Spring 1996 Samir Khuller

CMSC 754
Solutions to Homework 2

(1) There are probably several algorithms that would work. Here is one simple scheme: first sort all the points lexicographically on (x, y) the co-ordinates of the vertices. (Points are sorted left to right, and we break ties by putting the lower one first.) Let p_1 be the leftmost point and p_n the rightmost. Break the points into two groups, the ones "above" the line segment $\overline{p_1 p_n}$ and the points "below" the line segment. (This can easily be done by a simple calculation as describe in Chap 1 of the text.) Let S_A be the set of points strictly above the line and S_B be the set of points strictly below the line. (Points on the line can all be taken in S_A so long as S_B is not empty. If S_B is empty, we take all the points on the line in S_B. If S_A and S_B are empty, all the points are on the line and no simple polygon exists.)

We now find a "chain" from p_1 to p_n that has all the points in S_A on its boundary. (Using a similar procedure a chain connecting the points in S_B can be found.) It is easy to extract the points in S_A in sorted order from the sorted list of all n points. Now obtaining this chain is very easy, we simply connect the points of S_A in order from p_1 to p_n to obtain the "upper" chain. Connecting the two chains yields a simple polygon. Notice that it is important to handle the points with the same x co-ordinate carefully, and thats why we needed the points in lexicographic order.

(2) Let S be the set of intersection points. There are at most $2n$ points that could lie on the convex hull of S. Each line has at most $n-1$ intersection points on it, and only the extreme ones are candidates for being on the hull. Call this subset of points S'. If we can identify the points in S' in $O(n \log n)$ time we can then compute their convex hull in $O(n \log n)$ time. Can this be done?

Rather than trying to identify all such points, we will identify the following subset of points: sort all the lines in order of slope. Let the lines be $\ell_0, \ell_1, \ldots, \ell_{n-1}$. Let p_i be the intersection of ℓ_i and $\ell_{(i+1) \bmod n}$. This is a collection of n points (say set P). We will prove that CH(P)=CH(S). One can apply any standard convex hull algorithm on set P.

Consider a point p on the boundary of the convex hull of S. Let this point be an intersection of ℓ_i and ℓ_j. Moreover, if many lines intersect at p, we take a pair whose slopes are "closest" to each other. Consider a supporting line ℓ, of the convex hull, at point p. We need to argue that the lines ℓ_i and ℓ_j are adjacent in the sorted order (by slope) and thus $p \in P$. All points of S are "below" line ℓ, hence any other line ℓ_k has both its intersections with ℓ_i and ℓ_j "below" ℓ. Thus it cannot have a slope "in-between" ℓ_i and ℓ_j. (It could be that ℓ_k's intersection with ℓ_i and ℓ_j is also at point p, in which case if it lies "in-between" ℓ_i and ℓ_j that contradicts our assumption on the choice of the pair of lines.)

(3) [This answer is based on material found in Guibas et. al "Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons" Algorithmica 1987]

Observe that the shortest path from s to t is a poly-line with turning points occurring only at the vertices of P. We compute this shortest path, π, as follows. First, triangulate the polygon. (This takes $O(n)$ time if we use Chazelle's algorithm; however, nobody, not even a theorist, would want to do such a thing. In practice, the $O(n \log n)$ algorithm is used.) Next, locate the two triangles that contain the source point s and the target point t. (This too can be done in $O(n)$ time.) Now do a breadth first search on the dual of the triangulation to locate the sequence of triangles through which π passes. We are interested in the diagonals of these triangles because all candidate turning points of π correspond to the vertices of these diagonals. (These sequences again take $O(n)$ time to compute).

We maintain two lists of vertices, R_1 and R_2 which we can view as *ropes* anchored at a point and winding about the polygon. Starting with the first diagonal uw that we will cross, set $R_1 = \{s, u\}$ (meaning that we have a rope anchored at s and stretched to u) and set and $R_2 = \{s, w\}$. Now, execute this (tail) recursive procedure, with $a = s$:

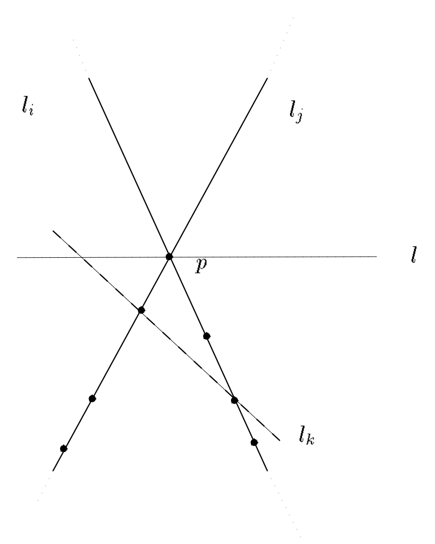

Figure 1: Pair of adjacent lines in slope order.

