Original Lecture #15: February 27, 1992
Topics: Point Location Methods
Scribe: Eric Veach *

1 Point Location in Two Dimensions

Given a polygonal subdivision of the plane and some query point, we would like to know which region (or edge or vertex) contains it. In one dimension, the problem reduces to locating a query point in a subdivision of the line into intervals, and is easily solved by binary search. We will look for a method to extend the idea of binary search into two dimensions.

Given a subdivision with m edges, we would like a method which will use $O(m)$ preprocessing time and achieve $O(\log m)$ query time using $O(m)$ space. The algorithm we will discuss below employs two nested binary searches and is based on monotone polygonal lines called *separators*. For more details on this algorithm, see [1].

1.1 Monotone Subdivisions

A region of the plane is *y-monotone*, or simply *monotone*, if its intersection with any line parallel to the y axis is a single interval (possibly empty). A planar subdivision is said to be monotone if all of its regions are monotone and it has no vertical edges (this last condition is not strictly necessary, but helps simplify our proofs). See Figure 1 for an example of a monotone subdivision.

The process of adding edges to make a subdivision y-monotone is called *regularization*. There is a simple algorithm based on a vertical sweep line, due to Lee and Preparata [2], which can regularize a planar subdivision in worst-case time $O(m \log m)$, where m is the number of edges. It is based on the observation that a subdivision is monotone if and only if every vertex is incident to at least two edges, one to the left and one to the right (such a vertex is called *regular*). Their algorithm adds at most one extra edge for each vertex which is not regular.

Starting at this point we will assume that the planar subdivision we are dealing with is y-monotone. This is a weak restriction, since convex polygons (including triangles) are monotone in every direction. Given non-monotone polygons, we can regularize them as described above or simply triangulate them (for which a worst-case linear time algorithm is known).

*A slightly modified version of Karen Daniels' notes at MIT

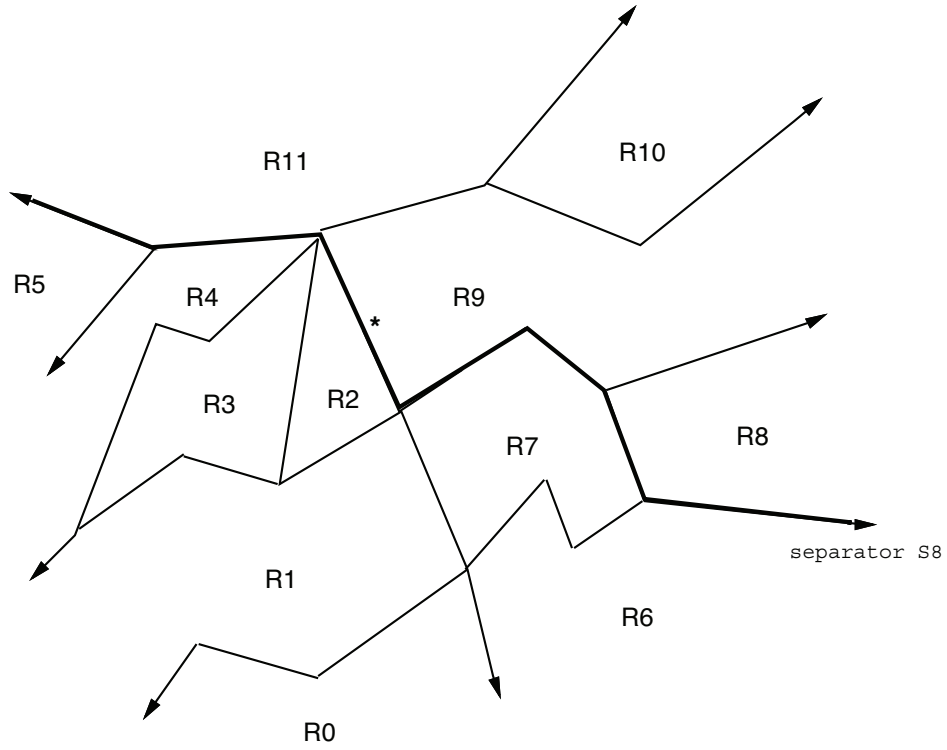

Figure 1: A separator for y-monotone subdivision.

