will show that, after each insertion, the expected number of structural changes in the diagram is $O(1)$.

As with the incremental algorithm for trapezoidal maps, we need some way of keeping track of where newly inserted sites are to be placed in the diagram. We will store each of the uninserted sites in a *bucket* according to the triangle in the current triangulation that contains it. We will show that the expected number of times that a site is rebucketed throughout the course of the algorithm is $O(\log n)$, which, when summed over all the sites, leads to a total time of $O(n \log n)$.

Incremental update: It will be convenient to assume that each newly added site lies within some triangle of the triangulation. This will not be true when sites are added that lie outside the convex hull of the current point set. To address this, we start by adding three bogus *sentinel sites* that form an infinitely large triangle containing all the sites. After the final triangulation is completed, we will remove these sentinel sites and their incident triangles. (In our trapezoidal map algorithm, this is analogous to putting all the segments in an enclosing rectangle.)[1] We won't show this triangle in our figures, but imagine that it is there nonetheless.

We permute the sites in random order and insert one by one. When a new site p is added, we find the triangle $\triangle abc$ of the current triangulation that contains this site (we will see how later), insert the site into this triangle, and join the site to the three surrounding vertices (see Fig. 1(a)). This creates three new triangles incident to p, $\triangle pab$, $\triangle pbc$, and $\triangle pca$. For each triangle (say, $\triangle pab$), we check whether the vertex of the triangle that lies on the *opposite side*

[1] Some care must be taken in the construction of this enclosing triangle. It is not sufficient that it simply contains all the sites. It should be so large that the vertices of the triangle do not lie in the circumcircles of any of the triangles of the final triangulation. Our book suggests a symbolic alternative, which is more reliable.

of the edge ab lies within the circumcircle of $\triangle pab$. This is called a *local Delaunay condition*. (If there is no such vertex, because this edge is on the convex hull, then we are done.) If this vertex, call it d, fails the local Delaunay condition, we swap the edge ab out and replace it with pd. We repeat the same test process recursively with these triangles (see Fig. 1(b)).

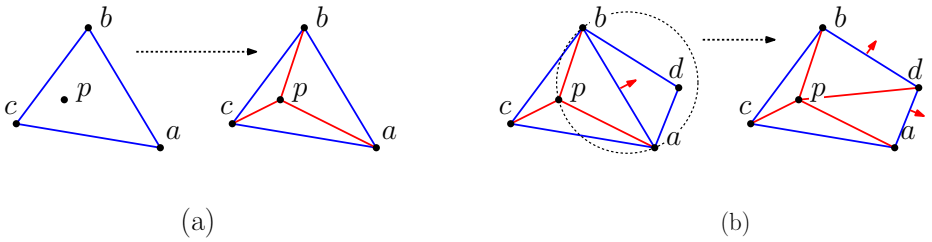


Fig. 1: Delaunay site insertion.

Local vs. Global Delaunay: An obvious issue with the above process is that we test only sites adjacent to the triangle for circumcircle violations, not all sites. This works because of an important property of Delaunay triangulations. The empty circumcircle condition implies that, in a Delaunay triangulation, the circumcircle of every triangle contains no other sites. This suggests two different criteria for testing whether a triangulation is Delaunay:

Global Delaunay: The circumcircle of each triangle $\triangle abc$ contains no other site d. (Fig. 2(a) shows a violation.)

Local Delaunay: For each pair of neighboring triangles $\triangle abc$ and $\triangle acd$, d lies outside the circumcircle of $\triangle abc$. (Fig. 2(b) shows a violation.)

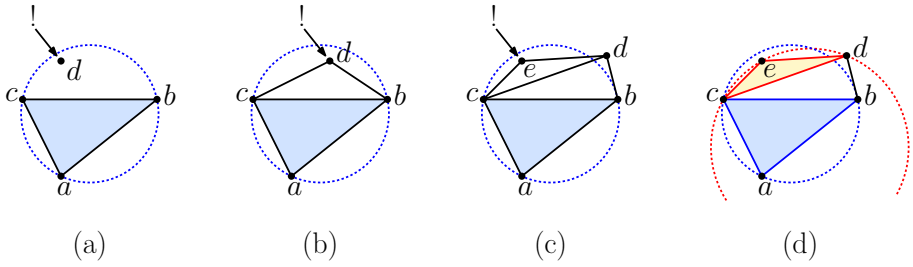


Fig. 2: Global- and local-Delaunay conditions.

