CS 6301.002.20S Lecture 13—February 25, 2020

Main topics are #Voronoi-diagrams, and #Delaunay_triangulations.

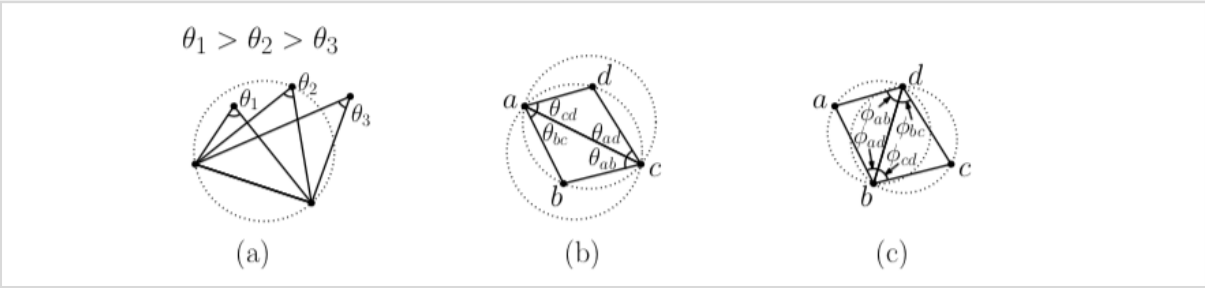
Delaunay Triangulations

- Last time, we considered the planar dual of a Voronoi diagram over sites P = {p_1, p_2, …, p_n}.
- Assuming general position, each face of this dual is a triangle with sites as its vertices. The planar subdivision is called a Delaunay triangulation.

Maximizing Angles

- Earlier, I mentioned how the Delaunay triangulation is a particularly good choice for making a triangulated irregular network, a triangulation over a set of points sampled over a terrain. We want a good triangulation so we can extrapolate the height of any point based on the heights of its triangle's vertices.
- Suppose we want to build a TIN over P. One thing we'd like to avoid is very skinny triangles. In particular, a pair of skinning triangles sharing a long edge means you're determining the height of a point on that long edge based entirely on a pair of very far away sites.
- It turns out the Delaunay triangulation maximizes the smallest angle over all the triangles. Even stronger, among all triangulations maximizing the smallest angle, it maximizes the second smallest. Even stronger, among all triangulations maximizing the smallest and then the second smallest angle, it maximizes the third, and so on.
- Let's make this claim more formal. Any triangulation is associated with an *angle sequence* <alpha_1, alpha_2, …, alpha_m> which is the set of triangle angles sorted in increasing order.
- If we were to compare two angle sequences lexicographically, we would compare their first angles. In the event of a tie, we would compare their second angles, and so on.
- Theorem: Among all triangulations of P, the Delaunay triangulation has the lexicographically largest angle sequence.
- Recall, two sites p_i and p_j are connected by a Delaunay edge if and only if there is an empty circle passing through them.
- The book proves that given any triangulation that doesn't have the empty circle property, there exists some convex quadrilateral abcd where the long diagonal ac is in the triangulation.
- We can replace this long diagonal with the short one bd. This operation is called an *edge flip*.

- We can argue that the smallest new angle is not smaller than the smallest old angle. Details appear in the book, Mount, and my notes.
- We can repeat this process as long as there is some non-empty circle. Since there are a finite number of triangulations, the process will terminate.
- And it will terminate with the Delaunay triangulation, meaning it has the lexicographically largest angle sequence.
- The proof uses an important geometric fact:
- Consider a circle through two points. Let theta_1 be the angle formed with a middle point inside the circle, theta_2 be the angle formed with a third point on the circle, and theta_3 be the angle formed with a third point outside the circle. Then
  - theta_1 > theta_2 > theta_3
- Before the flip, the circumcenters for both triangles contains the other point on the quad.



- After the flip, the two circumcircles do not contain the forth point.
- But take, for example, this angle labeled theta_ab. c lies on the boundary of a circle through those points, but d lies interior. So this angle labeled phi_ab is bigger. Similarly, phi_bc > theta_bc, phi_cd > theta_cd, and phi_da > theta_da.
- You can also argue that the other two new angles are not smaller than the smallest phi angle.

