Computational Geometry
15-451 CMU Fall 2014
Danny Sleator

Nov 12, 2014

Some of this material came from notes by Gary Miller

Boilerplate:

What is Computational Geometry(CG)?
Why CG in algorithms class?
Where is CG used?

At present CG is the design and analysis of algorithm for geometric problems that arise on low dimensions, 2 or 3. Most algorithms work well in 2D while 3D problems are not as well understood.

CG will us to see our old algorithms in use and see new ones.

Some application of CG are:

Computer Graphics
  images creation
  hidden surface removal
  illumination

Robotics
  motion planning

Geographic Information Systems
  Height of mountains
  vegetation
  population
  cities, roads, electric lines

CAD/CAM computer aided design/computer aided manufacturing

Computer chip design and simulations

Scientific Computation
  Blood flow simulations
  Molecular modeling and simulations

Basic approach used by computers to handle complex geometric objects

Decompose the object into a large number of very simple objects with simple interactions.

Examples:

* An image might be a 2D array of dots.
* An integrated circuit is a planar triangulation.
* Mickey Mouse is a surface of triangles

Basic algorithmic design approaches

Divide-and-Conquer
* There are two different types of Divide-and-Conquer.
  e.g. Merge sort
  e.g. Quick sort

* Line-Sweep in 2D.

* Random Incremental

We will have three lectures on computational geometry:

* 2D convex hull of a point set
* random incremental algorithm for closest pair
* divide and conquer algorithm for closest pair
* sweep line algorithm for intersecting a set of segments
* an expected linear time algorithm for 2D linear programming.

First we discuss what abstract object we will need and their representation

Abstract Object	Representation
Real Number	Floating Point, Big Number
Point	Pair of Reals
Line	Pair of Points, A set of constraints
Line Segment	Pair of Points, A set of constraints
Triangle	Triple of Points, etc

Suppose $P_1, P_2, \dots P_K \in \mathbb{R}^d$

Linear Combinations

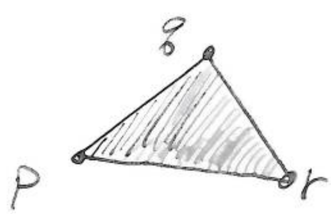
Subspace $= \sum \alpha_i P_i$ for $\alpha_i \in \mathbb{R}$

Affine Comb

Plane $= \sum \alpha_i P_i$ s.t. $\sum \alpha_i = 1$, $\alpha_i \in \mathbb{R}$

Convex Comb

Body $= \sum \alpha_i P_i$ s.t. $\sum \alpha_i = 1$, $\alpha_i \geq 0$

e.g. 

$= \{ \alpha p + \beta q + \gamma r \mid \alpha + \beta + \gamma = 1, \; \alpha, \beta, \gamma \geq 0 \}$

Primitive Operations

1) Equality P=Q? (Can be hard in practice due to round-off error)

2) Line side tests

```
Input: P1,P2,P3
Output: LEFT If P3 is to the left of ray P1 -> P2
        RIGHT If P3 is to the right of ray P1 -> P2
        ON If P3 is to on ray P1 -> P2
```

Subtract P1 from both of the other vectors. Let V2 = P2-P1 and V3 = P3-P1. Now the cross product V2xV3 is the signed area of the parallogram formed by V2 and V3. This area is > 0 if and only if P3 is left of ray P1 -> P2.

```
       V2
       *--------*
      /        /
     /        /
    /        /
   *--------*V3
```

-------------------------Ocaml Code-------------------------

```
let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let (++) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let dot (x1,y1) (x2,y2) = (x1*x2) + (y1*y2)

type side = LEFT | ON | RIGHT

let line_side_test p1 p2 p3 =
  let cp = cross (p2--p1) (p3--p1) in
  if cp > 0 then LEFT else if cp < 0 then RIGHT else ON
```

---------------------End Ocaml Code-------------------------

3) Line segment intersection testing

We can use line side tests to answer this. Test that P1 and P2 are on opposite sides of the line (P3,P4), and that P3 and P4 are on opposite sides of the line (P1, P2).

-------------------------Ocaml Code-------------------------

```
let area (p,q) r = cross (q--p) (r--p)
let sign x = compare x 0
let segments_intersect (a,b) (c,d) =
  (sign (area (a,b) c)) * (sign (area (b,a) d)) > 0 &&
  (sign (area (c,d) b)) * (sign (area (d,c) a)) > 0
```

---------------------End Ocaml Code-------------------------

4) In-circle test
   Input: (P1,P2,P3,P4)
   Output: Determines the relationship of P4 to the circle thru (P1,P2,P3)

