18.415/6.854 Advanced Algorithms November 26, 2008

Lecture 21: Convex Hull in $\mathbb{R}^2$ and Small-d LP's

Lecturer: Michel X. Goemans

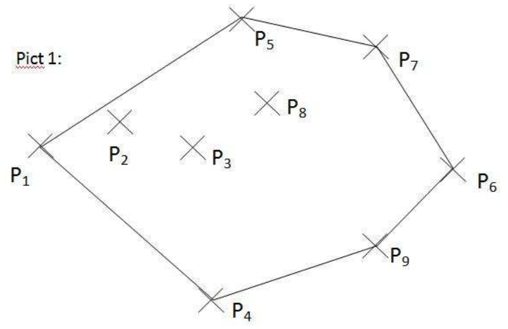
1 Introduction

The first two thirds of this lecture serve as an introduction to this class's coverage of computational geometry. The reader is referred to [1] for additional coverage. We'll consider several approaches for finding the convex hull of a set of points. The first three algorithms discussed pertain to points in $\mathbb{R}^2$. Then, and more extensively in later lectures, we'll generalize to consider the $\mathbb{R}^n$ case.
In the final third of this lecture we will return to the previous topic of linear programming, this time considering programs relevant to computational geometry- dealing with a small fixed number of variables. In the general n-dimensional case, no strongly polynomial algorithm is known. For the case of a small fixed dimension, however, we will see a deterministic algorithm that runs in polynomial time without dependence on the size of the coefficients defining the problem. In the next lecture, we'll see a randomized version that runs in linear time.

2 Convex Hulls in 2-Dimensions

Definition 1 *A set of points, C, is* ***convex*** *if the line segment joining any pair of points of C lies entirely in C. The* ***convex hull*** *of a set of points, S, is the intersection of all convex sets containing S.*

We wish to find the convex hull of a given set of n points $S = p_1, p_2, ...p_n \in \mathbb{R}^2$. Representing a general convex hull is a nontrivial problem, but in two dimensions it is simple enough to come up with a convention. For purposes of this lecture, the convex hull H of $S \in \mathbb{R}^2$ will be expressed non-uniquely as a clockwise ordered list of the vertices defining the boundary of the hull, and we will refer to this list as the convex hull. For example, given the points:

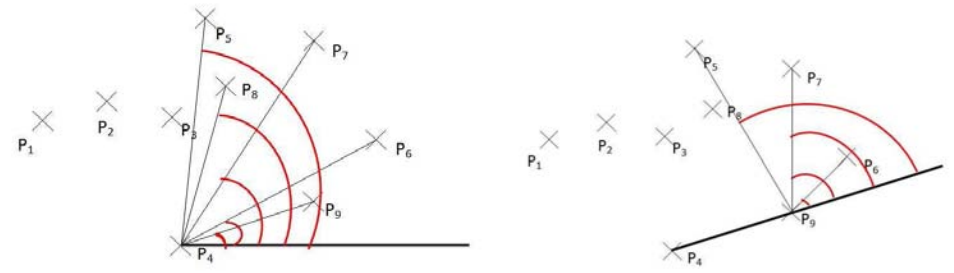


the hull H would be an ordered list $(p_4, p_1, p_5, p_7, p_6)$. Alternatively, the list $(p_1, p_5, p_7, p_6, p_4)$ would also be a correct output.

2.1 The Gift Wrapping Algorithm

The idea of the gift-wrapping algorithm is as follows: if we can start from some point on the boundary of the desired hull H, we can consider wrapping a string around all points in the set. This string (or wrapping paper in three dimensions) will contact only points on the boundary of the hull, one-by-one in order. The algorithm therefore starts with a point known to be in the hull. We can take, for example, the lowest point in the set (smallest y-coordinate)- if there are more than one, then the leftmost of these. In the example, this is the point p_4. The algorithm then considers every edge (p_4, p_i) and calculates the angle between that edge and horizontal, and chooses the point corresponding to the edge with smallest angle. The motion of "sweeping around" continues by iterating this step, finding the edge forming with smallest angle with the previous previous edge in the list for each subsequent point.