1.2 Separators

A *separator* for a subdivision S is a polygonal line, consisting of edges and vertices of S, such that it meets every vertical line at exactly one point (see Figure 1). We introduce the following notation for our discussion of separators:

- The index of a region is denoted: $index(R_8) = 8$.
- The relations $above(e)$ and $below(e)$ denote the regions which lie immediately above or below an edge e.
- A separator s_i is above regions $R_0, R_1, \ldots, R_{i-1}$, and below regions with index i or higher. For example, the separator shown in Figure 1 is s_8.
- For two subsets A and B of the plane, we can define acyclic relations $\gg$ and $\ll$. We say that $A \gg B$ (A is *above* B) if "for every pair of vertically aligned points (x, y_a) of A and (x, y_b) of B we have $y_a \geqslant y_b$, with strict inequality holding at least once." [1] Similarly, $B \ll A$ means that B is *below* A.

Separators have many subdivision edges in common. For example, the edge marked with a '*' in Figure 1 is in separators $s_2, s_3, \ldots, s_8$. In general, we have the following lemma:

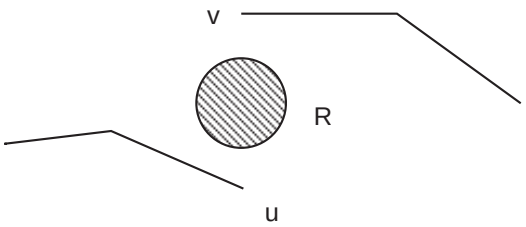

Figure 2: A contradiction for R.

Lemma 1. *For an edge e, let $i = index(below(e))$ and $j = index(above(e))$. Then e is contained in all separators $s_{i+1}, \ldots, s_j$.*

For a given subdivision containing n regions, many separators exist. The *complete family of separators* is of special interest to us. It is a collection of $n-1$ distinct separators, such that: $s_1 \ll s_2 \ll s_3 \ll \ldots \ll s_{n-1}$. Complete separators have the property that between each pair of separators there exists one region: $R_0 \ll s_1 \ll R_1 \ll s_2 \ll R_2 \ll \ldots \ll s_{n-1} \ll R_{n-1}$.

If $n-1$ separators exist, they form a complete separator family, in the sense that they completely separate all regions. Intuitively, we can think of a separator as a rope conforming to edges of the subdivision. The complete family then corresponds to a topological sweep from bottom to top in which the rope moves upwards. At each step, the rope loops over a region.

Theorem 2. *Every monotone subdivision admits a complete family of separators.*

Proof: The proof is by construction, and is given in [1]. We begin by constructing a linear (total) ordering of the regions, consistent with the $\gg$ relation, from the transitive closure of a partial ordering of the regions. This can be done because any acyclic relation over a finite set can be extended to a linear order [1]. The resulting linear extension contains some region R_0 with nothing below, one region R_{n-1} at the top, and, in between, the regions are ordered so that the region above has a higher number.

Given a total ordering of the regions, it is easy to obtain a complete family of separators. Consider a vertical line l through the subdivision. It cuts a set of regions $R_{i1}, R_{i2}, \ldots, R_{iq}$, from bottom to top, and also cuts R_0 and R_{n-1}. Since $R_0 \ll R_{i1} \ll R_{i2}, \ldots, \ll R_{iq} \ll R_{n-1}$, we have $0 \leqslant i1 < i2 \ldots < iq \leqslant n-1$. "Therefore, there is exactly one point on l that is on the frontier between a region with index $> i$ and a region with index $\geqslant i$, that is, on s_i. Furthermore, the intersection of l with s_i will be equal to or above that with s_{i-1}, and for some such lines (those that meet R_{i-1}) the two intersections will be distinct. So, we have $s_1 \ll s_2 \ll \ldots \ll s_{n-1}$" [1].

The proof is completed by noting that each separator must be monotone, consist of a sequence of monotone edges, and be connected. It can't be unconnected, because, if it were, some vertex v of separator s_i would have the same x-coordinate as an adjacent

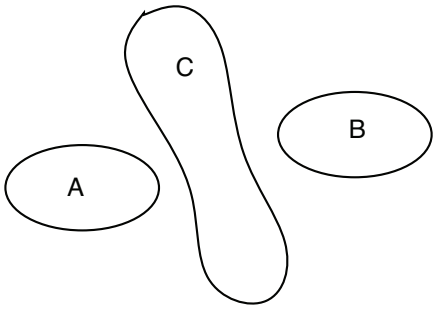

Figure 3: An intervening region C.