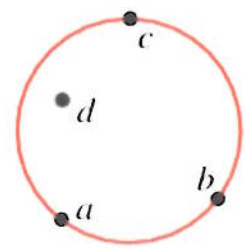
```
-------------------------Ocaml Code-------------------------
(* This returns 0 if the four points are on the same circle.
   I walk around the circle with my right hand on the circle
   from a --> b --> c.  It returns >0 if d is on the same side
   as my body, and <0 otherwise.  It's a fourth degree function
   in the given coordinates. See http://www.cs.cmu.edu/~quake/robust.html  *)

let incircle (ax,ay) (bx,by) (cx,cy) (dx,dy) =
  let det ((a,b,c),(d,e,f),(g,h,i)) =
    a**(e**i -- f**h) -- b**(d**i -- f**g) ++ c**(d**h -- e**g)
  in

  let row ax dx ay dy =
    let a = ax -- dx in
    let b = ay -- dy in
      (a, b, (sq a) ++ (sq b))
  in
    det (row ax dx ay dy, row bx dx by dy, row cx dx cy dy)
```

Incircle

Does d lie on, inside, or
or outside of abc?



$$\begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix} = \begin{vmatrix} a_x-d_x & a_y-d_y & (a_x-d_x)^2+(a_y-d_y)^2 \\ b_x-d_x & b_y-d_y & (b_x-d_x)^2+(b_y-d_y)^2 \\ c_x-d_x & c_y-d_y & (c_x-d_x)^2+(c_y-d_y)^2 \end{vmatrix}$$

The Convex Hull Problem

The sorting problem of CG.

A point set A subset R^d is convex if it is closed under convex combinations. That is, if we take any convex combination of any two points in A, the result is a point in A.

DEF: ConvexClosure(A) = CC(A) = smallest convex set containing A

There are two different definition of the convex hull. The first one is the one we will use. The second is used by some books.

DEF1: ConvexHull(A) = boundary of CC(A)

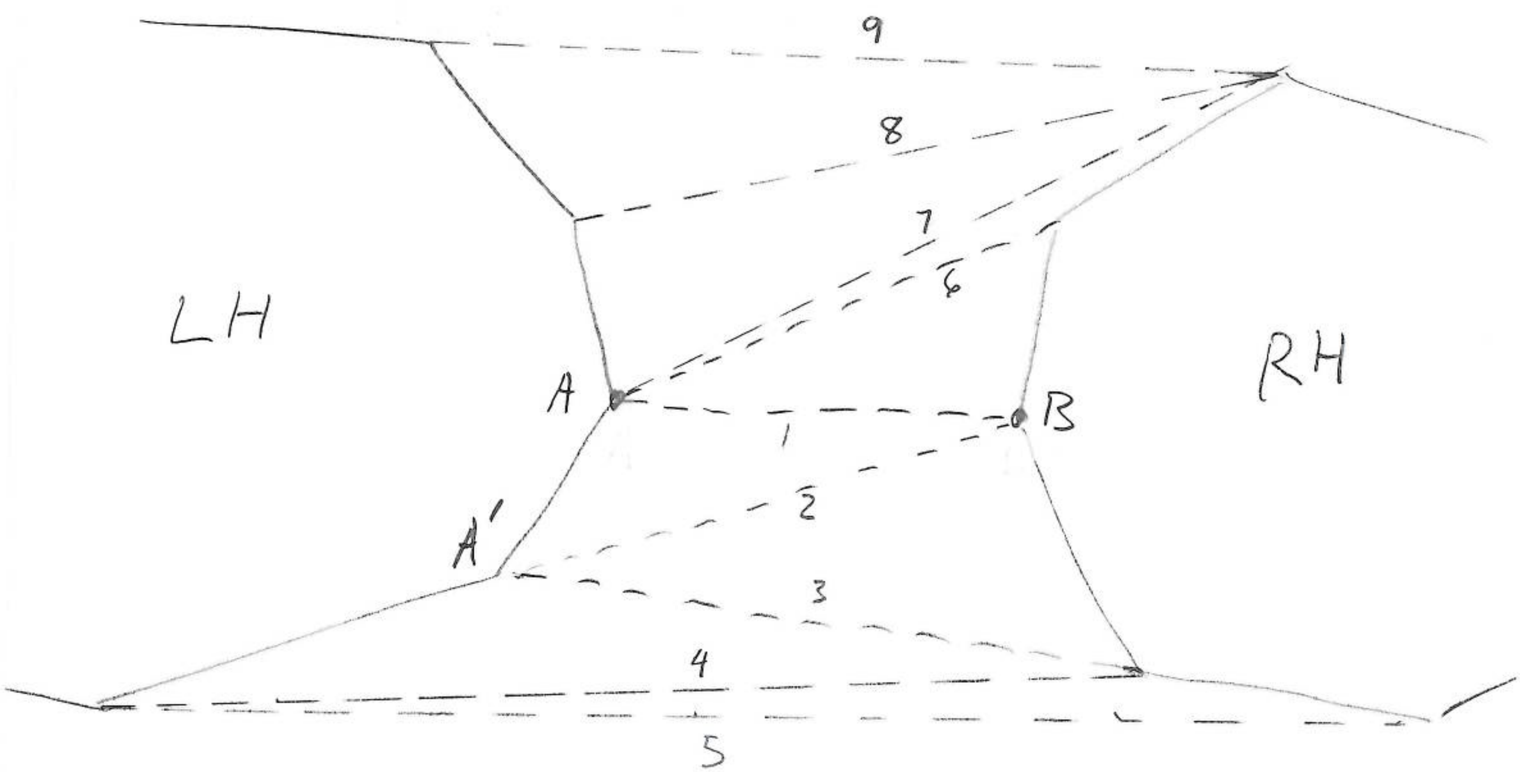
DEF2: ConvexHull(A) = CC(A)