Clearly, if a triangulation is globally Delaunay, it is locally Delaunay. Our incremental algorithm only checks the local-Delaunay condition, however. Could it be that a triangulation might satisfy the condition locally, but fail to satisfy it globally (see Fig. 2(c))? Delaunay proved, however, that the two conditions are in fact equivalent:

Delaunay's Theorem: A triangulation is globally Delaunay if and only if it is locally Delaunay.

Proof: (Sketch) The global to local implication is trivial, so it suffices to prove that local implies global. Consider any triangle $\triangle abc$ in a locally Delaunay triangulation, and let d be the remaining vertex of a neighboring triangle on the opposite side of edge bc. We

assert that if d lies outside the circumcircle of $\triangle abc$, then no other site can lie within this circumcircle.

A formal justification would take too much work, so we'll consider a limited scenario that illustrates the key idea. Suppose d is outside the circumcircle of $\triangle abc$ (the blue circle in Fig. 2(d)), but (contrary to the claim) the vertex e opposite the edge cd lies within this circumcircle (see Fig. 2(d)). Consider the circumcircle of $\triangle cde$ (the red circle in Fig. 2(d)). By an elementary (but somewhat tedious) analysis of the configuration of these sites, it follows that b lies within this circumcircle. Since b is in the neighboring triangle to $\triangle cde$, this implies that the triangulation is *not* locally Delaunay, which yields the contradiction.

Because the algorithm checks that all newly created triangles are locally Delaunay, its correctness follows directly.

Incircle Test: Before presenting the algorithm, we shall introduce the geometric primitives involved in testing whether triangles satisfy the Delaunay condition. Recall that a triangle $\triangle abc$ is in the Delaunay triangulation if and only if the circumcircle of this triangle contains no other site in its interior. (Recall that we make the general position assumption that no four sites are cocircular.)

How do we test whether a site d lies within the interior of the circumcircle of $\triangle abc$? Assume the vertices of $\triangle abc$ are given in counterclockwise order. We claim that d lies in the circumcircle determined by $\triangle abc$ if and only if the following determinant is positive (see Fig. 3(a)).

$$\text{inCircle}(a,b,c,d) \;\equiv\; \det \begin{pmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{pmatrix} \;>\; 0.$$

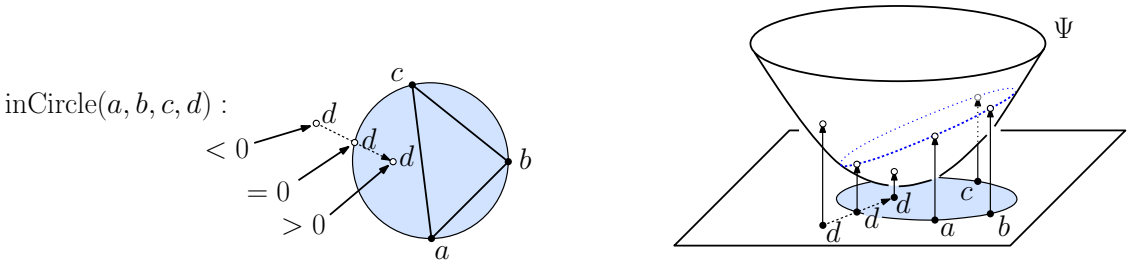


Fig. 3: Incircle test.

This is called the *incircle test* in $\mathbb{R}^2$. (The incircle test can be generalized to any dimension.) Notably, the incircle test in $\mathbb{R}^2$ can be viewed as an orientation test in 3-D, where we lift the sites onto a paraboloid Ψ in $\mathbb{R}^3$ by adding a z-coordinate equal to x^2+y^2. (As d moves from within the circle to outside, the lifted point $d^\uparrow$ moves from below to above the hyperplane through the lifted points $a^\uparrow$, $b^\uparrow$, and $c^\uparrow$ (see Fig. 3(b)).