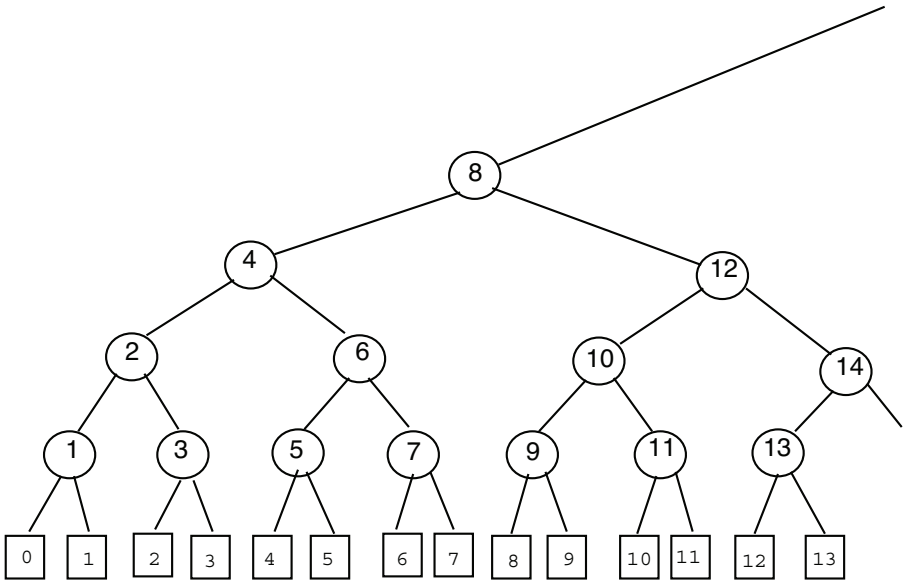

Figure 4: A tree of separators.

(but unconnected) vertex u, producing the contradiction that, for some region R, R is both above and below the same separator (see Figure 2). □

The linear extension is not uniquely defined, and some choices exist for comparable cases. Subtleties can occur when ordering the regions. When deciding the *above* relation for two regions A and B, it is important to look at the full picture, not just A and B. For example, it is not true that, for A and B whose projections are disjoint in x, we have complete freedom in deciding whether $A \gg B$ or $B \gg A$. Intervening regions can force relations, as in Figure 3. If A and B overlap in both x and y, the ordering is clear. If not, it may depend on other factors. In Figure 3, $C \gg A$, and $B \gg C$, so we must have $B \gg A$.

1.3 Point Location Algorithm

The point location algorithm employs two levels of binary search. The inner loop locates an $x-$interval for the query point p by searching, on a separator s_i, for an edge e of s_i whose $x-$interval contains p_x. The outer loop locates a $y-$interval via binary search on i. This finds for us two separators that bound y from above and below and, together with e from the inner loop, pinpoint a region containing p. The outer loop uses an infinite tree T, in which the separators of a complete family are represented as internal nodes, and regions correspond to leaves (see Figure 4). Our goal is to walk down to a leaf to find a region containing p.

The algorithm uses the notion of a least (lowest) common ancestor of two leaves (regions). $\text{lca}(i, j)$ is defined as the "root of the smallest subtree of T that contains both i and j" [1]. The lca of two nodes of T can be calculated in constant time (see section 3.5.2).

1.3.1 Algorithm

$i \leftarrow 0, j \leftarrow n-1, k \leftarrow \text{lca}(0, n-1)$

while $i < j$ do

  if $i < k \leqslant j$ then

    find the edge e of s_k that covers p (contains p_x)
    $a \leftarrow index(above(e))$
    $b \leftarrow index(below(e))$
    if $p \in e$ then set $location \leftarrow e$ and terminate
    if p *is above* e, then $i \leftarrow a$; else $j \leftarrow b$

  else if $k > j$ then $k \leftarrow lson(k)$; else $k \leftarrow rson(k)$

set $location \leftarrow R_i$ and exit

At the start of the point location algorithm, i and j represent the bottommost and topmost regions, respectively, and k is $\text{lca}(i, j)$. Throughout the algorithm i and j approach each other. At each iteration of the *while* loop, either i increases or j decreases, and k moves down one level in T. The loop invariant is: $s_i \ll p \ll s_{j+1}$. When i and j become equal, R_i is returned as the region containing p.

