# Preprocessing for the Chain Method of Point Location

Lecture 7
Date: March 1, 1993
Scribe: Maria Loughlin

## 1 Introduction

The chain method of point location was introduced in Lecture 6. This lecture describes the preprocessing required to implement the chain method. It also describes how to transform an arbitrary subdivision of the plane into a subdivision in which all regions are monotone. This allows the chain method to be applied to arbitrary point location problems.

Recall that the planar point location problem is stated as: Given a division of the plane into regions, $R_0$, $R_1$, ... $R_n$, and given a test point, $p$, find which region, $R_i$, contains $p$. The chain method uses monotone chains to divide the plane into regions. Point location is performed by discriminating $p$ against the chains.

## 2 Representation of the data required for the chain method

We must first decide on a data representation for the regions of the planar subdivision. Each region is described by the edges that delimit it from neighboring regions.

**Vertices:** Let each vertex be represented by its $x$ and $y$ coordinates in the plane ie $v_i = (x_i, y_i)$. If there are $n$ vertices in the Planar Straight Line Graph, PSLG, that defines the regions, all vertices can be stored in $O(n)$ space.

**Edges:** Edge information is stored in the adjacency lists of the vertices. Since each edge will be listed twice (eg the edge from vertex $v_i$ to vertex $v_j$ will be listed in the adjacency lists of both $v_i$ and $v_j$), the space requirement to capture all edge information of the PSLG is also $O(n)$ If the edges have names, we associate the edge names with the vertices in the adjacency lists.

**Regions:** For each region, we store the list of vertices that delimit it from neighboring regions. For simplicity, we follow the vertices in a defined order, such as clockwise around the boundary of the region. Therefore, the region stored as $v_1$, $v_2$, $v_3$ ... $v_k$ corresponds to the region with edges $(v_1\ v_2)$, $(v_2\ v_3)$ ... $(v_k\ v_1)$ The space requirement here is also $O(n)$.

We will assume that the input data for the point location problem is in this format. If the vertices on the region boundaries are not already sorted, they should be sorted at this input stage of the process. This could take $O(n^2 logn)$.

# 3 Building the chain tree

As decribed in Lecture 6, the chain tree data structire, used in the point location method, is a binary tree in which the leaves represent regions in the plane and the nodes represent chains. To reduce the size of the binary tree, we store at each node only the portion of the chain that isn't already stored at an ancestor of that node. Therefore, every edge is stored once in the chain tree - at the highest node of the chains in which it appears.

The task of building the chain tree can be split into two parts: We first associate regions with the leaves of the tree and chains with the nodes of the tree, such that each chain, $C_k$, is positioned such that regions $R_1$ to $R_k$ are to its left, and regions $R_{k+1}$ to $R_n$ are to its right in the binary tree. This is achived in the following steps

- Draw the inverse graph of the PSLG.
- Do a topological soft on the inverse graph.
- Using the order of the regions defined by the topological sort, assign a region to each leaf of the tree.
- Assign a chain to each node of the tree, such that regions to the left of each chain are to the left of the node associated with that chain, and regions to the right of the chain are to the right of the node.

Then we insert edges in the tree, such that the edges of each chain are at the node associated with that chain or at ancestors of that node.

The steps of this process are described in the following sections.

## 3.1 Draw the inverse graph

The inverse graph of the PSLG is created by traversing the graph, and for each edge, $e$, connecting the region to the left of $e$ to the region to the right of $e$. Since there is one node of the inverse graph for each region, we can associate the nodes of the inverse graph with the regions of the PSLG.

To find the regions to the right and left of an edge $e$:

1. Scan the vertex list of all regions to select the two regions which contain $e$ on their boundary.
2. For the selected regions, find the vertices with the largest and smallest y-coordinate, $p_{hi}$ and $p_{lo}$.

3. Determine whether $e$ is on the left or right chain joining $p_{hi}$ and $p_{lo}$. If the edges of each region have been stored in clockwise order, this is achieved by simply comparing the x-coordinate of the endpoints of $e$ with the x-coordinate of an edge on the opposite path from $p_{hi}$ to $p_{lo}$.