Constructing Delaunay Triangulations

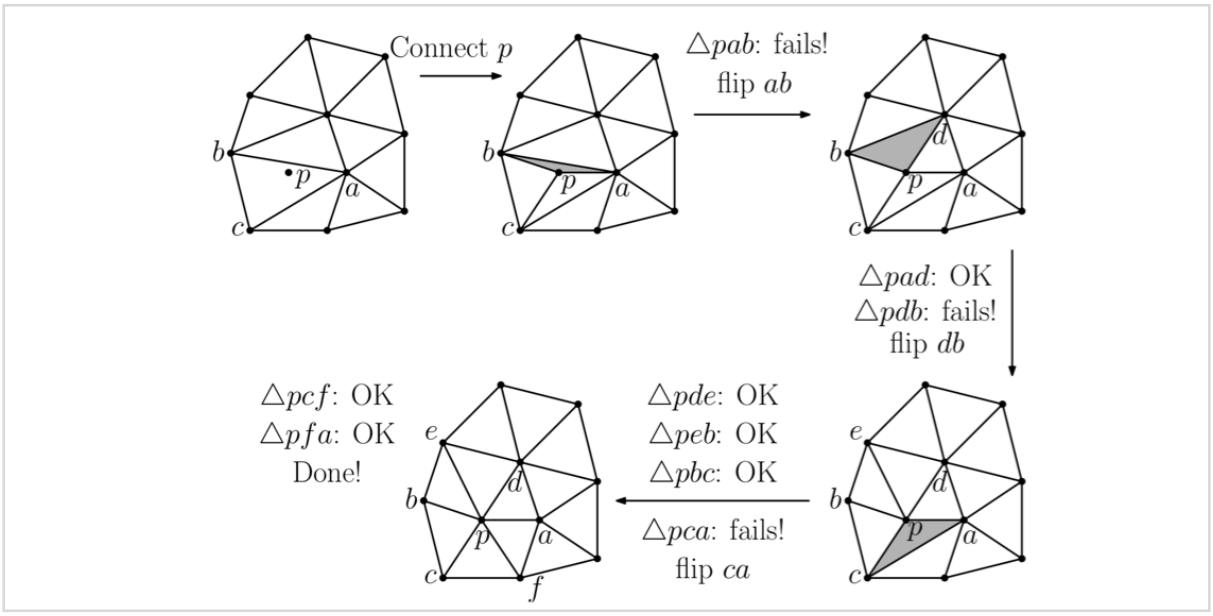
- The definition I gave for the Delaunay triangulation was to take a Voronoi diagram and connect pairs of sites if they have neighboring Voronoi cells.
- Last week, I described an O(n log n) time plane sweep algorithm to construct a Voronoi diagram. We could just run that algorithm and then construct the Delaunay triangulation.
- However, there is a more practical randomized incremental construction algorithm for building the Delaunay triangulation directly. It runs in O(n^2) time in the worst case, but O(n log n) time in expectation.

Adding a Site

- Like usual, I'll assume we already have some sort of solution before even adding the first site. To do that, let's add three points that form a really really big triangle around the input sites.

- It must be so big that when we remove it later, we only destroy triangles that lie strictly outside the Delaunay triangulation of the input sites.
- The book describes a way to do this symbolically so that you can be sure the triangle is large enough.
- The incremental construction algorithm uses two basic operations that change the tentative triangulation.
  - Joining a site inside a triangle to the triangle's vertices. This will be the way we initially add sites.
  - Performing an *edge flip* like we saw in the angle sequence proof. Edge flips fix bad parts of the triangulation.

- Since it's an incremental construction algorithm, we'll add the sites one-by-one.
- Suppose we want to add a new site p inside triangle abc. I'll show later how to find this triangle quickly.

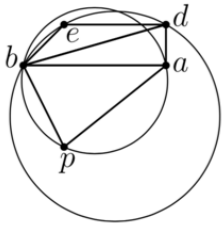


- We add site p by connecting it to each of a, b, and c. But is our new triangulation Delaunay?
- Remember, the triangulation is a Delaunay triangulation if and only if the circumcircle around each triangle is empty.
- Every triangle that doesn't contain p was present before we added p. We'll just check triangles that do contain p.
- Each of those triangles has one edge opposite p. I'll argue later that it suffices to check if the triangle contains the vertex on the other side of that edge.
- The algorithm checks the empty circle condition for each triangle incident to p. For each that doesn't meet the empty circle condition, we do an edge flip.

- But this creates more triangles incident to p! We recursively check the new triangles as well.
- Here's the code:
  - Insert(p):
    - find triangle abc containing p.
    - insert edges pa, pb, and pc.
    - SwapTest(ab)
    - SwapTest(bc)
    - SwapTest(ca)
  - SwapTest(ab):
    - if ab is on external face, return
    - let d be the vertex in ab's other triangle
    - if d is in circumcircle for p, a, and b:
      - flip edge ab for pd
      - SwapTest(ad)
      - SwapTest(bd)
- So the algorithm is pretty simple, but is it correct? And how long do all these recursive empty circle tests and edge flips take?