A computer representation of a convex hull must include the combinatorial structure. In two dimensions, this just means a simple polygon in, say counter clockwise order. In many of our algorithms we'll need to be able to traverse the hull in either direction starting form a vertex or an edge. We'll also need to be able to add and delete points and edges. These operations are all easily done in constant time with the appropriate representation.

2D Convex Hulls by divide and conquer

Let the points be P1...Pn, where Pi = (xi, yi)
Preprocess the points by sorting all of them by their x coordinates. (This step happens just once. In all recrusive calls, we can assume the points are sorted.)

Now recursively compute the convex hull of {P1...Pn/2}, call it LH, and and {Pn/2+1...Pn} call it RH.



Now the problem is to stitch the left and right halves together to form a hull for all the points. Here's how to do this:

Let A be the rightmost point in LH and B be the leftmost point in RH. Start with the line segment [A,B]. Let A' be the next point clockwise around the LH from A. Consider the three points [B,A,A'] if this is a left turn then A = A' and continue. If it's a right turn, then stop. Now do the analogous walk around RH starting from B. If at some point neither walk can proceed, the segment [A,B] is the new segment to be added to the CH of all the points on the bottom.

An analogous algorithm suffices to compute the new edge to be added on top of the hull. Once these new edges are found, the new convex hull is stitched together form the relevant parts.

Preprocessing takes O(n log n) to sort. After that it's a standard divide and conquer algorithm with the following recurrence:

T(n) = 2T(n/2) + n

This solves to O(n log n).

Graham Scan for 2D convex hulls

Here's another algorithm. The input points are P1, P2, ... Pn. Find the point with the smallest Y coordinate and swap it with P1. Now for each point Pi, compute the angle of the ray P1-->Pi, and sort the points by this angle. (These angles are all between 0 and pi.)

Start with segment P1--->P2. In general there will be a sequence of segments, starting from P1, P2 .... Each turn in this sequence is a left turn. Our goal is to process a new point, and add it onto this sequence.

Here's the algorithm. Add the new point Pi to the end of this sequence. If the sequence of segments now has a right turn at the end, then delete the second to last point of this sequence. Repeat this process until the sequence is restored to one that has only left turns. The direction of a turn can be done using a line side test.

After all the points are processed in this way, we can just add the last segment from Pn to P1, to close the polygon, which will be the convex hull.

Each point can be added at most once to the sequence of vertices, and each point can be removed at most once. Thus the running time of the scan is O(n). But remember we already paid O(n lg n) for sorting the points at the beginning of the algorithm, which makes the overall running time of the algorithm be O(n lg n).

(Alternatively, keep a token on each edge of the current edge sequence. Use the token to pay for a segment's removal. Adding a new segment is one unit of work, plus another token to keep on it.)

Complete ocaml code for the graham scan is at the end of these notes.

Lower bound for compute the convex hull

Suppose the input to a sorting problem are the number X1, ..., Xn

Consider the convex hull problem:

$(X_1, X_1^2), \ldots, (X_n, X_n^2)$

The convex hull of these points must return them in sorted order.

Thus:

Any comparison based CH algorithm must make Omega(n log n) comparisons.

QuickSort and Backwards Analysis

Before we present and analyze a simple randomized CH algorithm, let's go back and look at another way to analyze QuickSort called Backwards Analysis.

We assume that the keys are distinct

Recall randomized QuickSort:

```
QS(M)
    1) pick random b in M
    2) split M into S and L with S < b < L
    3) return [QS(S), b, QS(L)]
```

The cost of the call QS(M) (Not counting recursive calls) is the size of the array M minus 1. (This is how many comparisons are done.)

To do the backwards analysis we propose a simple game with a cost function that has the same behavior.

