Point Location: Edelsbrunner's Algorithm*

Scribe : Gabriel Wachman

Spring, 2004

1 Introduction

In this lecture, we present Edelsbrunner's algorithm for planar point location. Point location is in 2-d is defined as follows: Given a graph (or map) and a query point q, locate the region that contains q. This problem is fundamental to many other problems in computational geometry. Any time we need to answer the question "Where am I?", we have a point location problem. Edelsbrunner's algorithm constructs "chains" from a given planar graph. These chains are rooted at two artificially inserted points, one at negative infinity and the other at positive infinity. The chains themselves are sets of connected line segments with an above/below relationship, that allow us to perform a binary search to determine the two chains that bound our query point. Kirkpatrick's algorithm takes a different approach and computes an incremental triangulation of the graph, first by completely triangulating the graph, then removing independent sets until there is only one triangle left. We can make use of the information stored in this incremental triangulation to find our point in logarithmic time.

2 Edelsbrunner (1986)

For the Edelsbrunner Algorithm, we assumet that we are starting with a planar graph. The preprocessing portion of Edelsbrunner requires that we first construct the dual of the graph. This is done by converting every face

*These notes are partially based on notes scribed by Ning Wu in 2002

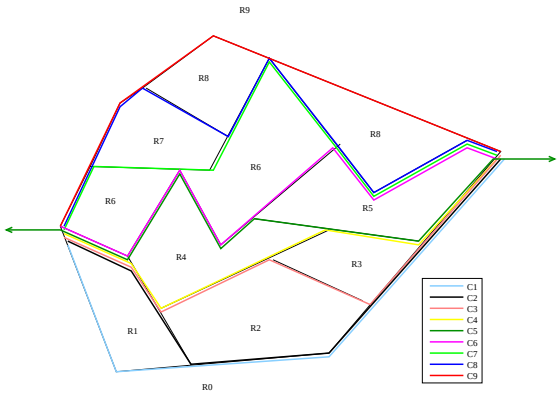

Figure 1: Example of chains for a given planar graph

to a vertex, and drawing an edge between any vertices that were adjacent faces in the original graph. Then, construct a set of "separator chains" on the dual such that each separting chain Ci separates a given region from all those regions less than it. Here, less than/greater than is determined by a region's ordering in the Y direction. See Figure 1, which shows a planar graph that has been appropriately divided using chains. This preprocessing step can be accomplished using a Topological Sort, in $O(n^2)$ time and space. A good data structure for storing the graph would be a DCEL, as this would easily allow us to "walk around" the regions of the graph to construct our chains. The chains themselves can simply be linked lists of vertex indices in the DCEL.

Note that the separating chains of the dual need not be disjoint. In fact, it is more likely that they are not disjoint. It is possible that there are $n-1$ separator chains of length $\Omega(n)$ in a graph with n regions. See Figure 2. Hence, the number of chains is $O(n^2)$.

Our algorithm uses the constructed separating chains, and performs a binary search on the chains to locate the region in which our query point falls. This takes $O(\log n)$ per chain $\times O(\log n)$ chains for a total of $O(\log^2 n)$. The space complexity is $O(n^2)$ if we store each vertex and edge of each chain separately, but remember that they are not disjoint, so in fact we can reduce our space complexity to $O(n)$ by storing each edge and vertex exactly once. The following algorithm uses $O(n)$ space complexity. Note that in the following algorithm, the separating chain s_i is above the regions $R_0, R_1, , , , R_{i-1}$ (See figure 1 for visual). We denote *above(e)* to be the region

