CISC 868 Fall 2011
Week 1

September 12, 2011

Preliminaries

The objective of this course is to familiarize students with the field of computational geometry. There are several motivators for obtaining this skill. Obviously if one wishes to become a researcher in the field or in a related field then the motivation is crystal clear. However, even if your interests are fairly removed from theoretical computer science, algorithms, or computational geometry in particular, an ability to access the computational geometry literature may provide you with a palpable advantage over others in your field who do not possess this skill.

This course will for the most part deal with abstractions. Problems will be succinctly described, with only a passing mention of applications. The main objective will be to state problems clearly and precisely, design algorithms that are arguably correct and analyze the computational complexity of the algorithm. There is no claim that this paints the entire picture for the practitioner. On the contrary there is a significant effort required to go from the paper version of an algorithm to a robust and practical implementation. However, as stated above, the objectives for this course is to understand the theoretical aspects pertaining to the algorithms. To gain an appreciation of the techniques used to argue about the correctness and complexity of algorithms, that is, how to effectively read and understand results written in the style of an algorithms literature.

The basic input we assume is a set, which we can denote by $S = \{s_1, s_2, \ldots s_n\}$. The set has n distinct elements in no particular order. The n will usually represent the size of the input, denoted by $|S|$, regardless of the nature of the objects. The complexity of an algorithm will be expressed using so called "Big-Oh" notation. I like to think of $\mathrm{O}(f(n))$ as a set of functions where $g(n) \in O(f(n)$ if $g(n)$ is bounded from above by a constant multiple of $f(n)$ for sufficiently large values of n. This can be expressed formally as:

$g(n) \in O(f(n))$ if there exists constants $c > 0$ and $m > 0$ such that $g(n) \leq cf(n)$ whenever $m \leq n$

When we get a Big-Oh result we are providing an upper bound for the complexity of an algorithm,

that is we show that in the worst case the number of operations performed by our algorithm is bounded by a function in a Big-Oh set.

We use Big-Ω to denote lower bounds. The definition is similar to the one above:

$g(n) \in \Omega(f(n))$ if there exists constants $c > 0$ and $m > 0$ such that $g(n) \geq cf(n)$ whenever $m \leq n$

The notation $\Theta(f(n)$ denotes functions that are in both $O(f(n)$ as well as $\Omega(f(n))$.

Examples

Suppose A and B are sets with $|A| = |B| = n$. Show that $A \cup B$ can be determined in $O(n \log n)$ time.

Suppose A and B are sets with $|A| = |B| = n$. Show that $A \cap B$ can be determined in $O(n \log n)$ time.

What is the maximum/minimum size of $A \cup B$? of $A \cap B$?

Our first geometry problem

The convex hull of a set of points is the standard first problem to look at when studying computational geometry.

The *real plane*, also known as $\mathbb{R}^2$ or $\mathbb{R} \times \mathbb{R}$, or even *two dimensional Euclidean space*, is the domain of most of the problems that we will look at. If you draw a circle with diameter 1meter and measure a piece of string that is equal to the perimeter of that circle, you will have a length of string that is π meters long. The quantity π cannot be expressed as a rational number, so it is hopeless to expect to be able to represent it exactly as a finite precision floating point. Nevertheless our underlying assumption when developing algorithms is that we can represent any real number in a constant amount of space. We will also assume that we can do arithmetic, that is, add, subtract, multiply and divide, real numbers in constant time. Quite often we will even assume that we can compute square roots and evaluate trigonometric functions in constant time. The algorithms that we develop will sometimes make decisions based on quantities that may be irrational. Thus, using approximations of a real number would add an additional detail to the all ready complex task of designing an efficient algorithm. By allowing the luxury of a model that can do real arithmetic we can concentrate on the combinatorial aspects of the problem, and this is what will be emphasized in this course.

Returning to the convex hull, our input will be a set of points in the plane. Typically we use the letter n to denote the size of the input. In our example we have n points, where each point p is

represented by an x, an a y coordinate, which will be denoted as p_x, and p_y.