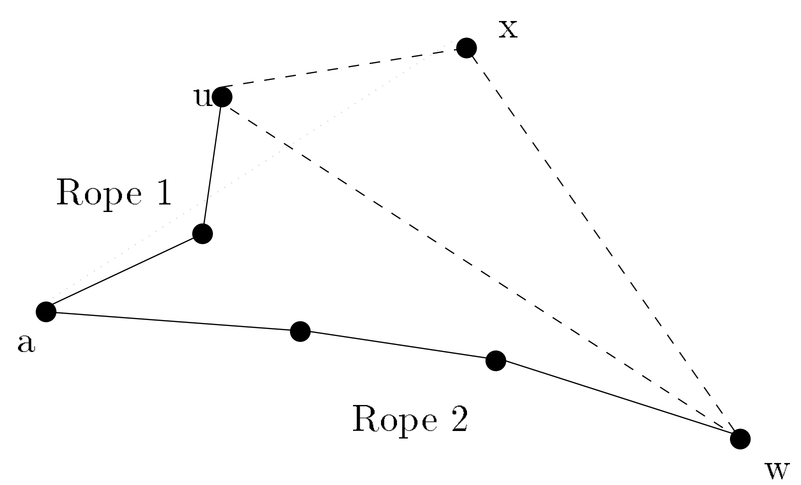

Figure 2: Figure for case (b). Use the slope of segment ax to find the shortest path from a to x.

SPP(a,R_1,R_2,)

(a) Let u be the last element in R_1. Let w be the last element in R_2. (We maintain the invariant that the line segment uw will always be a diagonal of the triangulation that π crosses). Consider the two triangles sharing diagonal uw. One of them is on the same side of the polygon as s and the other is on the same side of the polygon as t. Call the latter triangle uwy. If triangle uwy contains t, then when we refer to x below, we are speaking about t, otherwise we are speaking about the vertex y.

(b) We wish to find the shortest path in the polygon from a to x. This path will follow one of the ropes (which always follow the boundary of the polygon except when anchored at s) until we find a vertex v that is visible to x, and will follow the straight line from v to x. In other words, we must search both ropes R_1 and R_2 for the vertex v such that vx is contained in (or on the border of) P and that the shortest path from a to v plus the length of vx is minimal. Note that if the line segment ax does not intersect P, then $v = a$. Otherwise, we can compute the slope of the line passing from a to x and use this as the basis of our search. Details are left to the reader. Notice that in doing the search, the time we spend can be "charged" to segments of the "ropes" that are deleted (see Fig. 2).

(c) Set $\pi(s, x) = \pi(s, v) \cup vx$. We can maintain π in linear space and time by judicious use of pointers.

(d) If $x = t$, then we are done. Otherwise there are 4 cases to consider: whether ux, or wx is the next diagonal for both of those cases, which of the two ropes contained the vertex v. The last two cases are symmetric to the first two.

   i. If the line segment ux is the next diagonal to be crossed as we follow the diagonals to the triangle containing t (we know this from the BFS in the preprocessing section) and v was found in R_1 then set $R_1 = \{v, \ldots u\}$ (that is "re–anchor" R_1 at v and stretch it along the shortest path from v to u (which can be obtained by walking along the polygon) and set $R_2 = v, x$. Now call SPP(v, R_1, R_2) recursively.

   ii. If the line segment ux is the next diagonal to be crossed, and v was found in R_2, then set $R_1 = \{a, \ldots u\}$ and set $R_2 = \{a \ldots v, x\}$. Now call SPP(a, R_1, R_2) recursively.

   iii. If the line segment wx is the next diagonal to be crossed, and v was found in R_1 then set $R_1 = a, \ldots v, x$ and set $R_2 = \{a, \ldots, w\}$. Now call SPP(a, R_1, R_2) recursively.

   iv. If the line segment wx is the next diagonal to be crossed, and v was found in R_2, then set $R_1 = \{v, x\}$ and set $R_2 = \{v, \ldots w\}$. Now call SPP(v, R_1, R_2) recursively.

We claim this recursive procedure runs in $O(n)$ time. Each vertex may get added and removed at most a constant number of times – and that is done at each step. A more rigorous proof, contained in the above cited paper, involves bounding the regions defined by the "ropes" and employing Euler's Formula.

(4) There are different ways of solving the problem. The main "difficulty" with the algorithm discussed in class is that when we process a point on the chain opposite the reflex chain, we can add diagonals to all points on the reflex chain. That is not true anymore. One way to fix the algorithms is to maintain two reflex chains rather than one (when necessary). Many people came up with variations of this method.

