COMPSCI 330: Design and Analysis of Algorithms 10/16/2014

Lecture 14

Lecturer: Kyle Fox *Scribe: Yilun Zhou*

1 Overview

This lecture introduces one aspect of computational geometry, convex hull. The definition of convex hull is given and two algorithms, Jarvis's March and Graham's Scan are introduced that efficiently computes convex hull. Their running time is analyzed and correctness proven.

2 Convex Hull

Definition 1. *Given point set $P = p_1, p_2, ..., p_n$, the convex hull (CH) of P is the smallest convex polygon containing P.*

Remark 1. *There are several constraints in the definition that should be made clear:*
1. smallest: no subset of polygon encloses P.
2. convex: between any two points q_1 and q_2 in the polygon, the line segment connecting these two points also lies in the polygon.
3. polygon: a cycle of edges connected by its vertices.

2.1 Application

There are several applications for convex hull algorithm. One application is collision detection. Detecting if two arbitrary geometric shapes collide (overlap each other), is generally a computational expensive calculation. However, there are efficient algorithms to detect whether two convex polygons collide. Thus, we can try to detect whether the convex hulls of the two shapes of interest collide. If the convex hulls do not collide, then the two shapes of interest cannot collide because all points in the shape of interest are contained in its convex hull. This preliminary test can avoid many computational expensive calculations.

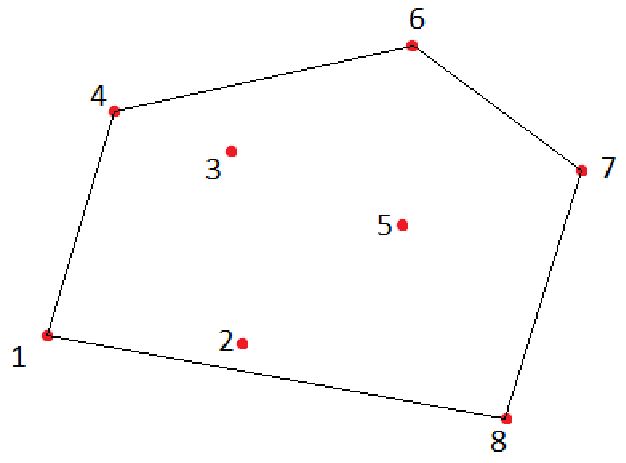
3 Convex Hull Algorithm Overview

There are many algorithms to find convex hull. This section will first introduce the input and output format for convex hull algorithm, then make one assumption about input data, and finally give intuitive examples to some simple cases.

3.1 Input and Output Format

The input to the algorithm is given as two arrays, `x[1...n]` and `y[1...n]` where `x[i]` and `y[i]` give the coordinate of the i-th point.
The output of the algorithm is two arrays, `next[1...n]` and `prev[1...n]` where `next[i]` gives the index of the point that is counter-clockwisely (CCW) next point of the i-th point and `prev[i]` gives the index

of the point that is clockwisely (CW) next point of the i-th point, if the i-th point is a vertex of the convex hull. If the point is not a vertex of the convex hull (i.e. in the interior of the polygon), then `next[i]=prev[i]=0`. For example, in the following graph, `next[8]=7, prev[8]=1, next[2]=prev[2]=0`.

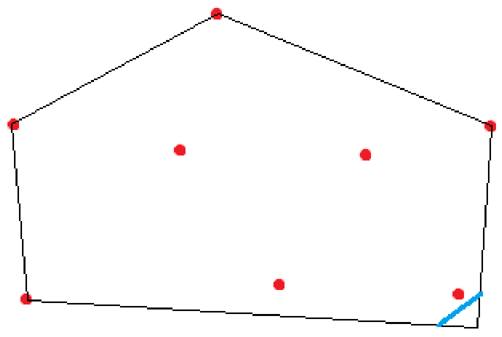


Remark 2. Technically speaking only `next[]` *or* `prev[]` is needed. However, keeping track of two arrays will prove convenient in one algorithm that will be introduced below.

The previous output specification requires the following lemma:

Lemma 1. *Each vertex of the convex hull is a point in P.*

Proof. If a vertex is not in P, then there exists a smaller convex polygon by shortcutting that vertex. The following graph shows the shortcutting. The blue line is the shortcut that can decrease the area of the current convex hull.



□

3.2 Assumption

To ease the analysis of algorithm, we make the following assumption:
General position assumption: no three points q, p, r are co-linear (lie on the same line).