Definition 1. *A subset S of the plane is called* convex *if for any pair of points $p, q \in S$ the line segment $\overline{pq}$ is completely contained in S.*

At this point it would be useful if you could draw examples of convex and con-convex subsets of the plane. If you can't look it up in the course material or on the web.

Definition 2. *The convex hull of a set of points in the plane P, which we will denote as $conv(P)$ is the smallest convex subset of the plane that contains P.*

A way of visualizing $conv(P)$ is to think of the points P as nails protruding from a plane flat board. Stretch a large elastic band so that it surrounds the nails, and let the band shrink so that it is as small as possible. The elastic band is on the boundary of the smallest convex region enclosing the nails.

Another way to say this is that $conv(P)$ is the intersection of all convex sets that contain P. Can you see why this statement follows from the definition?

Definition 3. *Given a set of points P an* extreme point *is a point $p \in P$ such that there is a line L that passes through p and avoids every other point of $conv(P)$. Such a line L is called a* supporting line*. Observe that the line bounds a minimal half-plane that contains P.*

It will be useful to have a concise representation of the convex hull of a set of points. Clearly we can represent the convex hull by its corners or *vertices* which are simply the extreme points of P.

Finding the extreme points is another matter. It turns out that it will be most convenient to get the extreme points in pairs, that is, the pair of corners at the ends of a common side of the convex hull, or formally the *endpoints* of an *edge* of the convex hull.

Given an ordered triple of 3 points p, q, r in the plane, we say that they have

- positive orientation if they define a counter-clockwise orientated triangle;
- negative orientation if they define a clockwise orientated triangle;
- zero orientation if they are collinear.

In the plane, this can be determined by solving a determinant as follows:

$$orientation = \det \begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}$$

Given a pair of distinct points $p, q \in P$ p and q are endpoints of an edge of the convex hull of P, and the edge is oriented clockwise about the boundary of $conv(P)$ if every other point $r \in P$ is to

the right of the edge $\overline{pq}$ oriented from p to q, that is p, q, r have negative orientation, or if p, q, r are collinear, they are oriented with r between p and q, that is p, q, r have zero orientation and r is between p and q. We will leave the notion of being between p and q vague for now.

Algorithm 1: SlowConvexHull.

Input: A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in the plane. Note the points are given in no particular order.

Output: A set *CH* containing the extreme points of P in no particular order.

```
1 CH ← ∅ ;
2 for all ordered pairs (p, q) ∈ P × P with p ≠ q do
3 |   valid ← true;
4 |   for all points r ∈ P, r ≠ p ≠ q do
5 |   |   if r lies to the left of the directed line from p to q or if p, q, r are collinear and r is
  |   |   not between p and q then valid ← false;
6 |   end
7 |   if valid then CH ← CH ∪ {p, q}.;
8 end
9 return CH
```

It should be clear that Algorithm 1 implements our ideas. At this point we need to determine the worst case cost of running the algorithm. The outer loop (for all ordered pairs) uses $O(n^2)$ iterations. The inner loop (for all points) uses $O(n)$ iterations resulting in an overall complexity of $O(n^3)$.

We can design a far more efficient algorithm to compute the convex hull of a set of points. Our efficient algorithm makes use of the same geometric observations as the slow algorithm. The gains in efficiency are obtained by processing the points in a particularly favourable order. The order that we chose is to process the points from left to right. In the case where two points have the same x-coordinate we break the tie using the y coordinate. This is called *lexicographic* ordering or the ordering that is used to sort words in a dictionary. (For example the two letter word "if" comes before "it".)

If we look at the convex hull of a set of points one observes that the points on the boundary of the convex hull can be partitioned into two. An *upper hull* and a *lower hull*. To be precise consider p_1 and p_n in the lexicographic order. The upper hull is a lexicographically increasing sequence of points beginning at p_1 and ending at p_n, so that taking any two points p, q in sequence from the upper hull all other points $r \in P$ are to the right, or possibly collinear with p and q but between them. The lower hull ends at p_1 and begins at p_n, and is lexicographically decreasing, so that taking any two points p, q in sequence from the lower hull all other points $r \in P$ are to the right, or possibly collinear with p and q but between them.