If $e$ is on the left chain that joins these vertices, then $R$ is to the right of $e$. Conversely, if $e$ is on the right chain, $R$ is to the left of $e$. Figure 1 illustrates this process.

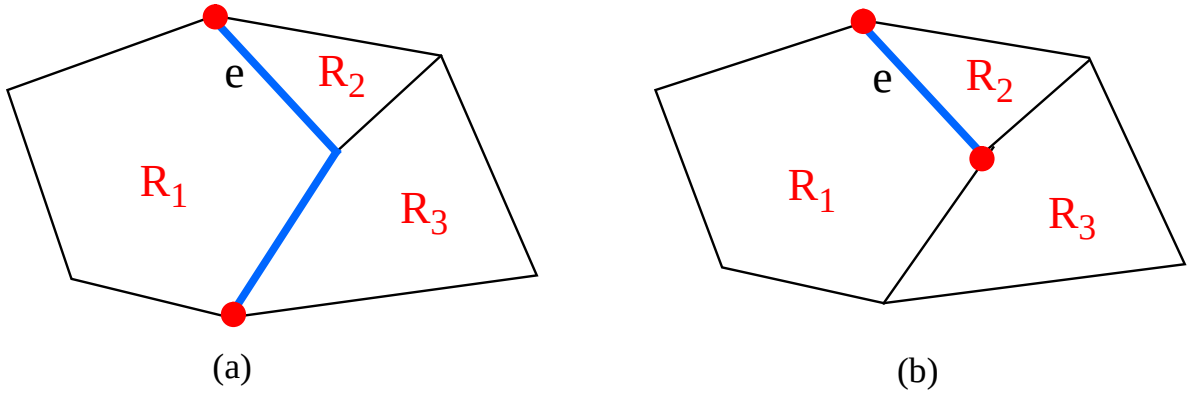


Figure 1: Finding the regions to the left and right of an edge (a) $R_1$ is to the left of edge $e$. (b) $R_2$ is to the right of $e$

Each of the steps above can be done in $O(n)$ time, so the process of finding regions to the left and right of all edges will require $O(n^2)$ time. Therefore, the time required to create the inverse graph is $O(n^2)$.

## 3.2 Do a topological sort on the Inverse Graph.

We perform a topological sort on the inverse graph to impose an order on the regions of the planar subdivision. A topological sort can be implemented as a variation of the breadth first search algorithm and runs in time $O(n)$. Refer to [3] for further details of the topological sort.

Next we asociate the regions in the plane with the leaves of the binary tree, in the order produced by the topological sort. That is, region $R_1$ will be associated with the leftmost leaf of the tree, region $R_2$ with the second leftmost leaf etc. We then associate chains with nodes of the tree. Chains $C_1$ to $C_n$ will be assigned to the nodes in an pre-order traversal of the tree. This assignment gives the required relationship between the chains and regions in the binary tree, as illustrated in figure 2

## 3.3 Associate Edges with Regions

The next task is to insert the edges of the PSLG in the nodes of the chain tree

The brute force method to do this is to traverse each chain, in the order defined by an in-order traversal of the chain tree. For each edge on each chain, check if this edge is stored at an ancestor node. If not, store the edge at the current node. This approach requires $O(n^2)$ operations.

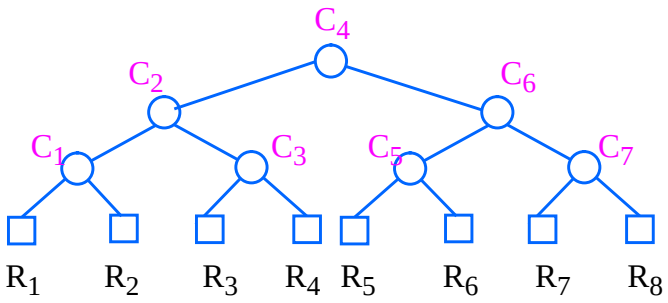


Figure 2: Assignment of regions and chains to the leaves and nodes of a chain tree

The time requirement can reduced by considering each edge in turn (rather than each chain in turn). Using the information about the regions to the left and right of the edge (computed in step 3.1), we identify the leaves of the chain tree associated with these regions. We place the edge at the node of the lowest common ancestor of these leaves. An efficient (constant time) method for finding the lowest common ancestor of leaves of a binary search tree is described in Appendix A.

