CISC 868 Fall 2011
Week 10

November 21, 2011

Randomized Algorithms

Several weeks ago we discussed the expected value, that is, $O(\log n)$, for the number of extreme points of a set of n points that are uniformly distributed in a square. We also saw an algorithm, the gift wrapping algorithm, that computes the convex hull of a set of points with complexity $O(nh)$ where n and h represent the number of input and extreme points respectively. Thus, by our result for points from a uniform distribution one can say that the gift wrapping algorithm has an expected complexity of $O(n \log n)$. This statement is the expectation of the complexity if the algorithm is run over many different input sets that are come from a *uniform random* distribution. The weakness of this type of analysis is that it does not say much about what one would expect to happen for a particular input, or inputs from a particular source, where we know little about how the points are distributed.

In this lecture we will see a result that is quantifiably superior. We will do an expected time analysis, but the analysis will be independent of the distribution of the input. Rather it will only depend on the order in which elements from the input set are processed. Thus, if we run the algorithm repeatedly over the same input set, on average we will get a good running time.

Lets begin with a simple one dimensional problem. Our input set is a set of numbers, in no particular order, and the task at hand is to determine the median value. One obvious way in which we can do this is to sort the numbers first. A linear scan will find the middle element and complete the task giving us an $O(n \log n)$ algorithm. Consider the following approach which we will show has an expected running time of $O(n)$. Our recursive procedure will be more general as it will be able to find the kth ranked element, for any value of k, such that, $1 \leq k \leq n$. See Procedure Select.

It should be clear that Procedure Select successfully determines the median element of S by setting $k = n/2$. The question is how much work does Select do. It is not hard to show that in the worst case, if we are very unlucky, Select can recurse $O(n)$ times resulting in a complexity of $O(n^2)$.

We are much more interested in the expected behaviour of the algorithm. The following lemma provides the ingredients for obtaining the expected running time of the algorithm.

Procedure Select(S, k)

Input: S a set of n numbers, and k an integer such that $1 \leq k \leq n$.
Output: The kth smallest element of S.

Choose a random element of S, v;
// Partition S into S_L, S_v, and S_R.
for *all element* $s \in S$ **do**
case $s < v$: $S_L \leftarrow S_L \cup \{s\}$;
case $s = v$: $S_v \leftarrow S_v \cup \{s\}$;
case $s > v$: $S_R \leftarrow S_R \cup \{s\}$;
// Now recurse over the appropriate subset.
case $|S_L| \geq k$: Select(S_L, k);
case $|S_L| < k \leq |S_v| + |S_L|$: **return** (v);
case $|S_L| + |S_v| < k$: Select$(S_R, k - |S_L| - |S_v|)$;

Lemma 1. *After two recursive calls the expected size of the input to Procedure Select is no more that* $3n/4$.

Proof. Any choice of a v that is ranked between $n/4$ and $3n/4$ would give the favourable split. Since half the elements of S are within this range, so we would expect at most two choices to get an element v that gives a favourable split. □

We can now characterize the expected behaviour of Select with the recurrence relation $T(n) \leq T(3n/4) + g(n)$, where $g(n)$ is a linear function.

Thus we can conclude that $T(n)$ is in $O(n)$.

Randomized Algorithm to Compute a Delaunay Triangulation

Algorithm 1: RandomizedDT

Input: A set of points in general position $P = \{p_1, p_2, \ldots, p_n\}$, with the sequence $p_1, p_2, \ldots, p_n$ in random order.
Output: The Delaunay triangulation (DT) of P

Add three dummy points a, b, c, far enough and outside of P so that all of P is enclosed within a triangle a, b, c , and so that when we are done we can remove a, b, c without affecting the correct DT;
Set DT_0 to triangle abc;
for $r = 1 \ldots n$ **do**
Compute the DT of $\{a, b, c, p_1, p_2, \ldots, p_r\}$, DT_r, by calling the procedure Insert(p_r, DT_r);
end
Remove a, b, c and all edges incident to them, and **return** the resulting triangulation of P

The algorithm itself is quite simple to explain. There are some implementation details that are

somewhat complicated. However, perhaps the most complicated part is the analysis. One can see the inherent advantage of having a simple algorithm that is hard to analyze over a complicated algorithm that is easy to analyze. The algorithm that I describe in these notes assumes that the points are in general position, no three input points on a line, and no four on a circle. The algorithm in the text book is more general and handles the case where there can be four or more input points on a circle. Refer to Algorithm RandomizedDT for a detailed description.

