Convex Hulls Algorithms - part II

1 Incremental Approach

Assume that the given points are in general position, i.e. no three are collinear and no four are cocircular. This approach works as follows:

1. Sort all points on the x-coordinate.
2. Let the first three points define a triangle which is in fact their convex hull. Call it C_3.
3. for $i = 4$ to n
   find the pair of bridges from p_i to the convex hull C_{i-1}.
   form C_i.

In the above algorithm, a *bridge* is an edge from the current point under consideration to the previously formed hull. Since the current point has the maximum x-coordinate (so far), it has to be on the hull, so two new edges have to be introduced and some of the edges previously on the hull would have to be removed. During the course of the algorithm, each point p_i is the rightmost endpoint of at most two edges. Hence the total number of edges created is $\leq 2n$. So steps 2 and 3 above take $O(n)$ time overall. Step 1 requires $n \log n$, time and hence the algorithm runs in $n \log n$. This algorithm also extends to higher dimensions because the same technique of sorting along one coordinate can be used there. We discuss the higher dimension algorithm later in the course.

Note that in this problem, if the input points all had integer coordinates, then step 1 could be accomplished in $\mathcal{O}(N \frac{\log N}{\log \log N})$ using the Fredman-Willard result if bitwise Boolean calculations are allowed. Thus the entire algorithm would have the same complexity.

2 A divide and conquer approach

Some of the above methods are optimal but they do not really give us more information than the convex hull. They are better for static cases when the set of points is known in advance and the only objective is to find the convex hull of that set. In dynamic cases, it may be necessary to add/delete points over time in which case we hope to modify the hull quickly rather than start from scratch everytime. A divide and conquer approach which helps to retrieve information quickly works as follows:

1. Sort points on x-coordinate. Let them be $v_1, \ldots, v_n$.
2. form convexhull(1,$\frac{n}{2}$,A).
3. form convexhull($\frac{n}{2}+1$,n,B).
4. merge(A,B,C).

The merge process is carried out by finding the bridges for the upper convex hulls as well as the bottom convex hulls of both halves. Given bottom convex hulls of both sides, the bottom bridge can be found in $\mathcal{O}(n)$, or even $\mathcal{O}(\log n)$, time (done by looking at the kind of angles formed by the current bridge under consideration and the edges on the given bottom convex hulls; 8 different cases are possible for each prospective bridge). So the recurrence in this case is either

$$T(n) = 2T(\tfrac{n}{2}) + O(n),$$
or
$$T(n) = 2T(\tfrac{n}{2}) + O(\log n).$$

The solution to both of the above recurrences is $T(n) = n \log n$, but the second method is better in case of dynamic updates. To facilitate dynamic updates, the hulls at each stage are maintained at all times. The partial hulls have pointers to them. All the hulls are maintained as concatenable queues so as to be able to perform splits and joins in $\log n$, time. When the convex hull of the points is being constructed, the divide and conquer tree is formed and each node in the tree corresponds to a partial hull at some step during the recursive construction. Rather than store the complete hull at each stage, part of the hull which is removed from the hull during a merge step is stored and pointers are maintained to the position where the hull was

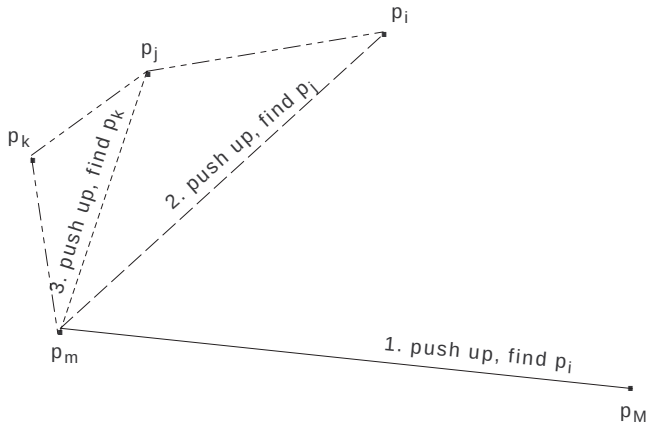

Figure 1: Quickhull

split. To insert or delete a new point, begin at the root, split the current bc-hull and form hulls of the two children and see which of them would contain the inserted/deleted point and move down the tree accordingly. Do the same at the interior nodes as well. This descent requires $O(\log^2(n))$, time. While going back up, which is the real updation process, only a constant number of merges are performed at each node (each merge is only $\log n$), so the entire ascent requires $\mathcal{O}(\log^2 n)$ time. Consequently, the bc-hull of a set of n points can be maintained at a cost of $O(\log^2(n))$, per insertion/deletion.

3 Quickhull

[1]

The QuickHull algorithm tries to discard as many points as possible which are definitely interior to the hull, and then tries to concentrate on the ones that are closer to the hull boundary.

1. Find the point p_m with the smallest x-coordinate (in case of a tie, choose one with largest y-coordinate)

2. Find the point p_M with the largest x-coordinate (in case of a tie, choose one with largest y-coordinate)

[1]Scribe: Marko Bukovac, Spring 03

As shown above, we find two extreme points p_m and p_M in the set. We then need to find the third extreme point p_i, which is the point which is furthest away from the line $p_m p_M$ - it is extreme in the direction *orthogonal* to line $p_m p_M$ and as such lies on the hull boundary. At this point we can discard all the points in the triangle $\triangle p_m p_i p_M$. We can then repeat the procedure for all the points right of $p_m p_i$ and points right of $p_i p_M$.

QUICKHULL(p_m, p_M, S)

1. if $S = \emptyset$ return ()
2. else

   $p_i \leftarrow$ index of point with max distance from $p_m p_M$

   $A \leftarrow$ points strictly right of $p_m p_i$

   $B \leftarrow$ points strictly right of $p_i p_M$

   return QUICKHULL(p_m, p_i, A) + (p_i)+ QUICKHULL(p_i, p_M, B)

The analysis of Quickhull is similar to the analysis of Quicksort. In the best case, if we always have a balanced partition, we get that $T(n) = 2T(\frac{n}{2}) + O(n)$, whose solution is $T(n) = O(n \log n)$. However, in the worst case, the partitioning may not be balanced and we can get a lopsided tree. In this case, just like Quicksort, the running time of the algorithm is $T(n) = O(n^2)$.