An algorithm to compute an upper hull processes points in lexicographic order. At every iteration we have an upper hull that is correct for the points that have already been processed (and oblivious to points that haven't been processed yet!) Initially we have two points on the upper hull p_1 and p_2, so these two points must be on the upper hull of the set $\{p_1, p_2\}$. At iteration k we add p_k to

the upper hull, because p_k must be on the upper hull of the set $\{p_1, p_2, \ldots, p_k\}$. However there may be points that are on the upper hull of the set $\{p_1, p_2, \ldots, p_{k-1}\}$ but are not on the upper hull of $\{p_1, p_2, \ldots, p_k\}$. The crux of our algorithm is to determine which points those are. We will express these ideas in pseudo code.

We can now make the notion r between p and q precise. If p, q, r are collinear then r is between p and q if it is between p and q in lexicographic order. Consider the following primitive operation which takes three points p, q, r in lexicographic order as input and returns true if r is to the right of the directed edge p, q or r is collinear with p and q and between the two.

Function good(p, q, r)

Input: Three points p, q, r, such that $p < q < r$ in lexicographic order
Output: true if the orientation of p, q, r is negative
return *orientation*(p, q, r) = *negative*

We are now ready to describe the algorithm in pseudo code

Algorithm 2: Efficient Convex Hull

Input: A set of points P in the plane. Note the points are given in no particular order.
Output: A set *CH* containing the extreme points of P in no particular order.
`// Initialize`
Sort the points, resulting in a lexicographically ordered sequence $p_1, p_2, \ldots, p_n$;
`// Upper Hull`
Put the points p_1 and p_2 in a lexicographically ordered list *Lupper*, with p_1 as the first point;
for $i \leftarrow 3$ to n **do**
    Append p_i to *Lupper*;
    while *Lupper contains more than two points* p, q, r *and good*(p, q, r) *is false* **do**
        Delete q the middle of the last three points from *Lupper*.
    end
end
`// Lower Hull`
Put the points p_n and p_{n-1} in a list *Llower*, with p_n as the first point;
for $i \leftarrow n - 3$ downto 1 **do**
    Append p_i to *Llower*;
    while *Llower contains more than two points* p, q, r *and good*(p, q, r) *is false* **do**
        Delete q the middle of the last three points from *Llower*.
    end
end
$CH \leftarrow Llower \cup Lupper$;
return *CH*

The correctness of the algorithm should be apparent from our development, and can be proved by induction on the number of points. Consider the following lemma that where we prove that the upper hull is computed correctly, the correctness of the lower hull follows by symmetry.

Lemma 1. *The upper hull is computed correctly by Algorithm 2.*

Proof. We use induction. For the base case p_1 and p_2 are obviously on the upper hull of $\{p_1, p_2\}$. We assume that we have the correct upper hull for $\{p_1, p_2, \ldots, p_k\}$. The point p_{k+1} is obviously on the upper hull of $\{p_1, p_2, \ldots, p_{k+1}\}$ because it is extreme. We then traverse back on the hull to see what to keep and what to dispose of. The test good(p, q, r) in effect tests to see whether the points in counter-clockwise order on the hull make a right turn. If not then the middle point can't be on the hull. Points in this way are discarded until we find a triple that are good. By the induction hypothesis we remain with an upper hull so that all consecutive triples are good. Thus the upper hull is computed correctly. □

The complexity analysis of this algorithm is a bit tricky. The first sorting step is in $O(n \log n)$. Subsequently we have a while loop nested within a for loop. The for loop iterates once for each element in the sequence. The while loop on the other hand iterates until a good triple is found, and this can vary depending on the example. One can be tempted to say that in the worst case the while loop iterates $O(n)$ times, that is over the entire upper hull, and conclude that the complexity is $O(n^2)$. However upon further reflection one notices that if during one iteration the while loop iterates frequently, then there should be a trade off with other iterations where the while loop does not iterate frequently. The following lemma make this very notion precise.