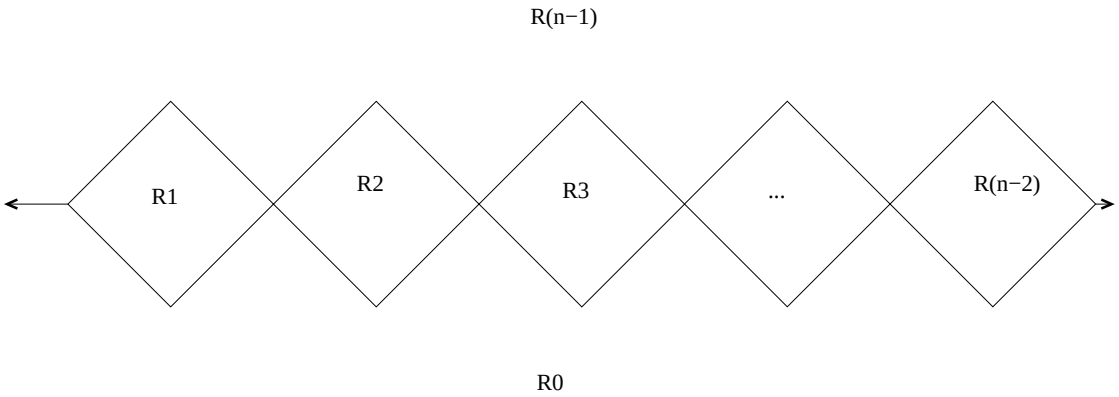

Figure 2: Planar subdivision with n regions and $n-1$ separator chains of length $\Omega(n)$ [3]

above the edge on vertex e. Then we know if $i = index(below(e))$ and $j = index(above(e))$, then separators containing e will be $s_{i+1}, s_{i+2}, , , , , s_j$.

The algorithm for location of a point P uses two levels of binary search. The inner loop takes a separator s_i and determines by binary search an edge e of s_i whose x-projection contains the abscissa P_x of P.

The outerloop performs binary search on i, in order to locate P between two consecutive separators s_i ands_{i+1} (i. e in a region R_i).

2.1 Algorithm 1[3]

1. set $i \leftarrow 0$, $j \leftarrow n-1$, $k \leftarrow lca(0, n-1)$.
2. while $i < j$ do
   /* p is above s_i and below s_{j+1}. so P is in one
   of the regions R_i, R_{i+1}, , , , , , R_j. */
   3. if $i < k \leq j$ then
      4. find by binary search an edge e in s_k s. t
      $P_x \epsilon$projection of e.
      let $a \leftarrow index(above(e))$;
      $b \leftarrow index(below(e))$.
      by testing P against e, we conclude it is either .
      5. if P is on e, set $loc \leftarrow e$;terminate.
      6. if P is above e, set $i \leftarrow$ a;
      else set $j \leftarrow b$.

7. else
    8. if $k > j$ set $k \leftarrow leftchild(k)$,
    elseif $k \leq i$ set $k \leftarrow rightchild(k)$.
9. set $loc \leftarrow R_i$;terminate.

2.2 Analysis

Binary search along any separator can be performed in$O(\log n)$ time since the edges are stored in a linear array sorted from left to right. At each iteration, k descends one level, so no of iterations= $O(\log n)$. So complexity of the algorithm = $O(log^2 m)$.

Space: Remember earlier we stated that it is possible to use only $O(n)$ space to store the chains. This is accomplished in step 6 of the algorithm above. We can mark an edge when we first use it so that it is removed from any subsequent chains that contain it. Look at Figure 1 again, and observe that when more than one chain share an edge, we really only need to look at that edge once in our binary search. If edge e is shared by chains i and j, then if the point falls in the x-region of e, it is above both i and j in that region - there is no need to check both chains. The general rule, is that in a binary search ordering of the chains, an edge is stored with the chain at the lowest level (root = level 0) in the binary search tree. In other words, an edge e that belongs to chains $s_{i+1} \ldots s_j$ need only be stored with the least common ancestor (lca) of i and j. When an edge is not stored with a chain, this represents a gap in the chain. The term "gap" will be used later on, and originates from our choice of linear storage.

Time: The algorithm above runs in $O(\log^2 n)$. We can improve this to $O(\log n)$ as follows:

When we compare a point P against a chain C_k, we find the edge in C_k whose x-region contains P. In other words, we find an x-region that contains P. We can represent all such x-regions of C_k as a list L_k.

Note that an element i of L_k exactly overlaps edge i of C_k, and overlaps *at most* 2 edges or gaps (remember our linear storage) in each of $C_{right\_child(k)}$ and $C_{left\_child(k)}$, and hence overlaps at most 2 intervals of $L_{left\_child(k)}$, and at most 2 intervals or gaps of $L_{right\_child(k)}$.

Before we go into the details f the algorithm, it is important to understand that what we are doing here is trying to avoid having to perform a binary search on each chain in order to find the edge whose x-region contains our query point. By using the information in the above paragraphs, we

can construct a data structure that will allow us to avoid havine to repeat this computation, and will ultimately save us a factor of $O(\log n)$ in time complexity.

