CS 6301.008.18S Lecture—February 20, 2017

Main topics are #Delaunay_triangulations, #randomized_incremental_construction.

Prelude

- I've put up a page about the course projects on the class website.
- Groups of up to three can participate in a project where you try to solve a theoretical problem, experiment with some different algorithms for a problem, or write a survey describing some problem you're interested in.
- My hope is that you'll work on questions you want to know the answer to and whose answers will be useful in future careers; whether you're prepping for industry or further research.
- Each individual should write a 1 to 2 page project proposal describing what you want to do, what is known or at least what you know is known.
- I want to post proposals to the website so you can look at each others' for inspiration and find your groups.
- And I'd like the proposals turned into eLearning as pdfs by March 6th. That's two weeks from today.
- I think I'll give two more regular homeworks after that, but I'll make them lighter so you can work on your projects in parallel.
- And at the end of the semester each group should write an about 10 page paper and do a 20 minute presentation describing their progress or what they learned.
- Again, please see the website for details, and feel free to talk with me if you're looking for project ideas.

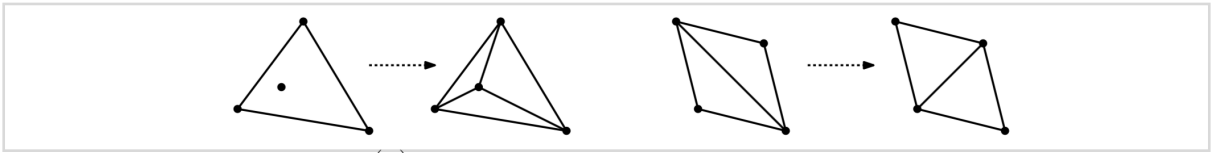
Constructing Delaunay Triangulations

- Last Thursday, I introduced the Delaunay triangulation. A way to connect n *sites* P = {p_1, …, p_n} via edges to create a number of triangles so that the resulting graph has some nice properties.
- Today, we're going to discuss constructing one.
- 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.
- On Tuesday, 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 simple incremental construction algorithm for building the Delaunay triangulation directly. This solution is much more practical, but adding vertices in the wrong order leads to an O(n^2) running time.

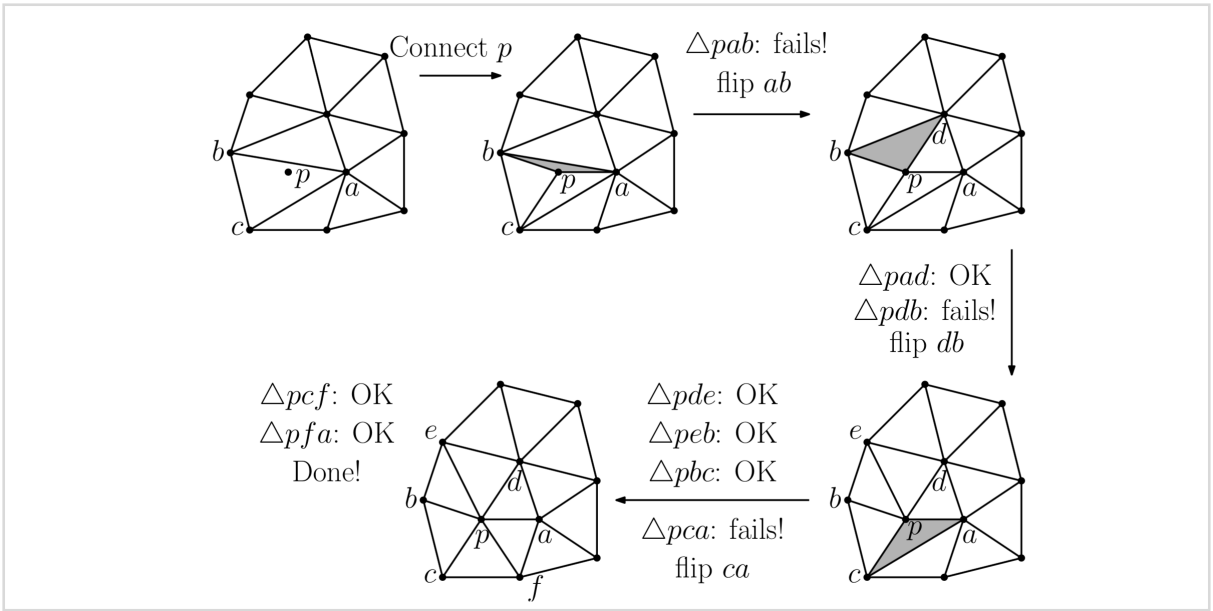
- But as you might expect from previous lectures, we can prove that adding the vertices in random order leads to an O(n log n) expected running time.

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 at the end of Thursday's lecture. 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. We call this the empty circle condition.

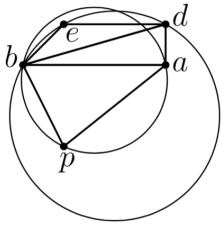
- Every triangle that doesn't contain p was present before we added p. We'll just check triangles that do contain p.
- Each of those triangles has one edge opposite p. I'll argue later that it suffices to check if the triangle contains the vertex on the other side of that edge.
- The algorithm checks 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)
- And that's it! Time permitting, I'll even show you how to do the empty circle test in O(1) time
- 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 $\triangle bde$.

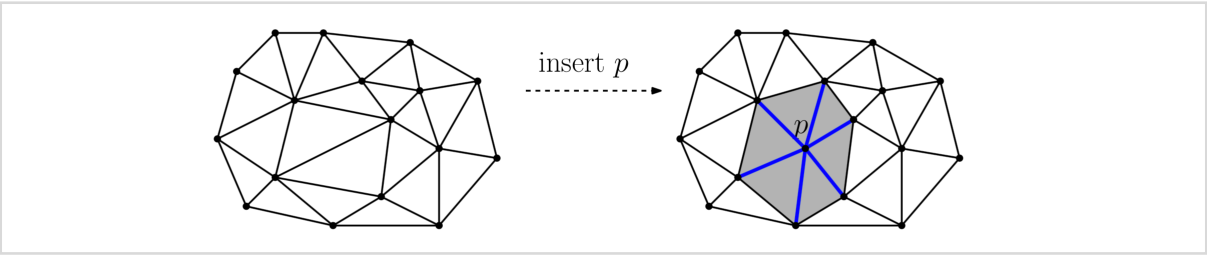
- But triangle bde existed before we even added p, and inductively we can assume we started with a Delaunay triangulation before we added p.

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 something a bit more simple we can do.
- We'll store the sites we have yet to insert in a collection of buckets, one per triangle. The bucket for a triangle contains the sites that lie inside the bucket.
- So when we add a site, we just check the triangle for its bucket.
- 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) 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
- We'll do both of those through backwards analysis.

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.
- Suppose p is the last one we added.
- 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 ≤ 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.
- That isn't an issue in your homework, by the way.
- 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.