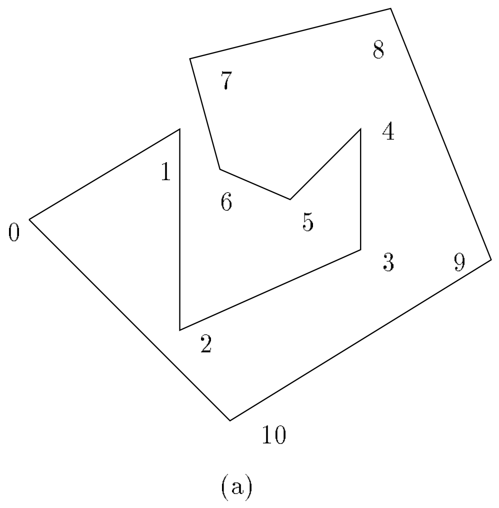
Spring 1996 **CMSC 754: Solution to Homework 4** Samir Khuller

Due in class: April 4, 1996.

(1) The algorithm fails for the polygon shown in Fig. 1(a). We start with the stack containing $(0, 1)$. The next turn is a right turn and the stack becomes $(0, 1, 2)$. When we process point 3, we eject 2 from the stack, which becomes $(0, 1, 3)$. The next turn is a left turn and the stack becomes $(0, 1, 4)$. We next process point 5. The next turn is a right turn, and the stack becomes $(0, 1, 4, 5)$. After that, all turns are right turns, and we terminate with the stack being $(0, 1, 4, 5, 6, 7, 8, 9, 10)$. This is clearly not the convex hull of the polygon.

In general, the algorithm fails when we encounter a region of the form as shown in Fig. 1(b). When we process point j, the stack has $(\ldots, i-1, i, j)$. When we move to point $j+1$, we treat it as a right turn. After that if we only have right turns, we obtain an incorrect convex hull. It seems that the algorithm works for all "weakly externally visible" polygons, these are polygons with the property that each vertex has visibility to a point at infinity.



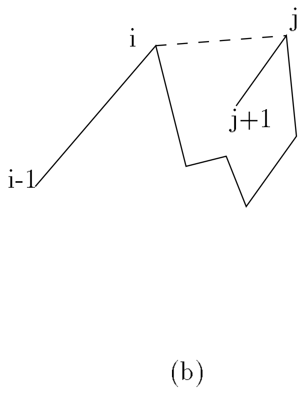


Figure 1: Counter-example for convex hull algorithm.

(2) Assume that the points do not have the same x co-ordinate, and have been sorted left to right. Assume that the hull for the first $i-1$ points is CH_{i-1}. Since p_{i-1} is the rightmost point, it must be on the hull. We now add point p_i. To add this point, we need to delete some edges from CH_{i-1} and add the two tangents from p_i to CH_{i-1}. This is done in the same manner as in the incremental algorithm on page 99. However, we start our search for the supporting tangent lines at p_{i-1}.

At each step, the work done is proportional to the number of edges deleted from the hull. To argue this formally, we define a potential function that is equal to the number of edges on the hull. When $i = 4$, the hull is a triangle and has a potential function value of 3. At the i^{th} step, if we delete d edges from CH_{i-1}, the time spent is d and the increase in potential is $2-d$. (If $d = 1$ the potential increases, if $d = 2$ it is unchanged, otherwise it decreases.) Thus the amortized time is constant.

(3) One can view the assignment problem as a matching problem on the line. Given n red points on the line, and n blue points, find a min weight matching of the red and blue points. (Each red point is to be paired with exactly one blue point. The cost of the pair is simple the difference in their x values. The minimum weight matching is the one that has minimum total weight.)

The matching algorithm is trivial. Simply sort all the ski's in increasing length and all the skier's in increasing height. This takes $O(n \log n)$ steps (using MERGESORT for example). Now assign the i^{th} skier to the i^{th} ski. We can prove that this yields the optimal assignment of skis to skiers.

Consider an optimal solution in which the skis are assigned out of order. In other words, $i < j$ and $p < q$ and the i^{th} skier has the q^{th} ski, with the j^{th} skier having the p^{th} ski. The claim is that swapping the assignment cannot increase the objective function value. In the event $p < q < i < j$ or $i < j < p < q$ the sum of the absolute differences between the height of the skier and the length of the assigned ski does not change. In all other cases, the sum of the differences in height of the skier and length of the ski actually decreases. In either case, we obtain a solution of cost no more than the optimal solution in which fewer pairs are "out of order". By repeating this we can transform this into a solution with the same cost, and that has the structure that we are interested in. If the ski's are never "out of order", since there are exactly the same number of skis as skiers, the assignment has to be the same as that produced by the algorithm. This completes the proof.

(Notice that this scheme does not work when there are more skis than skiers.) However, one can still obtain an $O(n \log n)$ algorithm.