It is also noteworthy that, while we have defined the incircle test as testing d with respect to $\triangle abc$, the basic laws of determinants imply that (subject to a possible sign change) this can be used to test any of the four sites with respect to the triangle defined by the other three.

Deriving the Incircle Test (Optional): We will not prove the correctness of this test, but we will show a somewhat simpler assertion, namely that if the four points are cocircular then the above determinant is equal to zero. (It follows from continuity that as d moves from inside the circle to the outside, the sign of the determinant changes as well.)

Suppose that a, b, c, and d are all cocircular then there exists a center point $q = (q_x, q_y)$ and a radius r such that

$$(a_x - q_x)^2 + (a_y - q_y)^2 = r^2,$$

and similarly for the other three points. (We won't compute q and r, but merely assume their existence for now.) Expanding this and collecting common terms, we have

$$\begin{aligned} 0 &= (a_x^2 + a_y^2) - 2q_x a_x - 2q_y a_y + (q_x^2 + q_y^2 - r^2) \\ &= (-2q_x)a_x + (-2q_y)a_y + 1 \cdot (a_x^2 + a_y^2) + (q_x^2 + q_y^2 - r^2) \cdot 1. \end{aligned}$$

If we do the same for the other three points, b, c, and d, and express this in the form of a matrix, we have

$$\begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} \begin{pmatrix} -2q_x \\ -2q_y \\ 1 \\ q_x^2 + q_y^2 - r^2 \end{pmatrix} = 0.$$

In other words, a linear combination of the columns of the 4×4 matrix equals the zero vector. We know from linear algebra that this is true if and only if the determinant of the matrix is zero.

Randomized Incremental Construction: We can now present the complete algorithm. Given the set $P = \{p_1, \ldots, p_n\}$ of sites, we first compute the sentinel triangle containing them all. We then permute the sites randomly and insert them into the triangulation one by one.

The algorithm for the incremental algorithm is shown in the code block below, and an example is presented in Fig. 4. We store the current triangulation in a doubly-connected edge list (DCEL). All of the basic operations described below (finding edges and vertices and flipping edges) can be done in $O(1)$ time.

As you can see, the algorithm is very simple. The only element that has not been shown is the implementation. First, we update the data structure for the simplicial complex. These updates can be done in $O(1)$ time each on any reasonable representation (a DCEL, for example). The other issue is locating the triangle that contains p. We will discuss this below.

Running-Time Analysis: To analyze the algorithm's expected running time, we establish two bounds, each averaged over all possible insertion orders. With the addition of each site:

(1) $O(1)$ structural changes are made to the triangulation (in expectation), and

(2) $O(\log n)$ time is spent determining which triangle contains each newly inserted site (in expectation).

Incremental Delaunay Triangulation Algorithm

```
Insert(p) {
    Find the triangle △abc containing p
    Insert edges pa, pb, and pc into triangulation
    SwapTest(ab)                          // check/fix the surrounding edges
    SwapTest(bc)
    SwapTest(ca)
}

SwapTest(ab) {
    if (ab is an edge on the exterior face) return
    Let d be the vertex to the right of edge ab
    if (inCircle(p, a, b, d)) {           // d violates the incircle test
        Flip edge ab                      // replace ab with pd
        SwapTest(ad)                      // check/fix the new suspect edges
        SwapTest(db)
    }
}
```

