Computational Geometry: Lecture 7

Don Sheehy

February 16, 2010

1 A worst case example for the flip algorithm

At a first look, it might seem like there is a lot of slack in our $O(n^2)$ timing analysis. Can it really be the case that in the course of the algorithm, all (or at least a constant fraction) of the edges get flipped? In fact, this can happen.

Consider two lines of $n/2$ points stacked on top of one another. Call the top points $p_1, \ldots, p_{n/2}$ and call the bottom points $q_1, \ldots, q_{n/2}$, numbered from left o right on both cases. For any given triangulation T assign a potential

$$\Phi(T) = \sum_{i=1}^{n/2} \min_{q_j \sim p_i} |i - j|.$$

It is not hard to see that for the Delaunay triangulation, $\Phi(\mathrm{Del}(P)) = 0$.

Let T_{bad} be the triangulation formed by adding edges from p_1 to every q_i and from $q_{n/2}$ to every p_i. Now, we have that $\Phi(T_{bad}) = \sum_{i=1}^{n/2} i - 1 = O(n^2)$.

Now, we observe that a flip can only change the potential by 1. It follows that $\Theta(n^2)$ flips are required to flip T to Delaunay. Thus the analysis of the FlipToDelaunay algorithm is tight.

2 Flips as projections

Last time, we discussed a nice intuition for what it means to do a flip. We said that if we lift 4 points into $\mathbb{R}^3$, that the two triangulations correspond to the upper and lower convex hulls of the tetrahedron formed by the lifted points.

But what if the points are not in convex position? In this case, switching between the upper and lower convex hulls eliminates the inner vertex. We will still call this a valid flip. To tighten up our terminology, we will call the flips that swap two triangles for two new ones, *2-2 flips* or *edge flips*. If a flip swaps one triangle for three, it is a *1-3 flip*, and if it does the opposite, it is a *3-1 flip*.

3 Incremental Delaunay Triangulation

This new kind of flip allows us to change the number of points in a triangulation. So, let's use it to build the Delaunay triangulation one point at a time.

Simplifying Assumption (for now). To simplify things, let's assume that the convex hull of our input set is a triangle and that we know what it is from the start. We'll get rid of this assumption before we're done, but for now, it'll save us some hassle.

Algorithm 1 IncrementalDelaunay

Input: $P \subset \mathbb{R}^d$, with Convex Hull vertices p_1, p_2, p_3
Start with a triangle p_1, p_2, p_3
for $i = 4$ to n **do**
  Find the triangle t containing p_i
  Use a $1-3$ flip to insert p_i into t
  while some edge e is not locally Delaunay **do**
    Flip(e)
  end while
end for

Notice that the inner loop is simply the FlipToDelaunay algorithm we saw before. The algorithm works by finding the triangle containing the next point, splitting that triangle into 3, and then flipping to Delaunay.

Analysis Let's make a couple observations about the flips that happen in the inner loop when inserting p_i.

1. All new triangles have a vertex at p_i.
2. Every flip adds one edge incident to p_i.
3. No flip removes an edge incident to p_i.
4. The number of edge flips performed is equal to the degree of p_i minus 3.

The first observation follows from the fact that any triangle that is Delaunay after inserting p_i but does not have a vertex at p_i, must have been Delaunay before adding p_i and thus is not new. The second and third observations follow because the only edges that are not locally Delaunayare those that are encroached by p_i. The fourth observation follows from the previous two and gives us a way to bound the work needed to flip to Delaunay in this case.

Recall that in our previous analysis, we saw that $O(n^2)$ flips might be needed to get a triangulation to Delaunay. In this case, we are better shape because the number of flips is at most the degree of the new point after the flipping is done. This means that fewer than i flips are needed to insert p_i. Thus, the total number of flips is $\sum_{i=4}^{n} i = O(n^2)$.

We still have an $O(n^2)$ algorithm and, in fact, the performance could be this bad. To find a hard example, we can use the same input that gave us trouble for FlipToDelaunay and insert the points from left to right, one line at a time.

4 Interlude: Dealing with the big triangle problem

Before we add a trick to make our incremental algorithm runin $O(n \log n)$ time, let's deal with that nagging assumption.

One thing we might try is to add a big triangle around the input, use that as the starting triangle, and then throw out the triangles that use these three vertices when we report the final answer. This doesn't really work, because although it is easy to make sure that all of the input lies inside the triangle, it is not easy to make sure that the triangulation you get when you throw out the "outer" triangles is the Delaunay triangulation. The reason is because the Delaunay triangles on the boundary of Del(P) could have arbitrarily large circumballs. Thus, there is no "big enough" for the initial triangle.

It is probably possible to hack this bounding triangle idea, perhaps by reversing the algorithm and flipping out those vertices, but we can do something smarter. Let's just imagine instead that we made the bounding triangle infinitely large. Then, we just have to make sure our predicates don't choke on points at infinity. This is a simple trick that is used all the time.

5 Enter Randomness

Now that the boundary issue is taken care of, let's speed up this algorithms by flipping coins. We'll run the algorithms exactly as before except that this time, we're going to shuffle the order of the inputs. We can now redo our analysis to find the expected runtime.

Recall that in our analysis of the deterministic algorithm, the cost of each insertion was bounded by the degree of that vertex after inserting it. Using our trusty backwards analysis, let's try to figure out the expected cost of inserting point p_i. We do this by assuming that we have inserted i points. Walking backwards one step would correspond to removing the ith point added. However, each of the points is equally likely to be p_i. Thus, the expected number of flips is the expected degree of a vertex in the triangulation.

Recall that Euler's formula implies that the number of edges in a planar graph with i vertices is fewer than 3i. Thus the sum of the degrees is less than $6i$. So,

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

So the expected number of flips for each insertion is only a constant, thus the expected work to update the triangulation is linear.

Linear is pretty good, right? So what was all this nonsense about $O(n \log n)$?

We haven't addressed an important point, that is, how do we find the triangle containing p_i. This phase of the algorithm is called point location and it's the source of the extra $\log n$ factor in the runtime.

6 The History DAG for point location in an incremental construction

The data structure we will be using to do our point location is called the History DAG (that's DAG as in Directed Acyclic Graph). We can think of the History DAG as a set of triangulations layered on top of each other. The ith triangulation is the Delaunay triangulation after i points have been added. There is an edge from a triangle in level i to a triangle in level $i+1$ if they overlap. If the same triangle appears in multiple consecutive triangulations, then only a single edge is necessary.

To do a search for a point q in the history DAG, you start by finding the triangle containing q in the initial triangulation (the one with just three points). Then, you check search through the DAG, choosing the triangle containing q among the outgoing edges from the current triangle.

Next time, we will talk about how to analyze this thing.