Mike Murphy came up with a different scheme that seems to work. Let us know if you think there are problems with it.

The algorithm, (a slight modification of the algorithm given in Preparata and Shamos page 239) is as follows:

(a) Pick the top and bottom vertex and break the polygon into two chains. If these are not unique, a small "rotation" can be applied to the polygon. (We do not actually do the rotation, but do it only conceptually to pick the points.) This breaks the polygon into two chains. Sort the vertices by y–coordinate. This is in essence merging two lists (chains) and hence can be done in $O(n)$ time. If during the course of merging, we encounter a case where we have two sets of vertices with the same y–coordinate with one set on the left chain and the other set on the right chain, then add the diagonal connecting the rightmost point on the left chain to the leftmost point on the right chain.

Note that this splits the polygon into smaller monotone polygons. Call the triangulation procedure recursively on the smaller polygons.

(b) If there many vertices that are at the same top and bottom y–coordinates, then apply the conceptual rotation to decide on a unique pair. Split the polygons into two chains and place the top two vertices on the stack. Now execute the algorithm as specified in Preparata and Shamos.

The reason this scheme works is that by the preprocessing step, we have partitioned the polygons into simpler polygons, with the property that even if we have a set of vertices on one chain with the same y–coordinate, we are guaranteed that the vertices on the other chain have different y–coordinates. This is sufficient to guarantee that we can add diagonals from a vertex in one chain to all vertices on the opposite reflex chain (I hope!).

(5) (Problem (5) of HW 1.) The basic idea is to "walk" from point 1 to n and at each step to check for intersections with a certain subset of previously traversed line segments, called *dangerous*. At each step we will add and delete points from the set. Each point will be added and deleted a constant number of times, thus this entire algorithm runs in $O(n)$ time.

To make the algorithm more concrete, we will maintain two lists of segments. The first is called *L-dangerous* and is the list of segments that we might potentially intersect from the left (assuming the segment is oriented from j to $j+1$). The second is the list called *R-dangerous*, the list of segments we might potentially intersect from the right. Each such list is a "continuous" list of segments. We record the segments by recording the endpoints.

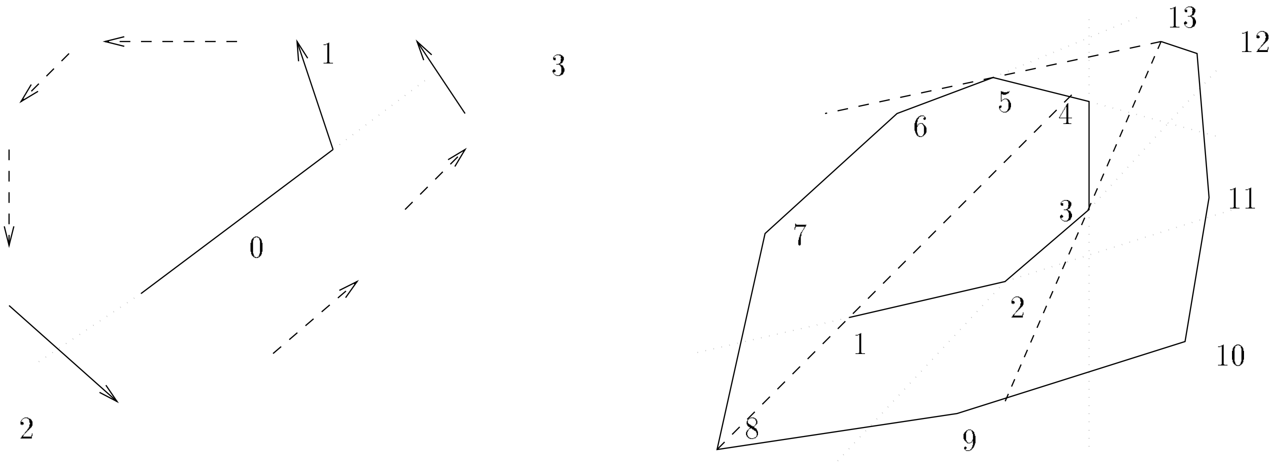

Figure 3: Characterizing dangerous segments.

Fig. 3(a) shows an arbitrary segment, together with an explanation of its life cycle. When we process segment 1, segment 0 is added to the set *L-dangerous*. When we cross the lower extension of segment 0 by processing segment 2, we add segment 0 to the set *R-dangerous* and delete it from *R-dangerous* when we cross its upper extension, by processing segment 3. (It may happen that we delete segment 0 from one of the sets even earlier; if certain conditions are detected that guarantee that if we intersect it from a certain direction, we would also intersect some other segment, and would find that intersection.)