3.3 Intuitive Analysis of Simple Cases

3.3.1 One Vertex

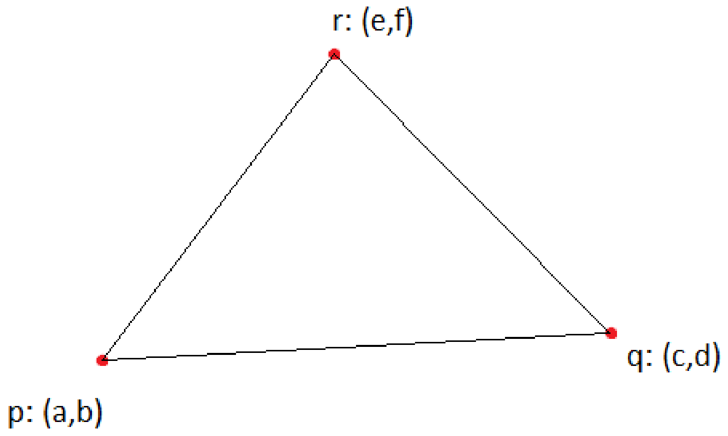
If there is only one vertex, then the convex hull is just that point itself. So `next[1]=prev[1]=1`.

3.3.2 Two Vertices

If there are two vertices, then the convex hull consists of the two vertices as well as the line segment between them. Thus, `next[1]=prev[1]=2, next[2]=prev[2]=1`.

3.3.3 Three Vertices

If there are three vertices, it is easy to see that the convex hull is the triangle formed by those three vertices (according to general position assumption, these three points are not co-linear). The following graph shows this situation with points as well as their coordinates denoted.



However, the convex hull algorithm also needs to specify the "order" of the triangle because following `next[]` has to traverse the triangle in CCW order. Thus, we introduce an algorithm `CCW(p,q,r)` that tests whether `p, q, r` are in CCW order (as shown by the above graph) or not. We assume the coordinates in the graph.

```
CCW(p,q,r):
    return (d-b)*(e-a)<(f-b)*(c-a)
```

This algorithm uses the fact that if the z-component of the cross product of $\bar{pq}$ and $\bar{pr}$ is positive, then rotating from $\bar{pq}$ to $\bar{pr}$ will be a CCW rotation. So `p, q, r` are in CCW direction.
Thus, using `CCW` defined above, we derive the following algorithm to output the convex hull for three vertices.

```
if CCW(p,q,r):
    next[p] = q, prev[q] = p
    next[q] = r, prev[r] = q
    next[r] = p, prev[p] = r
else:
    next[p] = r, prev[r] = p
```

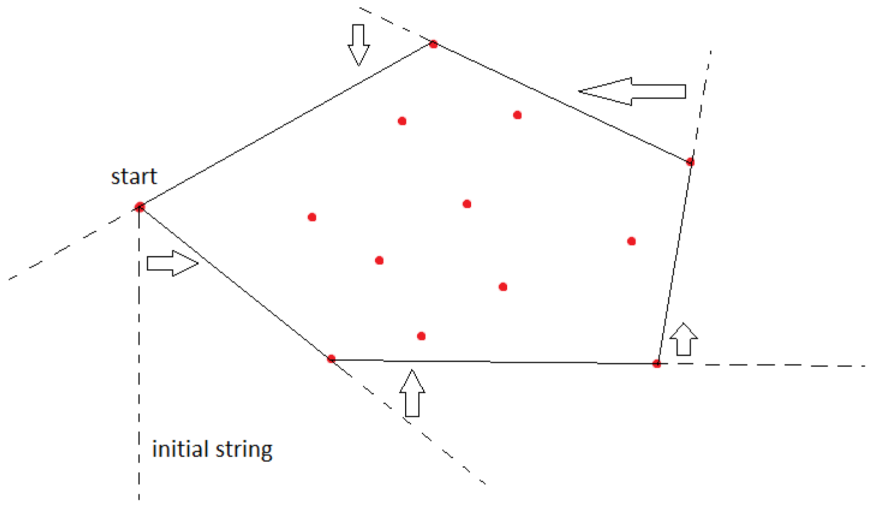
```
        next[q] = p, prev[p] = q
        next[r] = q, prev[q] = r
```

4 Jarvis's March

Jarvis's March is a straightforward algorithm that computes convex hull for a set of points. It relies on the following two facts:
1. The leftmost point must be one vertex of the convex hull.
2. If point p is a vertex of the convex hull, then the points furthest clockwise and counter-clockwise are also vertices of the convex hull.