Procedure Insert(p, DT)

1 Find the triangle $p_ip_jp_k$ in DT that contains the point p;
2 Insert the edges p_ip, p_jp, p_kp, into DT;
3 LegalizeEdge(p, p_ip_j, DT);
4 LegalizeEdge(p, p_jp_k, DT);
5 LegalizeEdge(p, p_kp_i, DT);

Procedure LegalizeEdge(p, qr, DT)

1 Let qrs be the triangle adjacent to triangle pqr;
2 // Determine whether the edge qr is illegal by checking to see if p is in the circumcircle of triangle qrs.
3 **if** *qr is illegal* **then**
4 | Flip edge qr with edge ps;
5 | LegalizeEdge(p, qs, DT) *LegalizeEdge*(p,rs,DT);
6 **end**

Obviously all of the work is performed with the Procedure Insert. And Insert uses LegalizeEdge. Figure 1 (This figure comes from lecture notes prepared by David Mount of the University of Maryland, and is used with his permission.) illustrates the result of inserting a point p.

Tracing through the algorithm one can see that the effect of inserting a point p into the existing DT, may create as few as three new triangles, if all adjacent edges are legal. However, it may be the case that there can be numerous calls to LegalizeEdge, and they can cascade. So it appears that we may do a considerable amount of work whenever a new point is inserted. Furthermore, it is not clear that these tests suffice. It can be shown that when we insert a point p into a legal triangulation the only edges that can possibly become illegal are those edges that are incident to a triangle that changes. It can also be shown that any edge incident to the new point p is guaranteed to be legal. These two observations are enough to prove that we are in fact doing all of the required tests when we insert a point p. The justification for why these observations are in fact true can be found in chapter 9 of Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, Computational Geometry:Algorithms and Applications (CGAA).

Accepting that the algorithm is correct, we now need to determine its complexity. We do an expected time analysis. Observe that the complexity of the algorithm is determined by the amount of effort required to find the triangle that contains the point p, in line 1 of Procedure Insert. To do the point in triangle quickly we maintain a pointer from every point to the triangle that contains it. Initially all the points are in the same triangle. As the algorithm progresses new triangles are created. We can update our pointers so that whenever we subdivide a triangle into three new ones, we update the pointers of the points within the triangle. Also whenever we flip an edge we have

to update pointers of points that lie in either of the two triangles. Once the triangle is located we need to perform some number of calls to the Procedure LegalizeEdge. If we store the triangulation in a robust data structure such as the doubly connected edge list (DCEL) we can, in constant time, find adjacent triangles. We also assume that we can determine the legality of an edge in constant time. Thus we can bound the amount of work that is done by the number of calls to Procedure LegalizeEdge.

We use a technique called backwards analysis, where we consider running the algorithm to its conclusion. We then symbolically rewind a tape of the algorithm to get a sense of how much work was done. So suppose we just ran AlgorithmRandomizedDT and the last point that was inserted is the point p. Notice that we can bound the number of calls to LegalizeEdge by the degree of p in the triangulation. In the worst case the degree of p can be $O(n)$. However, we expect that the degree of p is much smaller. Recall that we showed that the number of edges in a triangulation of n points is $3n - 3 - h$. Therefore the number of edges is less than $3n$, and the average degree of a node is at most 6. Therefore, we expect that the degree of p is 6, we can bound the expected number of calls to LegalizeEdge at any iteration at $O(1)$.

Now we go back to the analysis of locating a point within a triangle. We will track the expected number of times that an arbitrary point q has its triangle pointer updated. We will call this a rebucketing operation. So let us assume that we just completed iteration r where we inserted point p. The point q has not been inserted yet, that is q is not a vertex of T_r. What is the probability that q is effected by the insertion of point p at iteration r, and has to be rebucketed? To answer this question consider the triangle that q is in after iteration r. Call it Δ. If none of the vertices of Δ is p then we are certain that q is not affected by the insertion and no rebucketing is necessary for q. If one of the vertices of Δ is the point p then we might have had to rebucket q. From our previous analysis, we know that the number of rebucketing operations for q at this iteration is bounded by a constant. The probability that one of the vertices of Δ is p is $3/r$, because T_r consists of r vertices and Δ is a triangle (with 3 vertices). Extending this reasoning to every non inserted point at iteration r we can say that the total number of rebucketings is a constant times $(n - r)3/r$. To bound the total number of rebucketings throughout the life of the algorithm we have the sum

$$\sum_{r=1}^{n} \frac{3}{r}(n - r) \leq \sum_{r=1}^{n} \frac{3}{r}n = 3n \sum_{r=1}^{n} \frac{1}{r} \in O(n \log n)$$

Thus the total expected time for all the rebucketing operations is $O(n \log n)$ as was claimed at the start. It is not hard to see that the storage requirement is in $O(n)$. Implementing this algorithm is fairly straight forward. The only complications are the data structures for implementing the triangulation, and a search structure for the bucketing operations.

