ArsDigita University
Month 5: Algorithms - Professor Shai Simonson

Problem Set 4 – Applications of Geometric Algorithms

Geometric Classes and the Convex Hull Problem

Geometric algorithms are wonderful examples for programming, because they are deceptively easy to do with your eyes, yet much harder to implement for a machine.

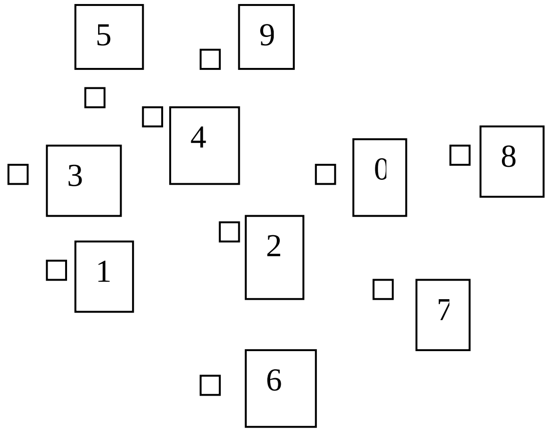
We will concentrate on a particular problem called convex hull, which takes a set of points in the plane as its input and outputs their convex hull. We will stay away from formal definitions and proofs here, since the intuitive approach will be clearer and will not lead you astray. To understand what a *convex hull* is, imagine that a nail is hammered in at each point in the given set, the convex hull contains exactly those points that would be touched by a rubber band which was pulled around all the nails and let go. The algorithm is used as a way to get the natural border of a set of points, and is useful in all sorts of other problems.

Convex Hull is the sorting of geometric algorithms. It is fundamental, and as there are many methods for sorting, each of which illustrates a new technique, so it is for convex hull.

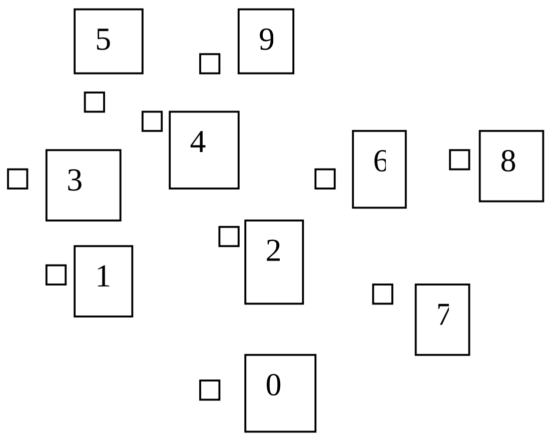
Graham Scan

The particular algorithm we will implement for Convex Hull is due to Ron Graham and was discovered in 1972. Graham Scan, as it is called, works by picking the lowest point p, i.e. the one with the minimum p.y value (note this must be on the convex hull), and then scanning the rest of the points in counterclockwise order with respect to p. As this scanning is done, the points that should remain on the convex hull, are kept, and the rest are discarded leaving only the points in the convex hull at the end.

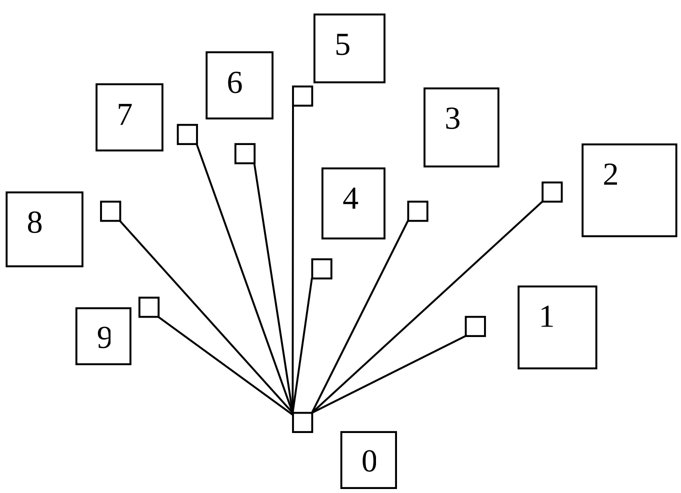
To see how this is done, imagine first that, by luck, all the points scanned are actually on the convex hull. In that case, every time we move to a new point we make a left turn with respect to the line determined by the last two points on the hull. Therefore, what Graham Scan does, is to check if the next point is really a left turn. If it is NOT a left turn, then it backtracks to the pair of points from which the turn would be a left turn, and discards all the points that it backs up over. Because of the backtracking, we implement the algorithm with a stack of points. An example is worth a thousand words. The input list of points is: (A, 0, 0) (B, -5, -2) (C, -2, -1) (D, -6, 0) (E, -3.5, 1) (F, -4.5, 1.5)
(G, -2.5, -5) (H, 1, -2.5) (I, 2.5, .5) (J, -2.2, 2.2).