Finally, we sort the edge list at each node. We do this by putting a ficticious vertex at the center of every edge of the PSLG. We do a topological sort on these vertices (assume edges are pointing upward). This will require time $O(n)$.

We have now produced the chain tree required for the chain method of point location in $O(n^2)$ time. The chain method can be applied for any input point $p$, as described in Lecture 6.

# 4 Using the Chain Method for Non-Monotone Regions

As described above, the chain method can only be applied to regions that have monotonic chains at their borders. However, a subdivision of the plane is not monotone in general, since it may contain regions delimited by arbitrary polygons. In this section, we discuss a procedure which transforms an arbitrary polygon with $n$ vertices into a set of connected monotone polygons and runs in time $O(nlogn)$. This is achieved by introducing a new edge at vertex at which the polygon is non-monotone.

A non-monotone polygon contains one or more *cusps*, that is vertices at which the chain is not monotone. Consider a horizontal line drawn through a vertex, $v$. If both edges leaving $v$ lie beneath the line, $v$ is an *upward cusp*. If both edges leaving $v$ lie above the line, $v$ is an *downward cusp*. Figure 3 shows a non-monotone polygon with upward and downward cusps.

The edge-addition algorithm uses two passes of a line sweep over the polygon. The first pass (descending pass) removes all upward cusps by introducing a new edge between each upward cusp and the vertex of lowest y-coordinate above it. The second pass (ascending pass) removes all downward cusps by introducing a new edge between each downward cusp and the vertex of highest y-coordinate below it. At all times during the line sweep, we maintain a list of the edges intersected by the horizontal line. We associate with each edge in the edge list the maximum (minimum) y-coordinate vertex, $v$, between this edge and the

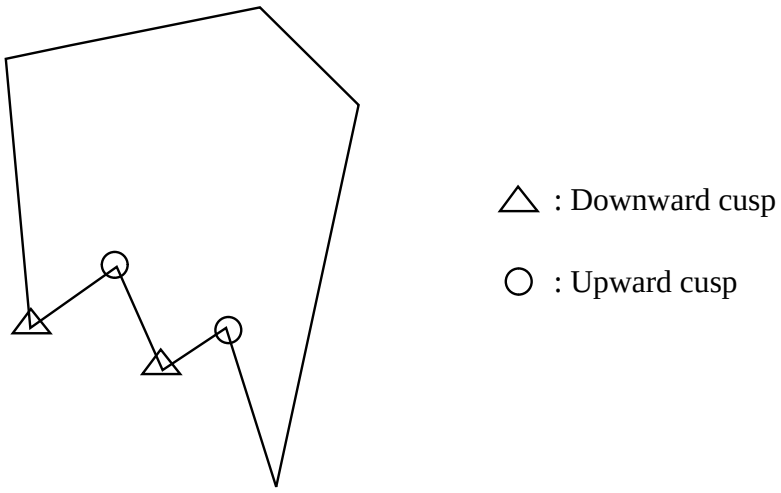


Figure 3: A non-monotone polygon showing upward and downward cusps

next edge in the edge list.

In detail, the steps to eliminate the upward cusps are:

1. Sort the vertices of the polygon by decreasing y-coordinate.
2. Initilize the edge list to contain a dummy edge $e_0$ from the uppermost vertex, $v_1$ to -infinity. Add the leftmost edge from $v_1$, $e_1$ with associated vertex $v_1$. Add the rightmost edge from $v_1$ with associated vertex $v_1$.
3. Set $i = 2$.
4. While $i < n$ do:
5. Set $j$ to the smallest integer so that $v_i$ is not to the left of $e_j$
6. If $v_i$ is a downward cusp, set $v_i$ to be the vertex associated with $e_{j-1}$ and with $e_{j+2}$. Then delete $e_j$ and $e_{j+1}$ (and their associsted edges) from the edge list.
7. If $v_i$ is an upward cusp, insert the edges emanating from $v_i$, between $e_j$ and $e_{j+1}$, in the edge list. Add the edge $e_j$ to a new list, called the auxiliary list. Associate $v_i$ with each of these edges.
8. If $v_i$ is not a cusp, asociate $v_i$ with edge $e_j$ and $e_{j-1}$. Set $e_j$ to be the outgoing edge of $v_i$.
9. Increment $i$. Goto step 3.
10. Halt

At this point the auxiliary list contains edges which, when added to the PSLG, will eliminate all upward cusps.