Pict 2



Algorithm 1 Gift Wrapping Algorithm

```
$i \leftarrow i_0$ with $p_{i_0}$ the lowest,leftmost point in the set $S$
$H \leftarrow \{i_0\}$
**repeat**
  let $j : (p_i, p_j)$ be the edge forming the smallest angle with previous convex hull edge
  $i \leftarrow j$
  $H \leftarrow \text{prepend}(H, p_i)$
**until** $i = i_0$
**return** $H$
```

Runtime Analysis. The initial "find min" runs in $O(n)$ for each vertex in the hull, say there are h of them, the algorithm must calculate the angle of the line, an operation requiring constant time, $O(1)$ thus, in total, $O(n)$ for each vertex in the hull and therefore, $O(nh)$ for the entire algorithm. This running time bound is *output-sensitive*. There exist $2d$ convex hull algorithms running in time $O(n \log h)$.

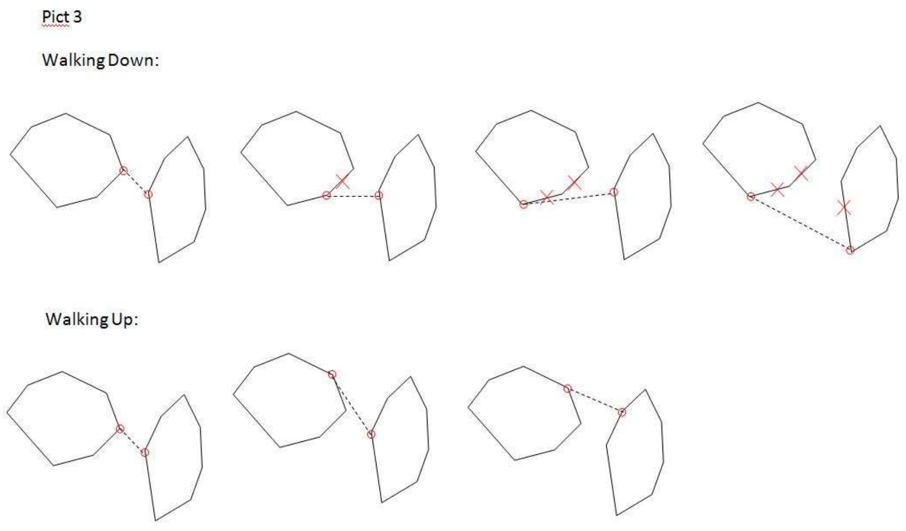
2.2 Divide and Conquer Algorithm

Given the convex hull A of the left half of a set of points, and B that of the right half, if one can easily compute the convex hull of the entire set, then this method can be used to recursively compute the convex hull of a set S with a divide-and-conquer approach. In the divide step we recursively partition our set using median search to divide sets into $>$ and $\leq$ this median. The conquer step becomes trivial since the convex hull of a single point is just the point itself. Finally we recursively merge the hulls of left and right subsets. We now examine the MERGE procedure in detail.
Our MERGE(A,B) merges (disjoint) left and right hulls by finding the lower and upper segments which connect the hulls to form the total convex hull. All points no longer on the boundary are

removed from the merged hull. In each case the key is for the lower segment to be *lower tangent* to both A and B and the upper segment to be *upper tangent* both A and B.

Definition 2 *A line segment is* ***lower tangent*** *to a set, S, if it intersects S at one point and if the remainder of S is above the line L formed by extending the segment to infinity in both directions. Similarly, a segment is* ***upper tangent*** *to a set, S, if it intersects S at one point and if the remainder of S is below L.*

The MERGE procedure (see Algorithm 2) finds these segments by beginning with the segment connecting the right-most point of A and the left-most point of B and then alternates between walking down B and A; it switches to walking down the other whenever the current segment becomes lower tangent to one of them. This continues until the segment is lower tangent to both A and B.