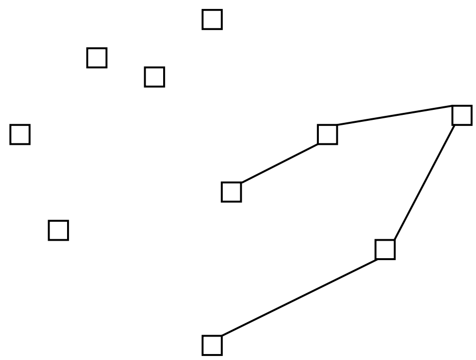
The array of input points is shown above labeled by index in the array (rather than their char label). The point labeled A is in index 0, B is in index 1, etc. The lowest point is computed and swapped with the point in index 0 of the array, as shown below.



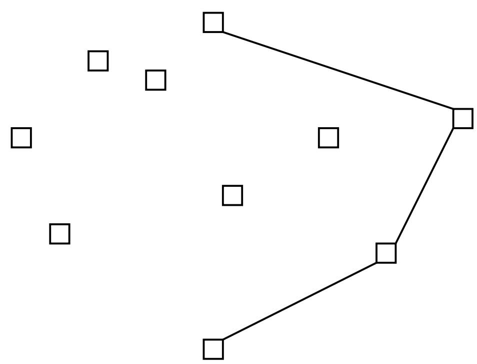
The points are then sorted by their polar angles with respect to the lowest point.



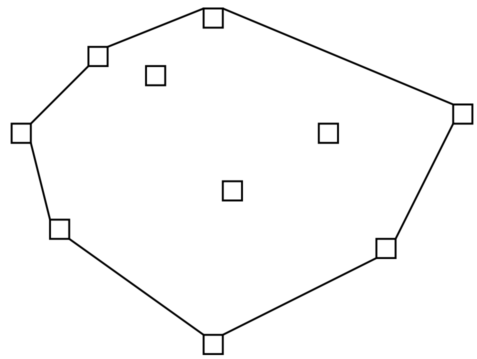
The points are sorted and rearranged in the array as shown above. The turn from line 0-1 to point 2 is left, from 1-2 to 3 is left, from 2-3 to 4 is left. Now the stack contains the points 01234. This represents the partial hull in the figure below.



The turn from line 3-4 to point 5 is right, so we pop the stack. The turn from 2-3 to 5 is right, so we pop again. The turn from 1-2 to 5 is left, so we push 5 on the stack. The stack now has 0125, and the picture looks like this:



The turn from line 2-5 to 6 is left so 6 is pushed on the stack. Then the turn from 5-6 to 7 is right, so 6 is popped and 7 is pushed because the turn from line 2-5 to 7 is left. The rest of the turns are left, so 8 and 9 are pushed on the stack. The final stack is 0125789, and the convex hull is shown below:



Graham Scan Pseudo-code: The algorithm takes an array of points and returns an array of points representing the convex hull.

1. Find the lowest point p, (the point with the minimum y coordinate). If there is more than one point with the minimum y coordinate, then use the leftmost one.
2. Sort the remaining points in counterclockwise order around p. If any points have the same angle with respect to p, then sort them by increasing distance from p.
3. Push the first 3 points on the stack.
4. For each remaining point c in sorted order, do the following:
   - b = the point on top of the stack.
   - a = the point below that on the stack.
   - While a left turn is NOT made while moving from a to b to c do
     - pop the stack.
     - b = the point on top of the stack.
     - a = the point below that on the stack.
   - Push c on the stack.
5. Return the contents of the stack.

Implementation Details for the Point Class:

Private Data:

We start by defining a simple geometric class *point* and deciding on the appropriate private data and member functions. A *point* should have three fields: two are float for the *x* and *y* coordinates, and one is a char for the name of the point.

Constructors:

A three parameter constructor should be created to set up points.

Methods:
An output method to print out a point by printing its name (char) along with its coordinates.