Consider a dart board with n empty squares.

```
+------------------------+
|  |  |  |  |  |  |  |  |
+------------------------+
```

All n squares start out empty.
The dart game proceeds as follows:

```
While  there exists a empty square do:
     Pick an empty square at random.  Put a dart into it.  The cost
     of the step is the number empty squares to the left and right
     of the new dart.
```

The expected cost of this game is exactly the same as the expected cost of randomized quicksort. Why? Let's prove this by induction. In the base case, n=0 or n=1. In this case QS(M) costs zero and the dart game costs zero. In the general case, the cost of the first dart is n-1, and the cost of the top level call to QS(M) is just n-1 (not counting the costs of the recursive calls). We know that the dart game eventually fills every cell with a dart. And that the darts to the right of the first dart have no influence on what happens to the left of it. So we can rearrange the sequence of darts to put those that are in the left half first, then those in the right half. This does not change the cost of the dart game. Now by induction the cost of the dart game in the left half is the same as quicksorting the left half. The right half is the same.

The dart game can be played backwards. Here's how to do it.

```
Layout a dart board with n squares each with a dart.
While  there exists a dart on the board do:
     Randomly pick a dart.  The cost is the number of empty squares
     to the left and right of the new dart.  Remove the dart.
```

The only random parameter in the forward dart game is the permutation that is chosen to add the darts. Similarly the only random parameter in the reverse dart game is the permutation that is chosen to remove the darts. Thus if we made a movie of the dart game and watched it backwards, it would be totally indistinguisable (drawn from the same distribution) from a movie of the backwards dart game.

Thus, if we can analyze the expected cost of the backwards dart game, it will tell us the expected cost of the forward dart game (and thus quicksort).

Let's analyze the backwards dart game.

Suppose there are i darts left. Define Ti as follows:

T_i = Expected cost to remove a random dart from a board of i darts.

One way of computing this cost is to take the total cost of each of the possibilities (the dart to remove) and divide by the number of darts (i). The total cost of all possibilities can be computed by allocating the costs to the empty cells. So when considering the option of removing a dart d, put a token into each of the empty cells in the blocks of empty cells to the left and right of d. The number of tokens distributed is the cost of the option of removing dart d. The sum of these costs over all darts is the number of tokens on all the empty cells. It's easy to see that each empty cell can get at most two tokens, one from each of the two darts at the boundaries of that block of empty cells.

Total cost of all options at step i <= 2*(# empty cells) = 2(n-i)

T_i = Average cost of all options at step i <= 2(n-i)/i = 2n/i - 2

Let E_n be the expected cost of the backwards dart game on a board of size n.

$$E_n = T_n + T_{n-1} + \ldots T_1$$

$$= \left[2n \sum_{i=1\ldots n} 1/i\right] - 2n$$

$$= 2nH_n - 2n = O(n \log n)$$

This is the same as the cost of the dart game running in the forward direction, and the expected cost of quicksort.

```
(* Ocaml code for the Graham Scan convex hull algorithm *)

let (--) (x1,y1) (x2,y2) = (x1-x2, y1-y2)
let (++) (x1,y1) (x2,y2) = (x1+x2, y1+y2)
let cross (x1,y1) (x2,y2) = (x1*y2) - (y1*x2)
let dot (x1,y1) (x2,y2) = (x1*x2) + (y1*y2)

type side = LEFT | ON | RIGHT

let line_side_test p1 p2 p3 =
  let cp = cross (p2--p1) (p3--p1) in
  if cp > 0 then LEFT else if cp < 0 then RIGHT else ON

let len (x,y) =
  let sq a = a*.a in
  let (x,y) = (float x, float y) in
  sqrt ((sq x) +. (sq y))

let graham_convex_hull points =
  let inf = max_int in
  let base = List.fold_left min (inf,inf) points in

  let points = List.sort (
    fun pi pj ->
      if pi=pj then 0
      else if pi=base then 1
      else if pj=base then -1
      else
        match line_side_test base pi pj with
          | ON -> 0
          | LEFT -> -1
          | RIGHT -> 1
  ) points in

  let rec scan chain points =
    let (c1,c2,chainx) = match chain with
      | c1::((c2::_) as chainx) -> (c1,c2,chainx)
      | _ -> failwith "chain must have length at least 2"
    in
    match points with [] -> chain
      | pt::tail ->
        match line_side_test c2 c1 pt with
          | ON ->
            if len (pt--c2) > len (c1--c2)
            then scan (pt::chainx) tail
            else scan chain tail
          | LEFT -> scan (pt::chain) tail
          | RIGHT -> scan chainx points
  in

  match points with
    | (p1::((_::(_::_)) as rest)) -> List.tl(scan [p1;base] rest);
    | _ -> points

let print_list l =
  List.iter (fun (x,y) -> Printf.printf "(%d,%d) " x y) l;
  print_newline()

let () = print_list (graham_convex_hull [(0,0);(0,2);(2,2);(2,0);(1,1)])
```

Running the program:

```
$ ocamlc graham.ml
$ ./a.out
(0,2) (2,2) (2,0) (0,0)
```