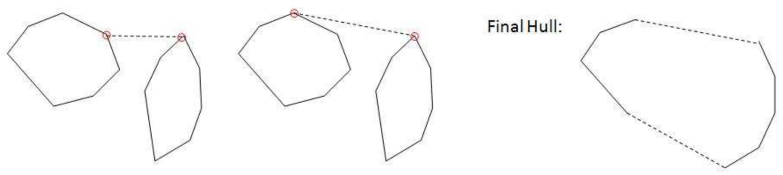
Claim 1 *The algorithm terminates.*

Lemma 2 *At any time during the execution of the algorithm, the segment (a_i, b_j) intersects neither the interior of A nor the interior of B.*

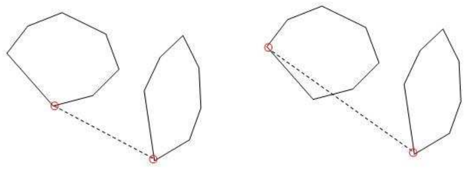
Proof: Clearly this is true initially, when a_i is the right-most point of A and b_j is the left most point of B. So the lemma is true iff either form of moving (taking a step clockwise around one hull or counterclockwise around the other) preserves the property. This is the case because, if we intersect the interior of say B for the first time by moving along B, in fact, we must have been at a lower tangent of B. The proves the lemma. $\square$
To prove the claim that the algorithm terminates, notice that the lemma implies that the algorithm will never consider a point in A past the leftmost point. Likewise for B and the rightmost point. This completes the proof that the algorithm must terminate.

Algorithm 2 MERGE left and right convex hulls.

```
Given: the convex hulls $A = (a_0, a_2, ...a_{m-1})$ and $B = (b_1, b_2, ...b_{n-1})$
Find: The convex hull $H$ of $A \cup B$
(i) Find the upper connecting segment
  $a_i \leftarrow$ the right-most point of $A$
  $b_j \leftarrow$ the left-most point of $B$
  while $(a_i, b_j)$ is not a upper tangent of $A$ and $B$ do
    while $(a_i, b_j)$ is not a upper tangent of $B$ do
      $j \leftarrow j + 1$
    end while
    while $(a_i, b_j)$ is not a upper tangent of $A$ do
      $i \leftarrow i - 1$
    end while
  end while{thus walking counterclockwise around $A$ and clockwise around $B$}
  $(u_A, u_B) \leftarrow (i, j)$
(ii) Find lower connecting segment
  $a_i \leftarrow$ the right-most point of $A$
  $b_j \leftarrow$ the left-most point of $B$
  while $(a_i, b_j)$ is not a lower tangent of $A$ and $B$ do
    while $(a_i, b_j)$ is not a lower tangent of $B$ do
      $j \leftarrow j - 1$
    end while
    while $(a_i, b_j)$ is not a lower tangent of A do
      $i \leftarrow i + 1$
    end while
  end while{thus, the algorithm walks clockwise around $A$ and walks counterclockwise around $B$}.
  $(l_A, l_B) \leftarrow (i, j)$
(iii) Merge hulls
  $H \leftarrow (a_{u_A}, b_{u_b}, \ldots, b_{l_b}, a_{l_a}, a_{u_A} - 1)$ {taking $a$ indices mod $m$ and $b$ indices mod $n$}
  return $H$
```