1.3.2 Example

To see how this algorithm operates, consider p positioned as in Figure 5. At the start of the algorithm, $i = 0$, $j = 11$, and $k = \text{lca}(0, 11) = 8$. The first binary search is performed on separator s_8 to locate edge e_1, whose $x-$interval contains p_x. The region

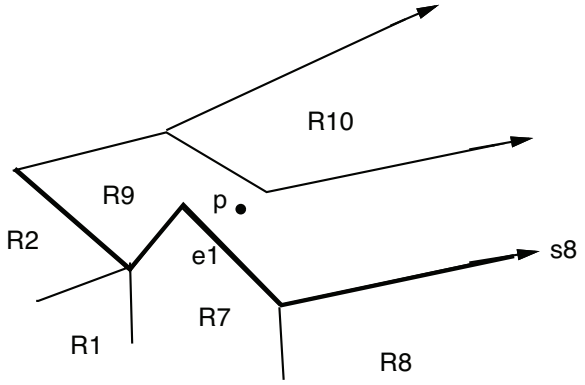

Figure 5: p above e_1

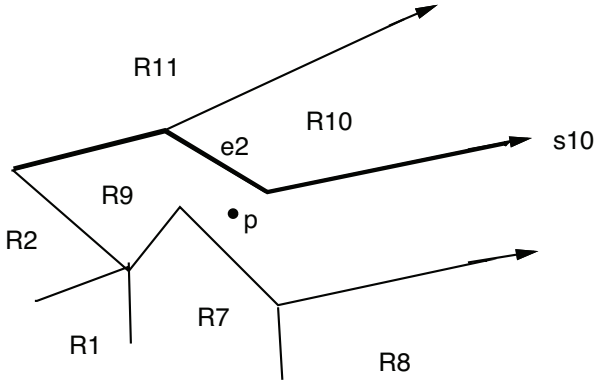

Figure 6: p below e_2

above e_1 is R_9, and the region below it is R_7, so $a \leftarrow 9$ and $b \leftarrow 7$. p is above e_1, so we raise the lower index i: $i \leftarrow 9$.

Now $i = 9$ and $j = 11$. Since $k = 8$ is outside the interval (9,11), k moves down one level of the tree: $k \leftarrow rson(8) = 12$. This value of k is still outside (9,11), so k again moves down, this time to $lson(12) = 10$. Now we search separator s_{10} and locate edge e_2 (see Figure 6). The region above e_2 is R_{10}, and the region below e_2 is R_9, so $a \leftarrow 10$ and $b \leftarrow 9$. p is below R_{10}, so $j \leftarrow 9$. Now $i = j = 9$, so the algorithm terminates by correctly locating p within R_9.

1.4 Analysis

The algorithm of section 3.2.1 requires the enhancements described in sections 3.4 and 3.5 in order to meet the goals of $O(m)$ preprocessing time, $O(\log m)$ query time, and $O(m)$ space.

1.4.1 Space

In its current form, the algorithm assumes that a complete list of edges is stored for each separator, so space requirements can be as high as $O(m^2)$. Section 3.4 shows how to reduce space requirements by storing each edge with only one separator.

1.4.2 Time

The two nested binary searches yield an overall running time of $O(\log^2 m)$. The updating of values of i and j in the outer loop differs from classical binary search. Although an edge can participate in many separators, i and j reflect the bottommost and topmost separators (respectively) within a group, making it possible, in some cases, to reduce the search space by more than $1/2$ during each iteration. The significant time savings, however, stems from two sources: fractional cascading and finding the lca in constant time. These topics are addressed in section 3.5.

1.5 Reducing Space Requirements

Each subdivision edge e is stored as part of only one separator, although it participates in many separators. (Recall lemma 1, which specifies the set of separators for e.) The choice of separator for e follows our philosophy of testing separators in a top-down fashion. We store e as part of the separator associated with the highest point in T involving e. That way, e will be available the first time it is needed. The highest point for e is given by $k = \text{lca}(i, j)$, where i and j are the regions below and above e; hence we store e in s_k. For example, using T in Figure 4, and referring back to Figure 5 for e_1, $k = \text{lca}(7, 9) = 8$, and for e_2 of Figure 6, $k = \text{lca}(9, 10) = 10$. This strategy results in $O(m)$ space for separators.