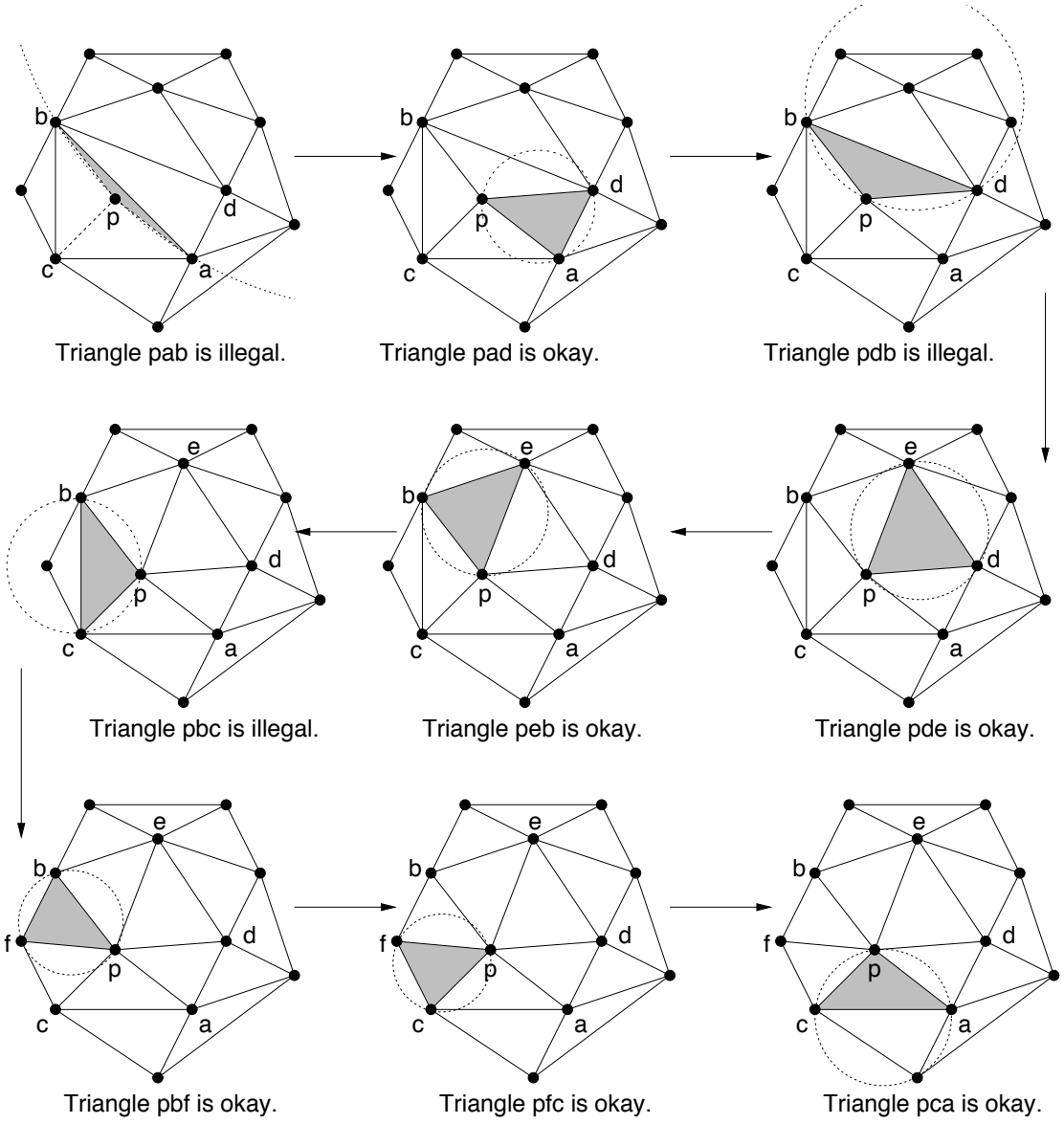

Figure 1: A trace of the calls to LegalizeEdge after inserting the point p.