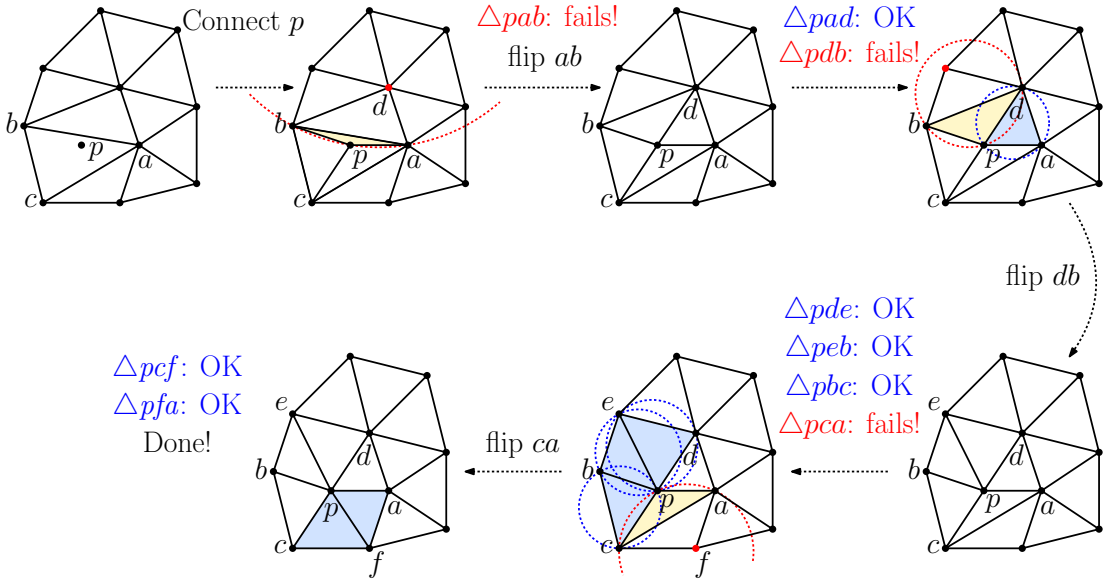


Fig. 4: Incremental site insertion.

These bounds depend *only* on the insertion order, not the distribution of the sites.

Bounding the Structural Changes: We argue first that the expected number of edge changes with each insertion is $O(1)$ by a simple application of backwards analysis. First, observe that (assuming general position) the structure of the Delaunay triangulation is independent of the insertion order of the sites so far. Thus, any of the existing sites is equally likely to have been the last site to be added to the structure.

Suppose that some site p was the last to have been added. How much work was needed to insert p? Note that the initial insertion of p created three new edges, all incident to p. Also, whenever an edge swap is performed, the algorithm adds a new edge incident to p. These are the only changes that the insertion algorithm can make. Therefore, the total number of changes to the triangulation when inserting p is proportional to p's *degree* after insertion (see Fig. 5). Although any one vertex may have a very high degree, we will exploit the fact that in a planar graph, the average vertex degree is just a constant.

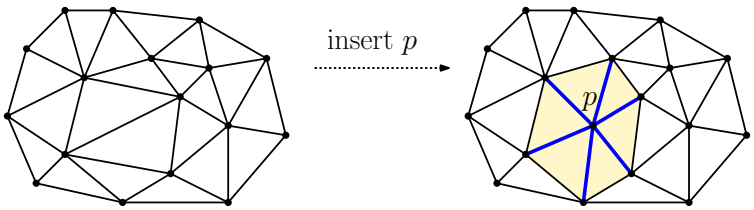


Fig. 5: The number of structural changes equals p's degree after insertion (three initial edges and three edge flips).

To perform the backwards analysis, we consider the situation after inserting the ith site. Let d_i be a random variable that indicates the degree of the newly inserted site in our randomized algorithm. Let $P_i = \{p_1, \ldots, p_i\}$ denote the first i sites to be inserted. Although the diagram depends on which particular i sites are in this subset, our analysis will not. For $1 \leq j \leq i$, let $\deg(p_j)$ denote the degree of site p_j in triangulation $DT(P_i)$ just after the ith insertion.

Because the diagram does not depend on the insertion order, each of the sites of P_i has an equal probability of $1/i$ of being the last site to be inserted. Recall that (by Euler's formula), the triangulation has at most $3i$ edges. The sum of vertex degrees equals twice the total number of edges (since each edge is counted twice), which is $6i$. We conclude that expected value of d_i, denoted $E[d_i]$, satisfies:

$$E[d_i] \;=\; \frac{1}{i}\sum_{j=1}^{i} \deg(p_i) \;\leq\; \frac{6i}{i} \;=\; 6.$$

Therefore, by the magic of backwards analysis, the expected number of structural changes after inserting the ith site is 6.