4.1 Intuitive Description

Jarvis's March can be visualized by considering all points as iron nails on a board. In addition a string is attached to the leftmost nail l. Then from initially vertically downward position, the string rotates CCW until hits the first nail. This nail is the furthest *CW* of l (it is the first CCW direction nail, which is equivalent to last CW direction nail). Thus, according to fact 2, it is a vertex of the convex hull. Around this nail the string continues to rotate CCW until it hits the next nail, which again is a vertex of the convex hull. This process continues until the string hits the starting point (leftmost point) again. The contour of the convex hull is then delineated by the string. The following graph shows how Jarvis's March works. The solid lines are the edges of the convex hull. The dashed lines are the "free end" of the string to be wrapped.



4.2 Pseudocode

The pseudocode of Jarvis's March algorithm is shown below:

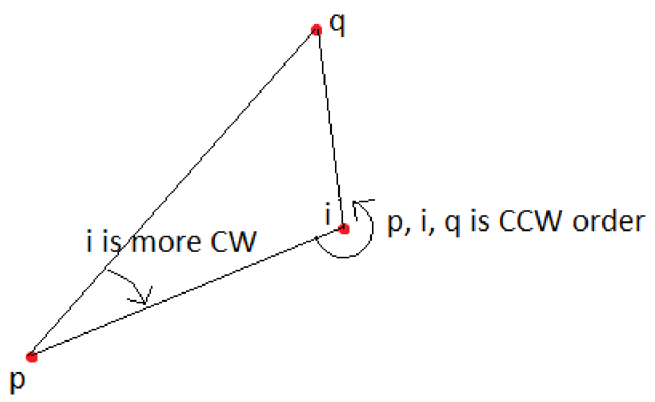
```
Jarvis(x[1...n], y[1...n]):
    l=1
    for i from 2 to n:
        if x[i]<x[l]:
```

```
5               l=i
6       p=l
7       repeat:
8           q=p+1
9           for i from 1 to n:
10              if p!=i and CCW(p,i,q):
11                  q=i
12          next[p] = q
13          prev[q] = p
14          p = q
15      until p=l
```

4.3 Analysis

4.3.1 Correctness

Jarvis's March consists of two stages.
During the first stage (line 2 to 6), it finds the leftmost point by comparing the x-coordinate. After it finds the leftmost point, `p` is initialized to that point.
During the second stage, it finds the point that is furthest CW to `p`. To do this, it first chooses an arbitrary point `q`, then it iterates over all points to to find the furthest CW point. This is very similar to finding the maximum element in an array but here the comparison is not based on numerical values, but on `CCW` property. More concretely, `i` is more CW than `q` for `p` if `CCW(p,i,q)=true`. This point is illustrated below:



This process ends when `p` becomes the leftmost point again, at which time the string wraps completely around the set of points. Therefore, this procedure satisfies both facts stated at the beginning of this section. So it correctly finds the convex hull for a set of points.

4.3.2 Running Time

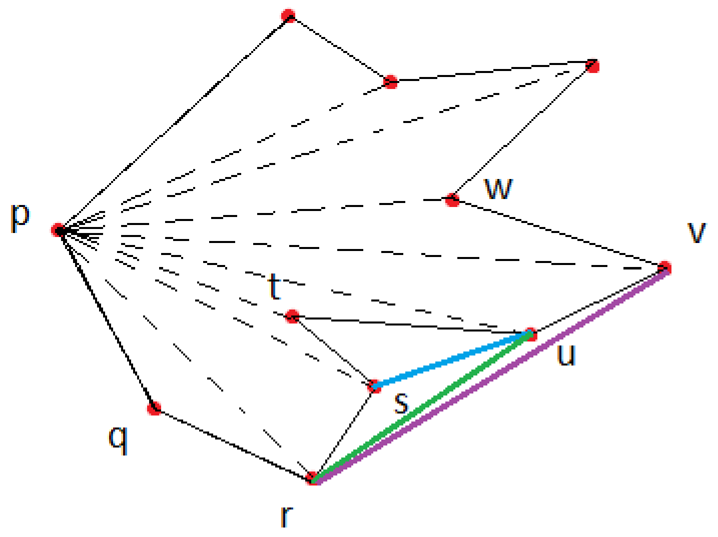
Finding the leftmost point requires traversal of all points and takes $O(n)$. In the second stage, finding the furthest CW point for a given point takes $O(n)$ and this routine needs to be run for h times, where h is the number of vertices of the convex hull. Thus, the running time is $O(hn) \leq O(n^2)$ because $h \leq n$. In addition, because the running time depends on the output of the algorithm (i.e. how many vertices are on the output convex hull), it is called *output-sensitive algorithm.*

5 Graham's Scan

Graham's Scan is another algorithm to identify convex hull. On worst-case it is faster than Jarvis's March, though it is a little bit more difficult to understand.