Correctness

- The only big question when it comes to correctness is whether just checking the three sites near a triangle suffices to satisfy the empty circle test for all triangles.
- We say a triangulation is *locally Delaunay* if for each triangle abc, the vertices lying opposite on the three neighboring triangles satisfy the empty circle property for abc.
- A triangulation is *globally Delaunay* if the empty circle property holds for every triangle and every site of P.
- *Delaunay's Theorem* states that locally Delaunay implies globally Delaunay.
- Let's show something a bit easier and more closely related to our algorithm. Say we have newly added triangle pab with d opposite edge ab. If d is outside the circumcircle for pab, then no other point is inside that circle.
- So let's say d is outside that circle, but there is another site e inside the circle. Just to make this easy, suppose e is on the other triangle of bd.
- You can argue then that this bigger circumcircle of triangle bde contains a.



if e violates the circumcircle condition for $\triangle pab$
then a violates the condition with respect to $\triangle bde$.

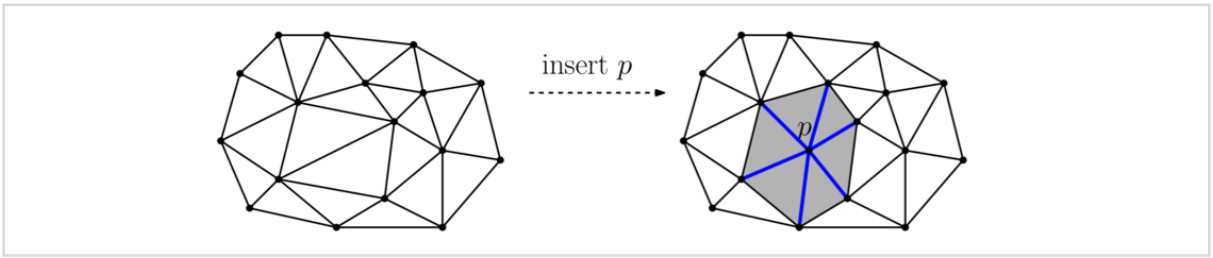
- But triangle bde existed before we even added p, and inductively we can assume we started with a Delaunay triangulation before we added p.

Point Location

- But there's one big thing I haven't addressed yet. How do we figure out which triangle contains p?
- The book suggests incrementally building a point location data structure like we did for trapezoidal maps. But there's another way.
- We'll store the sites we have yet to insert in a collection of buckets, one per triangle. Each bucket contains the sites for its triangle.
- So when we add a site, we just check which bucket it belongs to.
- Whenever an edge is flipped or we split a triangle into three through the insertion of a new site, we end up destroying an old triangle and creating some new ones.
- We'll just take all the points from destroyed triangle's buckets and redistribute them into buckets for the new triangles. This takes O(1) time per site we rebucket.
- So for the analysis, we need to determine
  - how many new triangles are created in expectation with each newly inserted site, and
  - how many times each site gets rebucketed

Making Triangles

- So suppose we've added i sites, and fix which ones they are. The Delaunay triangle for these i sites is the same no matter which order they were added.
- Let p be the last point we added. Point p is random.
- Well, when we added p, we added three new triangles that contained p. Then we performed a bunch of edge flips.



- But each time we did a flip, we added one more edge to p.
- Therefore, the time spent adding triangles is proportional to the degree of p after the insertion is complete.
- Ah, but we know there are at most 3i edges in the triangulation for those i sites.
- Therefore, there are 6i edge endpoints. There average site has degree 6.
- Since any one of the i sites is equally likely to be the last one added, the expected degree of p is therefore $\leq 6 = O(1)$.

- Summing over all n insertions, the expected total number of triangles added is O(n). Each one can be added in O(1) time.

Rebucketing

- So when we remove a triangle and add new ones, we have to rebucket its points. Rebucketing takes O(1) time per point being moved.
- It turns out each point needs rebucketed during O(log n) of the insertions in expectation.
- Consider a site q that still needs inserting after i insertions.
- What is the probability that the last insertion required a rebucketing of q? It is equal to the probability that q's triangle changed during the last insertion.
- But we only add a triangle if one of its incident sites was added. There are three sites, each of which is equally likely to be added, so q got rebucketed with probability at most 3/i.
- How many insertions rebucketed q in expectation? We'll be lazy and assume q was in a bucket during every insertion. sum_{i = 1}^n 3 / i = 3 sum_{i=1}^n 1/i = 3H_n = O(log n).
- So it seems like we'll spend O(n log n) time rebucketing in expectation.
- Alright, but this isn't really a perfect analysis of the algorithm. The problem is that I'm ignoring how you sometimes need to rebucket a site multiple times during a single insertion.
- Intuitively, each insertion causes O(1) triangle changes, so we really only need to multiply by a constant. But the real analysis is more subtle. Check out your book if you're interested.
- Overall, we end up spending O(n log n) time in expectation rebucketing sites.