Lemma 2. *Algorithm 2 computes the upper hull in $O(n \log n)$ time.*

Proof. The sorting step is clearly in $O(n \log n)$. The subsequent step has a for loop that iterates $O(n)$ times with a nested while loop. Observe that every time a triple fails a good test we remove a point from the upper hull. As soon a triple passes the good test the while loop terminates, Thus the total number of times the while loop iterates over the entire life of the algorithm is in $O(n)$. Therefore we can conclude that the complexity of computing the upper hull is in $O(n \log n)$. □

There is an obvious symmetric argument for the lower hull, thus we can conclude with our main result.

Theorem 1. *The convex hull of a set of points in the plane can be computed in $O(n \log n)$ time.*

Polygons

This is the first topic in the text book "Discrete and Computational Geometry" by Satyan Devados and Joseph O'Rourke.

Definition 4. *A* polygon *is the closed region of the plane bounded by a finite collection of line segments forming a closed curve that does not intersect itself.*

We can represent a polygon P as sequence of its vertices in counter clockwise order around the boundary of P. We can notate the boundary of P as ∂P.

Definition 5. *A* diagonal *of a polygon P is a line segment connecting two vertices of P lying in the interior of P and not touching ∂P except at its endpoints.*

Lemma 3. *Every polygon P with more that three vertices has a diagonal.*

Proof. Let v be the leftmost bottommost vertex of P, and let a and b be neighbouring vertices of v on ∂P. If $\overline{ab}$ is a diagonal then we are done. Otherwise let L be a line parallel to segment $\overline{ab}$ passing through v. Sweep L upward from v until it hits the first vertex, x, inside triangle abv. Observe now that $\overline{xv}$ is a diagonal. □

Definition 6. *A* triangulation *of a polygon P is a maximal set of non crossing diagonals.*

Using the fact that every polygon with four or more vertices has a diagonal we can use induction to prove that every polygon has a triangulation. Induction can also be used to prove that every triangulation of a polygon with n vertices has exactly $n-3$ triangles and $n-2$ diagonals. The inductive proof is constructive and thus leads to the following (inefficient) recursive algorithm.

Algorithm 3: Triangulate Polygon Slow

Input: A polygon $P = (p_1, p_2, \ldots, p_n)$.
Output: If P has more than three vertices a diagonal of P is output.

```
1 if P has more than 3 vertices then
2 |   Find a diagonal d of P;
3 |   output d ;
4 |   Let P_1 and P_2 be the two polygons obtained by partitioning P with diagonal d;
5 |   Triangulate Polygon Slow (P_1); Triangulate Polygon Slow (P_2);
6 end
```

A straightforward method can be used to get the diagonal in $O(n)$ time. In essence we implement the construction in the proof of lemma 1.3 of the text book. We test every point of the polygon to see if it lies in the triangle $\triangle abv$, and then find the one that would be hit by the sweeping line. That is doable in linear time. However this method is inherently inefficient. A recurrence relation describing the cost of the algorithm would be:

$$T(n) = cn + T(n_1) + T(n_2)$$

Where n_1 and n_2 respectively denote the number of vertices in P_1 and P_2. To interpret the recurrence we have the cost of finding a diagonal plus the cost of triangulating polygon P_1 and the cost of triangulating polygon P_2. In the worst case every time we find a diagonal with partition into a triangle plus the remaining polygon. Therefore the recurrence can be re-written as:

$$T(n) = cn + T(n-1)$$

This recurrence evaluates to the sum:

$$\sum_{i=0}^{n-3} c(n-i)$$

which in turn is in $O(n^2)$.

Concepts

Going through a sequence of definitions in class can be quite tedious. Review any of the concepts and definitions that were given this week. Look these up in the suggested texts for this class, other texts that you may have, or look them up on the web. See whether the definitions are consistent and whether you really understand what they mean.

If you have difficulties with proofs by induction, recurrence relations, summations, or big-O notation you should review these concepts in any standard algorithms text book.

Disclaimer

I will attempt to fill in holes left by our text with these notes. However, I don't expect that I will provide anything as lengthy as this week's notes every week. This week is different because I presented quite a bit of material that is not in the text.