5.1 Intuitive Description

Like Jarvis's March, Graham's Scan also first identifies the leftmost point. It then created a jagged polygon by connecting points in CCW order with respect to *the leftmost point*, starting from the most CW point. The result is the solid black line shown below:



Then Graham's Scan will go from `p` in CCW order (i.e. `p q r s t u v w ...`) and use `CCW` function to test whether three consecutive points form CCW order. This test will first fail on `s t u`, which means that `t` should be an interior point. Then the algorithm discards `t` and retracts to `r s u`. Note that it cannot blindly proceed to `s u v` because it only checked that `r s t` are in CCW order but not that `r s u` (after it discards `t`) are in CCW order.
Thus, it goes back to check `CCW(r,s,u)` and it fails again. So `s` is discarded. *Note that the first point of the three needs to go back by one. So* `CCW(q,r,u)` *is checked. This time it passes.* Then `CCW(r,u,v)` is checked. However, it turns out `u` is also an interior point so it is discarded. The same process repeats. Finally, `CCW(r,v,w)` passes so the final edge on the convex hull is $\bar{rv}$, which encloses `s t u`. This process is shown by the blue, green, and purple line segments.
This algorithm terminates when three consecutive points go back to the leftmost point again.

5.2 Pseudocode

The pseudocode of Graham's Scan algorithm is shown below:

```
1   Graham(x[1...n], y[1...n]):
2       find the leftmost point
3       sort all other points in CCW order with respect to the leftmost point
4       rename leftmost point to be 1
5       rename other points to be 2...n in CCW order
6       for i=2 to n:
7           next[i-1] = i
8           prev[i] = i-1
9       next[n] = 1
10      prev[1] = n
11      p=1, q=2, r=3
12      repeat:
13          if CCW(p,q,r):
14              p=q, q=r, r=next[r]
15          else:
16              next[p] = r
17              prev[r] = p
18              next[q] = 0
19              prev[q] = 0
20              q = p
21              p = prev[p]
22      until r==1
```

5.3 Analysis

5.3.1 Correctness

This algorithm follows the procedure described in the first section. First, it identifies the first point and sorts other points in line 2 to 5. Then it creates the ”jagged polygon” in line 6 to 10. After this, it progresses `p q r` around the polygon. Line 13 to 14 covers the case when `p q r` are in CCW order and simply advances them by one. Line 16 to 21 covers the case in which `q` is an interior point and needs to be discarded. This process repeats until `r` becomes the leftmost point again. Thus, this algorithm correctly identifies the convex hull.

5.3.2 Running Time

Identifying the leftmost point takes $O(n)$ time. The sorting by CCW order takes $O(n \log n)$ time. Finally, advancing the three consecutive points take in total $O(n)$ time. This can be seen by considering the fact that in each iteration either `r` advances by one or `q` is discarded. `r` can only advance $O(n)$ times until it hits the leftmost point again and there are at most $O(n)$ points to be discarded. Thus, the iteration finishes in $O(n)$ time. So the total running time is $O(n \log n)$.

6 3D Extension

Extending convex hull algorithm from 2D to 3D is a significant task due to the complexity of representing shapes in 3D. In 3D the convex hull becomes the convex polytope and the output needs to be a list of vertices, edges and faces (represented by edges in CCW order). Detailed discussion of constructing 3D polytope is beyond the scope of the course but one commonly used algorithm is incremental construction, whose basic idea is to start from one point and extend the polytope when a new point is discovered and it is not in the existing polytope. An animation of this process can be found at http://www.cse.unsw.edu.au/ lambert/-java/3d/hull.html. This algorithm runs in $O(n^2)$ in worst case and $O(n \log n)$ in average case if the points are given in random order.

7 Summary

This lecture introduces two algorithms for computing convex hull. Jarvis's March is an easy-to-understand algorithm that runs in $O(nh)$ where n is the number of points given and h is the number of vertices of the convex hull. Graham's Scan is less intuitive but results in a running time of $O(n \log n)$. Usually Graham's Scan is a better choice (especially when vertices are already sorted in CCW order) but when prior knowledge is available (e.g. that there are only a few vertices in the final convex hull), Jarvis's March may be preferable.