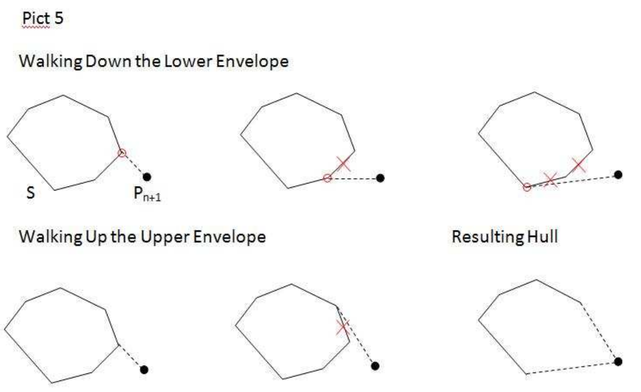
Pict 4



Runtime Analysis. Since MERGE(A,B) must terminate after at most n steps, where n is the total number of points in both hulls, MERGE(A,B) has runtime $O(n)$. Considering the recursion used in the divide step (merge sort requiring only $O(n)$ time), $T(n) = 2T(n/2) + O(n)$ thus, the entire procedure's runtime is $O(n \log n)$.

2.3 Incremental Algorithm

We now consider an algorithm based on the idea of efficiently adjusting a known convex hull H of a set S to obtain the convex hull H' of $S \cup \{p\}$ whenever we add a single point p to S.
One approach might be as follows: if the new point p lies inside H then ignore it; if it lies outside H, figure out how to add it to H to get H'. However, constructing an entire convex hull this way can easily take quadratic time since it takes $O(n)$ to check the position of each new p_{i+1} relative to each boundary segment of the hull H_i.
This approach can be rescued in two different ways. One is to randomly order the points and one can then prove that the expected runtime of this randomized incremental algorithm is $O(n \log n)$. Or, and this is the approach we follow now, we can first sort the points by their x coordinate. Then at each iteration, we know that the point p_i will be added to the hull, since it is the right-most point of the set $\{p_1, \ldots, p_i\}$, and we just have to work outward from p_{i-1} in the hull H_{i-1} to identify the vertices forming the upper and lower tangents of p_i with H_{i-1}. Hence we use a procedure similar to the technique we saw in the MERGE step of the divide-and-conquer approach, testing edges first clockwise around H_{i-1} to find a lower tangent, and then counterclockwise to find an upper tangent.


Pict 5
Walking Down the Lower Envelope
S
P_{n+1}
Walking Up the Upper Envelope
Resulting Hull

Edges and vertices are removed when they are between the intersections of the two tangent lines. The test for finding a tangent edge is simple: consider extending the current edge connecting p_{n+1} with the hull of S to a full line. If all of S is above this line, the line is lower tangent. If all of S is below this line, the line is upper tangent. Thus, a simple test to decide whether or not to continue walking is to check if the next point which would be walked to is above or below the extended line of (p_{n+1}, p_i). If looking for an upper tangent, one stops when the next point is found to be below the line. If looking for a lower tangent, one stops when the next point is found to be above the line.

Definition 3 *Points in the hull of S above the left most point of S are said to comprise the* ***upper envelope*** *of S. Points in the hull of S below the left most point of S are said to comprise the* ***lower envelope*** *of S*

Algorithm 3 MERGE(H,p) incremental merge step

```
(i) Find lower tangent segment of $p_{n+1}$ and $H$
  $p_i \leftarrow$ the right-most point of $H$
  while $(p_i, p_{n+1})$ is not a lower tangent to $S$ do
    Remove $p_i$ from $S'$
    $p_i \leftarrow p_{i+1}$
  end while
  $l_H \leftarrow i$
(ii) Similarly, find upper tangent segment $(p_{n+1}, p_{u_H})$ between $p_{n+1}$ and $H$.
(iii) Compute hull
  $H' \leftarrow (p_{u_H}, p_{n+1}, p_{l_H}, \ldots, p_{u_H - 1})$ {taking indices mod $n$ where appropriate}
  return $H'$
```

Runtime Analysis. The initial sort of the points by their x coordinate takes $O(n \log n)$ time. For each addition of a new point, p_{n+1}, the number of iterations performed equals the number of edges deleted from the hull of S in making the hull of S'. Therefore, since the total number of edges deleted is upper bounded by the total number of edges created and since at most two edges are created whenever we add a point, we derive that the entire algorithm performs $O(n)$ iterations. Each iteration takes $O(1)$ time, for a running time of $O(n)$ over all iterations, and a total of $O(n \log n)$ taking into account the initial sort.