It is important to note that each edge is available the *first* time it is needed, and it will *not* be needed further down the tree. This is because once we have performed a binary search using e, the adjustments to i and j effectively bypass the other separators containing e, so that no further tests use e. Hence, we learn a great deal during each iteration, as pointed out in section 3.3.2. We can envision the new streamlined separator as a chain which, instead of being merely a sequence of edges, is now a sequence of edges and gaps. For each gap, the corresponding edge is stored somewhere higher up in the tree. Clearly no test will ever be made against a gap.

1.6 Reducing Time Requirements

1.6.1 Fractional Cascading

The data structuring technique of fractional cascading is described in section B1 of handout 2 ("Ruler, Compass, and Computer"). Search time is reduced for a multi-level structure by propagating some fraction of the information at each level up through the

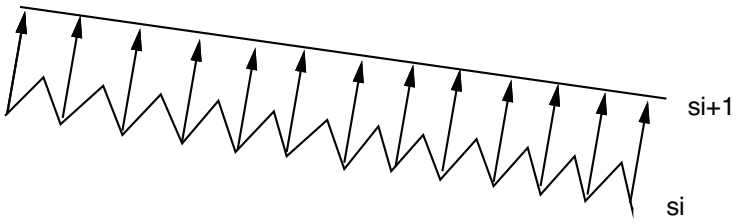

Figure 7: Propagating every other vertex

higher levels. In the context of the point location tree T, fractional cascading provides a powerful way of gaining $x-$interval knowledge during the first binary search of a separator at the highest level that helps reduce searching at lower levels.

Recall that the algorithm in its basic form requires a full binary search on each s_k to locate an edge covering p_x. Binary search on s_k yields an edge representing an $x-$interval, $[x_{1k}, x_{2k}]$, where $x_{1k} \leqslant p_x \leqslant x_{2k}$. To improve this situation, we can implement fractional cascading by preprocessing T. This involves propagating selected $x-$interval endpoints up the tree.

We begin at the lowest tree level, where each leaf "samples" itself by sending up to its parent a fraction of its data. (For example, in Figure 7, s_i uses a fraction of $1/2$ and sends every other endpoint up to s_{i+1}.) The parent now contains its data plus samples from its children. The parent in turn samples its new augmented self, and sends to its parent a fraction of its data. Propagation of samples continues all the way up the tree, with each node using the *same fraction*.

Searching the preprocessed tree is now much cheaper, because one binary search at the top node provides enough information to locate a position within each child of the top node in constant time. Instead of $O(\log m)$ binary searches, each requiring $O(\log m)$ time, we now only pay $O(\log m)$ once for the top separator, and perform $O(1)$ cost searches after that.

Although it seems, on the surface, that this method requires more than $O(m)$ storage, the storage remains linear because the amount of data propagating upward through the levels from any particular node decreases according to a geometric series. For example, for the fraction $1/2$, we have $1/2, 1/4, 1/8 \ldots$ and the storage only doubles. Although cascading can be done with any fraction, there is a space vs. time trade-off; sparser samples save space but increase search time.

1.6.2 Finding the LCA

In order to achieve $O(m)$ preprocessing time, we must be able to construct T, representing a complete family of separators, in $O(m)$ time. This process is described in [1]. It involves setting up separators in the manner set forth in section 3.4, which implies that $lca(i, j)$ can be computed in constant time. One way to accomplish this is to manipulate the binary representations of i and j. $lca(i, j)$ can be viewed as the

"longest common prefix of i and j, followed by 1 and padded with 0's [1]. It is therefore of the form: $(prefix10\ldots0)$. An efficient way to produce this is based on the *most significant bit* function $\text{msb}(k)$, which returns the most significant 1 in k, e.g. $\text{msb}(21) = \text{msb}(10101_2) = 10000_2 = 16$. If we precompute $\text{msb}(k)$ in a linear size array, then in constant time we can compute $lca(i,j) = j \wedge \neg(\text{msb}(i \oplus j) - 1)$, where $\wedge$ is bitwise-and, $\oplus$ is bitwise-xor, and $\neg$ is bitwise-negation.

References

[1] Edelsbrunner, Guibas, and Stolfi, "Optimal Point Location in a Monotone Subdivision", *SIAM J. Computing*, Vol. 15, No. 2, May 1986.

[2] Lee and Preparata, "Location of a point in a planar subdivision and its applications", *SIAM J. Computing*, Vol. 6, pp. 594-606.