We illustrate the main points via an example in Fig. 3(b) and then describe the details of the algorithm. The current point we are processing is point 13. The current set *R-dangerous* consists of segments (3,4) and (4,5). Segment (5,6) will be added only after crossing its extension. Segment (1,2) was added to *R-dangerous* when we went from point 7 to 8, then segment (2,3) was added in going from 8 to 9, segment (3,4) was added in going from 9 to 10. Segment (1,2) was deleted in going from 10 to 11, in going from 11 to 12 we add (4,5) and delete (2,3). Notice that at one end segments are added, and are deleted from the other end.

We now discuss the history of the set *L-dangerous*. When we are at point 13, the set consists of the segments (9,10), (10,11), (11,12). We add segments (1,2), (2,3), (3,4), (4,5), (5,6), (6,7). When we go from 7 to 8, we add the segment (1,2) in *R-dangerous*. We extend the line from 8 to 1, and can delete all the segments from *L-dangerous* to the right of this line. We repeatedly check segments from the end of the list and eject them if they are to the right. The list *L-dangerous* at this point contains segments (4,5), (5,6), (6,7) in that order. As we move from 8 to 9 to 10 etc, we add new segments at the head of the list and delete segments from the end of the list. The check is done by extending a line from the current point i through the last point in *R-dangerous*, which is point 1. Only when we reach point 11, will the first point in the list *R-dangerous* change (it changes to 2).

When we process i, compute the "turning angle", namely the angle $\theta = \angle(i-1, i, i+1)$ $(0^o < \theta \leq 180^o)$ and process the new segment $(i, i+1)$ according to this angle. Depending on the angle, the data structures and intersection tests have to be done appropriately. We will assume that we have already scanned the list of points to ensure that the turning angle is locally correct (makes a left turn).

Lh and Lt will indicate the head and tail of the list L-dangerous. Rh and Rt will indicate the head and tail of list R-dangerous. If $Rh = Rt$ it implies that the list is empty. We will use these variables to indicate points. We use $H(p, q)$ to denote the open half-plane to the right of the directed line through p and q.

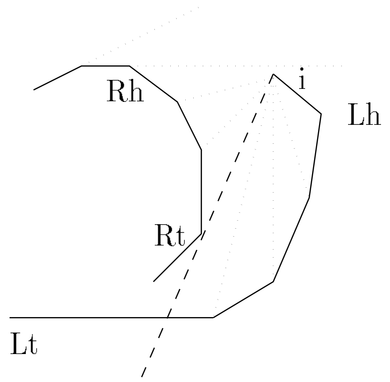

Figure 4: Steps that need to be done when we move from i to $i + 1$.

At i, the first thing we check for is if the outward spiral is continuing. If so, we update the set *R-dangerous*. If not, we update *R-dangerous* and then check for an intersection.

Check for intersection

```
Lt = 1; Lh = 2 (* Initialize the list L-dangerous *)
Rt = Rh = 1 (* Initialize the list R-dangerous *)
outward = true
for i = 3 to n − 1 do (* Process moving from point i to i + 1 *)
    θ = ∠(i − 1, i, i + 1) (* Compute turning angle *)
    if outward and θ > ∠(i − 1, i, Rh) then (* Outward spiral continues *)
      (* Extend R-dangerous by adding new segments at the head *)
      while i + 1 ∈ H(Rh, Rh + 1) do
            Rh + +;
      (* Delete segments from the tail in R-dangerous *)
      while Rh ≠ Rt and i + 1 ∉ H(Rt, Rt + 1) do
            Rt + +;
    else
      (* Delete segments from the head of R-dangerous due to a sharp turn *)
      while Rh ≠ Rt and θ ≤ ∠(i − 1, i, Rh − 1) do
            Rh − −; outward = false;
      (* Check for intersection with segment (Rh − 1, Rh) *)
      if Rt ≤ Rh and (i, i + 1) ∩ (Rh − 1, Rh) then exit
      (* Delete segments from the tail in R-dangerous *)
      while Rh ≠ Rt and i + 1 ∉ H(Rt, Rt + 1) do
            Rt + +;
    end-if
    Lh + +; (* Add the segment (i − 1, i) to L-dangerous *)
    (* Delete segments from L-dangerous due to segments of R-dangerous blocking *)
    while Lt + 1 ∉ H(Rt, i + 1) do
          Lt + +;
    (* Delete segments from the tail of L-dangerous due to a sharp turn *)
    while θ ≤ ∠(i − 1, i, Lt + 1) do
          Lt + +;
    (* check for intersection with segment (Lt, Lt + 1) *)
    if (i, i + 1) ∩ (Lt, Lt + 1) then exit
end-for
end-proc
```