The ascending algorithm is similar, but with the sweep moving in the opposite direction.

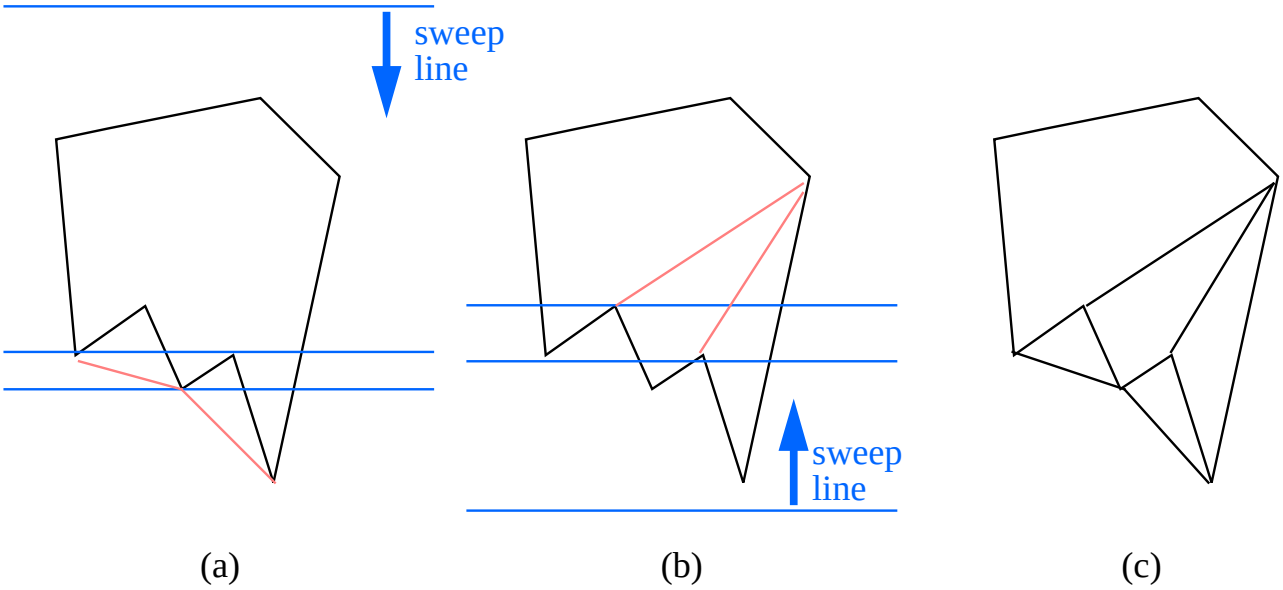


Figure 4: Adding edges to the polygon (a) Descending line sweep (b) Ascending line sweep (c) The resultant polygon.

Figure 4 shows the edges added during the descending and ascending line sweeps on the polygon of figure 3.

We label all regions that have been created during this edge addition process with the label of the original (non-monotone) polygon P. Then when the point location algorthm is implemented, if the point $p$ lies in any of these regions, the correct region is reported.

Analysing this algorithm, we see that the initial sorting pass of the vertices requires $O(nlogn)$ operations. Step 4 involves tracing a path in the edge list. If the edge list is implemented as a balanced tree, this requires at most $O(logn)$ operations. Thus the total amount of work involved in step 4 is $O(nlogn)$. Each update, deletion or insertion (steps 5,6 and 7) requires time at most $O(logn)$. So the total number of operations for these steps is $O(nlogn)$. In total, the algorithm requires $O(nlogn)$ time.