Lists $L_1, \ldots, L_n$ and their connections (see definition) are represented by a linked data structure called the *layered dag.*

Definition 2.2.1 *A **layered dag** is a **d**irected **a**cyclic **g**raph whose nodes correspond to tests of three kinds: x-tests, edge tests and gap tests. Each x-value of L_k (the x-values of the vertices of chain C_k) generates an x-test and each interval between successive x-values (the edges, or gaps, since we are not storing edges twice) generates an edge or gap test. An x-test node t contains the corresponding x-value of L_k, denoted by $xval(t)$ and two pointers $left(t)$ and $right(t)$ to the adjacent edge or gap nodes of L_k. An edge or gap test node t contains two links $down(t)$ and $up(t)$ to appropriate nodes of $L_{l(k)}$ and $L_{r(k)}$. The layered dag contains a distinguished node root where the point location search begins. This node is the root of a balanced tree of x-tests whose leaves are the edge tests corresponding to the list for the root node.*

2.3 Algorithm 2[3]

1. set $i \leftarrow 0$, $j \leftarrow n-1$, $t \leftarrow$ root of the layered dag.
2. while $i < j$ do:
    3. if t is an edge test then let $e \leftarrow edge(t)$ and do:
        4. if P is on e, set $loc \leftarrow e$ and terminate.
        5. if P is above e,
            set $t \leftarrow up(t)$ and
            $i \leftarrow index(above(e))$
        else
            set $t \leftarrow down(t)$ and
            $j \leftarrow index(below(e))$.
    6. else t is an x-test then do:
        7. if $P_x \leq xval(t)$ then
            $t \leftarrow left(t)$
            else $t \leftarrow right(t)$.
    8. else t is a gap test do:
        9. if $j < chain(t)$ then $t \leftarrow down(t)$ else $t \leftarrow up(t)$.
10. set $loc \leftarrow R_i$ and terminate the search.

Now we have to show storage requirement is still $O(n)$. To prove, we show that the total number of x-values in the lists $L_1, \ldots, L_n l$ is at most 4n.

Theorem 2.3.1 *Space requirement for Algorithm 2 is linear.*

Proof 2.3.2 *If $a(k)$ denotes the number of edges in C_k, then $\sum_{k \in T} a_k = n$, since each edge of the subdivision occurs in exactly one chain C_k. Let b_k denote the number of x-values in L_k, and $A_k(resp.B_k)$ denote the sum of $a_i(resp.b_i)$ over all nodes i in the subtree rooted at k. We will show that $B_r \leq 4a_r = 4m$ where $r = lca(0, n-1)$. Proof is by induction on height. $B_i + b_i \leq 4A_i$ for $i = l(k)$ or $i = r(k)$ which is trivially true for leaves of T. We know $B_k = B_{l(k)} + B_{r(k)} + b_k$ and $b_k \leq 2a_k + (b_{l(k)} + b_{r(k)})/2$ since each edge in C_k contributes at most 2 x-values to L_k.*

$$\begin{aligned} b_k + B_k &= B_{r(k)} + B_{l(k)} + b_k + b_k \\ &\leq B_{r(k)} + B_{l(k)} + 2(2a_k + (b_{l(k)} + b + r(k))/2) \\ &\leq B_{r(k)} + B_{l(k)} + 4a_k + b_{l(k)} + b_{r(k)} \\ &\leq 4A_{r(k)} + 4A_{l(k)} + 4a_k \\ &\leq 4A_k. \end{aligned}$$

Computation of lca(least common ancestor):

We have the following formula for computing the least common ancestor of i and j:

$$lca(i, j) = j \wedge \neg(msb(i \oplus j) - 1) \tag{1}$$

where $msb(k) = lca(0, k)$.

$msb(k)$ for $k = 1, 2, \ldots, n-1$ can be computed in $O(n)$ time and stored in a table with $n-1$ entries. There is another way of computing lca which uses bit-reversal function $rev(k)$ that reverses the order of bits in binary expansion of k. So $lca(i, j) = rev(k \oplus (k-1)) \wedge j$ where $k = rev(i \oplus j)$.

References

[1] H. Edelsbrunner, L.J. Guibas and J. Stolfi, Optimal point location in a monotone subdivision,*SIAM J.Compute.*,15,1986,pp.317-40.

[2] H. Edelsbrunner, L.J. Guibas, Topologically Sweeping an Arrangement, *Proc. 18th ACM STOC*,1986.

[3] Dobkin and Souvaine, "Computational Geometry - A User's Guide"