2.4 A Lower Bound on Two-Dimensional Comvex Hull Computations

Theorem 3 *Convex hull algorithms for n points in $\mathbb{R}^2$ is as hard as sorting.*

Proof: We reduce the problem of sorting n numbers $x_1, x_2, ..., x_n \in \mathbb{R}$ to a convex hull computation. Consider the set of points $S = ((x_1, x_1^2), (x_2, x_2^2), ..., (x_n, x_n^2)$ on a parabola in $\mathbb{R}^2$. Knowing the ordering in which these points appear on their convex hull allows to easily sort the original numbers $x_1, \cdots, x_n$ as the orderings are the same (up to a possible cylic shift). □

However, we have to be careful how we interpret that result. Indeed, the classical $\Omega(n \log n)$ lower bound for sorting applies in the *comparison* model, but in the comparison model, one cannot even compute the convex hull. See Sedgewick and Wayne [2] for a more detailed discussion. Yao [3] has shown that in the *quadratic decision tree model* in which one can test the sign of a quadratic polynomial, the number of operations required for computing the convex hull of n points in $\mathbb{R}^2$ is $\Omega(n \log n)$.

3 Convex Hulls in Higher Dimensions

In higher dimensions, we can no longer represent convex hulls as simple ordered lists. The boundary of a d-dimensional convex hull is a collection of $d-1$-dimensional polytopes, which in turn are described by "faces" of dimension $0, \ldots, d-2$. The terminology for faces is the following:

dim	name
0	vertices
1	edges
d-2	ridges
d-1	facets

To describe such a hull, one typically constructs an *incidence graph.* The vertices of this graph may either correspond to all faces, or just to the ridges and facets. We connect a k-dimensional face F with a $k-1$-dimensional face F' if F contains F'.

What is the complexity of the output? In 2 dimensions, the number of faces is $O(n)$. In 3 dimensions, Euler's formula says that $h - e + f = 2$ where h is number of vertices, e is number of edges, and f is number of faces, and this implies that $e, f = O(n)$. In 3 dimensions, the gift wrapping algorithm as well as an incremental algorithm run in $O(n^2)$ time, while a more complex divide-and-conquer algorithm can be made to run in $O(n \log n)$ time. For higher dimension D, one can show that the number of facets is $O(n^{\lfloor d/2 \rfloor})$, so this is definitely a lower bound on the time required to construct a convex hull. Not surprisingly, in general, convex hull algorithms are considerably more complicated in higher dimensions. In the next lecture, we'll see a simple randomized algorithm achieving the lower bound (for $d > 3$).

4 Linear Programming in fixed dimension

Consider a linear program:

$$\text{Max}\, c^T x$$
$$Ax \leq b,$$

where $A \in \mathbb{R}^{n \times d}$, $x \in \mathbb{R}^d$, and the dimension d is fixed (not part of the input).

As said in earlier lectures, a strongly polynomial (i.e. not dependent on the size of the entries of the data) time algorithm for linear programming in the general case is not known. However, for fixed dimension, we'll show that such algorithm exists, and we will present a simple randomized algorithm whose running time will be linear in n (for fixed d). This was sketched in this lecture, but the derivation will be formalized in the next lecture.

References

[1] M. de Berg, O. Cheong, M. van Kreveld and M. Overmars, “Computational Geometry”, 3rd edition, Springer, 2008.

[2] R. Sedgewick and K. Wayne, “Algorithms”, 4th Edition.

[3] A. C.-C. Yao, “A Lower Bound to Finding Convex Hulls”, *J. ACM*, **28**, 780–787, 1981.