# A Finding the Lowest Common Ancestor of Leaves in a Binary Tree

In this appendix, we present an efficient algorithm for finding the lowest common ancestor of two leaves in a binary tree.

We begin by assigning a label to all edges of the binary tree. We assign *0* to all edges between a parent node and a left child and assign *1* to all edges between a parent and a right child. We associate with each leaf the sequence of labels in the path from the root to the leaf. Next, we number the internal nodes of the binary tree in in-order. This labeling scheme is illustrated in figure 5(a). Note that the longest common prefix of the labels of any two leaves defines the path from the root to the lowest common ancestor of those leaves.

Now, we notice that for any two nodes numbered $a$ and $b$, the decimal value of $2^{\lfloor log_2 a \oplus b \rfloor}$

- 1, where $\oplus$ is the binary EXOR function, gives the number of the node which is the lowest common ancestor of $a$ and $b$.

For example, if we look at leaves numbered 1 (001) and 5 (101) of the binary tree of figure 5(b), we find that:

$2^{\lfloor log_2 a \oplus b \rfloor}$ - 1 = $2^{\lfloor log_2 001 \oplus 101 \rfloor}$ - 1 = $2^{\lfloor log_2 100 \rfloor}$ - 1 = $2^2$ - 1 = 3

We have correctly identified that node $C_3$ is the lowest common ancestor of leaves 1 and 5.

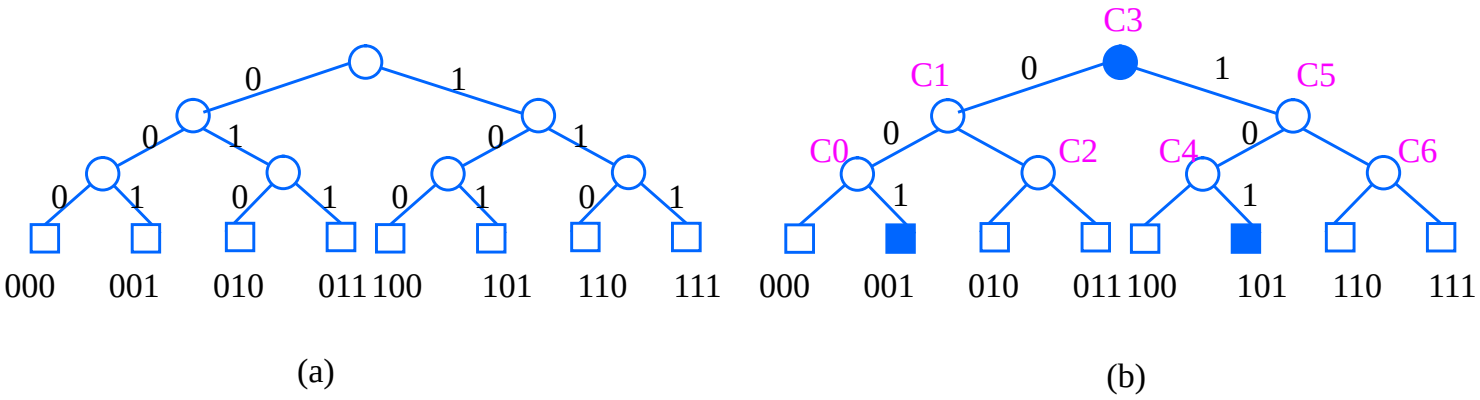


Figure 5: Finding the lowest common ancestor of nodes in a binary tree. (a) The labeling of edges and leaves (b) Illustration of lowest common ancestor of leaves 1(001) and 5 (101).

If we precompute and tabulate the values of $2^{\lfloor log_2 a \oplus b \rfloor}$ - 1 for all possible values of $a$ and $b$, we can find the lowest common ancestor of any two leaves in constant time with a simple table lookup.

# References


[1] F.P. Preparata and M.I. Shamos, *Computational Geometry, An Introduction* ,48-56, 1985.

[2] D.T. Lee and F.P. Preparata, "Location of a point in a planar subdivision and its applications", *Siam Journal on Computing* 6(3), 594-606, Sept. 1977

[3] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill 1990