Bounding the Location Cost: The second aspect of the expected-case running time is the cost of determining which triangle contains each newly created site. As mentioned earlier, we will employ a bucketing approach, as we did with the trapezoidal-map algorithm. Think of each triangle of the current triangulation as a *bucket* that holds the sites that lie within this

triangle and have yet to be inserted (see Fig. 6(a)). When a new site p is inserted, a number of old triangles are deleted (shaded red in Fig. 6(a)) and a number of new triangles are created (shaded blue in Fig. 6(b)). All sites in the old triangles' buckets must be moved into the associated new triangle. This process is called *rebucketing*.

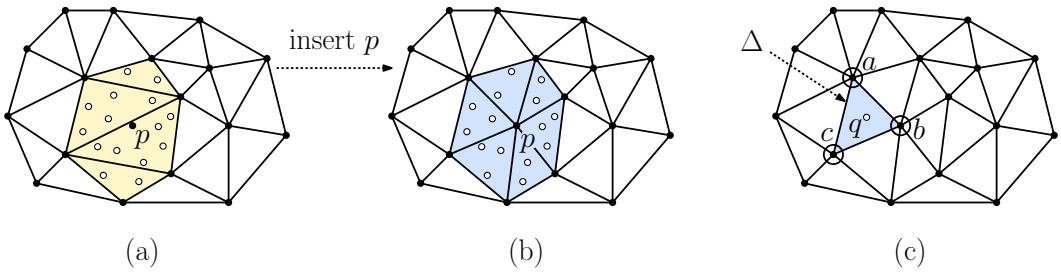


Fig. 6: Rebucketing sites after inserting site p.

To simplify the analysis, assume the cost of rebucketing a single site during a single insertion is $O(1)$.[2] We will show through a backwards analysis that, in expectation, any fixed site is rebucketed $O(\log n)$ times.

Let $q \in P$ be a fixed site. Consider the situation just after inserting the ith site. We assume q has not yet been inserted; otherwise, its rebucketing cost is zero after the ith insertion. For $1 \leq i \leq n$, let $X_i(q)$ denote the random event that q is moved to a new triangle after the ith insertion, and let $\text{Prob}(X_i(q))$ denote the probability of this event. Letting $B(q)$ denote the average number of times that q is rebucketed throughout the algorithm, we have

$$B(q) \;\leq\; \sum_{i=1}^{n} \text{Prob}(X_i(q)).$$

To bound $\text{Prob}(X_i(q))$, let Δ be the triangle containing q after the ith insertion. As observed above, after inserting the ith site, all newly created triangles are incident to this new site. Thus, Δ would have come into existence as a result of the last insertion if and only if one of its three incident vertices happened to be the last to be inserted (see Fig. 6(c)). Since Δ is incident to exactly three sites, and each site is equally likely to be the last inserted, the probability that Δ came into existence is $3/i$. (We are cheating a bit here by ignoring the three initial sites at infinity.) Therefore, $\text{Prob}(X_i(q)) \leq 3/i$.

From this, it follows that the expected number of times that the site q is rebucketed is

$$B(q) \;\leq\; \sum_{i=1}^{n} \frac{3}{i} \;=\; 3\sum_{i=1}^{n} \frac{1}{i}.$$

Recall that $\sum_{i=1}^{n} 1/i$ is the Harmonic series, and for large n, its value is very nearly $\ln n$. Thus we have

$$B(q) \;\leq\; 3 \cdot \ln n \;=\; O(\log n).$$

[2]The issue is that rebucketing cost depends on the degree of the newly inserted site. In the previous section we showed that the average degree is a constant, so this assumption is not unreasonable. Nonetheless, a complete analysis would need to account for this additional cost.

Although the diagram depends on the order in which the sites have been added, this bound does not. Summing over all the n sites, it follows that the total time spent rebucketing all the sites is $\sum_{i=1}^{n} B(p_i) = O(n \log n)$.

Point Location Data Structure: The above analysis resembles the analysis we performed for point location in the incremental algorithm for trapezoidal maps. As we did there, instead of simply rebucketing points, we could build a directed acyclic graph (DAG) that records the history of rebucketing operations on a triangle-by-triangle basis. The result is a data structure that can answer point-location queries. The query time is the same as that of the trapezoidal map (modulo constant factors).