Accessor methods to extract the *x* or *y* coordinates of a point.

A static distance method to determine the distance from one point to another.

A *turn-orientation* method that takes two points *b* and *c* and returns whether the *sweeping movement* from the line a-b to the line a-c goes clockwise (1), counterclockwise (-1) or neither (0). (The result is neither (0) when a, b and c are all on the same line.) This function is necessary for deciding whether a left or right turn is made when moving from *a* to *b* to *c* in step 4 of the pseudo-code above. It is also useful for sorting points by their polar angles.

It may not be obvious how to implement this function. One method is based on the idea of the cross product of two vectors. Let *a*, *b* and *c* be points, where *x* and *y* are accessor methods to extract the *x* and *y* coordinates respectively.

if $(c.x – a.x)(b.y – a.y) > (c.y – a.y)(b.x – a.x)$ then the movement from line *a-b* to line *a-c* is clockwise.
if $(c.x – a.x)(b.y – a.y) < (c.y – a.y)(b.x – a.x)$ then the movement from line *a-b* to line *a-c* is counterclockwise.
Otherwise the three points are co-linear.

To understand this intuitively, concentrate on the case where the lines a-b and a-c both have positive slope. A clockwise motion corresponds to the line a-b having a steeper (greater) slope than line a-c. This means that $(b.y – a.y)/(b.x – a.x) > (c.y – a.y)/(c.x – a.x)$. Multiply this inequality by $(c.x – a.x)(b.x – a.x)$ and we get the inequalities above.

The reason for doing the multiplication and thereby using this *cross product* is:
1. To avoid having to check for division by zero, and
2. So that the inequality works consistently for the cases where both slopes are not necessarily positive. (You can check for yourself that this is true).

Graham Scan should be coded using an abstract STACK class of points. The sorting in step two can be done by comparing pairs of points via the turn-orientation method with respect to the lowest point (object *p*). An *interface* (if you use Java) may be convenient to allow the sorting of points.

A Note on Complexity:

The complexity of Graham Scan is $O(n \log n)$. We will discuss informally what this means and why it is true. It means that the number of steps in the algorithm is bounded asymptotically by a constant times n log n where n is the number of points in the input set. It is true because the most costly step is the sorting in step 2. This is $O(n \log n)$.

Step 1 takes time *O(n)*. Step 3 takes *O(1)*. Step 4 is trickier to analyze. It is important to notice that although each of the *O(n)* points are processed, and each might in the worst case have to pop the stack *O(n)* times, overall this does NOT result in $O(n^2)$. This is because overall, every point is added to the stack exactly once and is removed at most once. So the sum of all the stack operations is *O(n).*

There are many *O(n log n)* and $O(n^2)$ algorithms for the convex hull problem, just as there are both for sorting. For the convex hull there is also an algorithm that runs in *O(nh),* where *n* is the number of points in the set, and h is the number of points in the convex hull. For small convex hulls (smaller than *log n*) this algorithm is faster than *n log n*, and for large convex hulls it is slower.

Jarvis' Algorithm for Convex Hull

Jarvis' algorithm uses some of the same ideas as we saw in Graham Scan but it is a lot simpler. It does no backtracking and therefore does NOT need to use a STACK class, although it still makes use of the ARRAY class template with your point class.

As before, we start by adding the lowest point to the convex hull. Then we repeatedly add the point whose polar angle from the previous point is the minimum. This minimum angle computation can be done using the clockwise/counterclockwise member function, similar to how the sorting step (step 2) of Graham Scan uses the function.

The complexity of this method is *O(nh)* where *h* is the number of points in the convex hull, because in the worst case we must examine *O(n)* points to determine the minimum polar angle for each point in the hull.

Problems

1. Convex Hull Algorithms

a. Write code to define the point class.
b. Implement the Graham Scan algorithm.
c. Implement Jarvis' algorithm.
d. Test your algorithms on the data from the example above.
e. **Optional:** Do it all graphically.

2. An Old Classic Video Game

Do the Ghostbuster problem (35-3 on page 914) of your text, and prove that your algorithms run in the time complexity requested.

3. A Line Segment Class

a. Write code to define a *line segment* class. The methods should include a static method testing whether or not two lines intersect.
b. **Optional:** Use this class to write a program that given a list of line segments, determines the largest number of segments that do not mutually intersect. What is the time complexity of your algorithm? Do you think this problem can be solved in polynomial time or is it NP-Complete?