(4) (a) For this part it is easier to consider the operations on the dual of the Voronoi Diagram, the Delaunay Triangulation. Observe that removing a point y from a Delaunay triangulation will only affect those points that are adjacent to y in the Delaunay graph. To see this, suppose that there is a point $q \in S$ such that q is not adjacent to y and its Delaunay neighbors change with the removal of y. For its Delaunay neighbors to change, it must be the case that one of the triangles incident upon q must not pass the in–circle condition (this is the Delaunay triangle characterization). But how could this be? We are removing points, not adding them. Thus, we need only to retriangulate those points that were adjacent to y (and make sure that we add no duplicate edges back into the triangulation. If there are k such points, it takes $O(k \log k)$ time. It is now straightforward to update the Voronoi Diagram given the change in the triangulation.

(b) Fig. 2 shows how we insert a new point into the voronoi diagram. Let us assume that the new point y lies in $V(p)$. Draw the bisector of p and y, and compute its intersection with $V(p)$. This is done as follows. We first compute the edge of $V(p)$ that the ray shot from p towards y would hit. This can be done by a binary search in $O(\log n)$ time since $V(p)$ is convex. Once we compute this edge, we can walk clockwise and anti-clockwise from the edge $V(p)$ to compute where the py bisector exits $V(p)$ (assume at points u and w). Since u and w are equidistant to three points they form voronoi vertices. All points on the edges of $V(p)$ that are closer to y than to p will be inside $V(y)$. The work done can thus be charged to the complexity of $V(y)$.

Notice that none of the voronoi vertices on the other side of the py bisector are affected, since their three closest points do not change. However, the voronoi vertices that are closer to y than to p all need to be updated (see Fig.3). It is clear that from u we have to draw a new voronoi edge using the bisector of $\overline{ry}$ When this edge hits the voronoi edge separating r and z, the distance to the points r, z and y is the same. Moreover, these are the three closest points. A new voronoi vertex is created here. Similarly, we can process all these points. (One way to view the process is to see which delaunay triangles are not point-free once we introduce the point y. The delaunay triangles that are destroyed are $\{r, p, z\}$ and $\{p, z, x\}$. These are replaced by four new delauanay triangles.

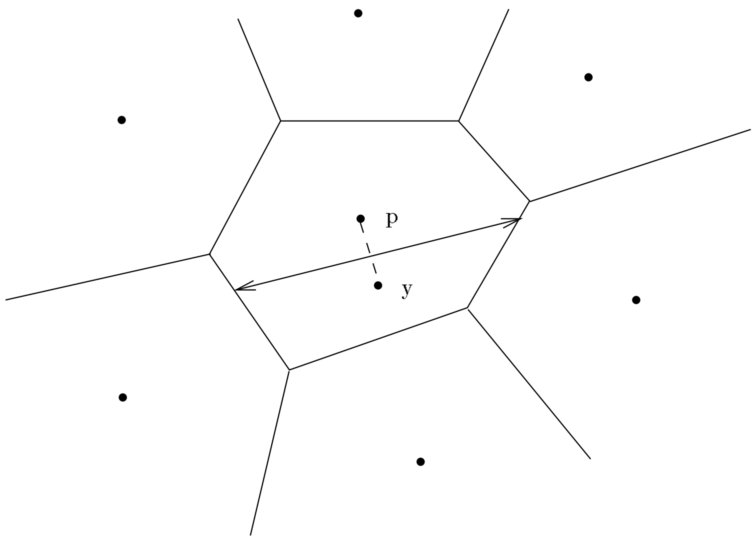


Figure 2: Insertion of points in the Voronoi Diagram.

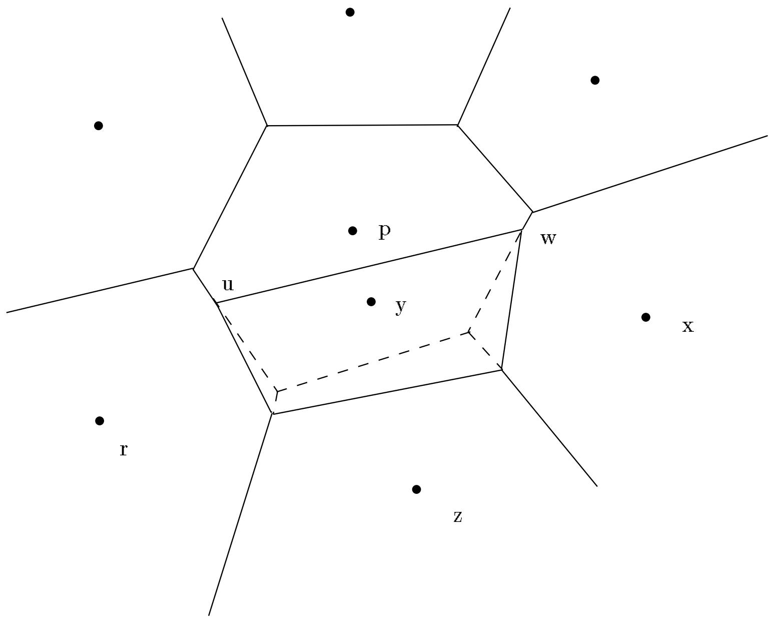


Figure 3: Creating $V(y)$.