6.854 Advanced Algorithms

Lecture 18: 11/12/2003 Lecturer: Erik Demaine, David Karger

Scribes: Nick Harvey

18.1 Trapezoidal Decompositions

In the previous lecture we discussed **randomized incremental construction**, a common technique used in computational geometry algorithms. In this lecture we will see this technique used again in an algorithm for finding **trapezoidal decompositions**.

18.1.1 Motivation

Consider the **line segment intersection** problem:

> Given n line segments in the plane, list all of their intersections.

In the previous lecture we saw a line-sweep algorithm to solve this problem in $O(n \log n + k \log n)$ time if there are k intersections. We will show that trapezoidal decompositions can be used to determine the intersection points in expected time $O(n \log n + k)$.

Next, consider the **point location** problem:

> Given a query point and subdivision of the plane defined by *line segments*, find the face containing the query point.

A more informal description of the problem is, given a map and a query point, find the region of the map containing the query point. Note that the present problem differs from the problem of point location in an arrangement of n *lines*, which we saw in the last lecture. In fact, we have already seen a solution to the line segment variant of the problem as well, based on persistent data structures. We will show that trapezoidal decompositions can also be used to solve this problem.

18.1.2 Definition and Preliminaries

Suppose we have a set of n line segments in the plane. We may assume that the points defining the line segments are in general position, so that no line segment is vertical. The endpoints of line segments and intersection points of line segments are called **vertices**. From each vertex draw vertical lines (called **extensions**) upward and downward until they meet another line. The point at which an extension meets another line is also called a vertex.

The line segments and extensions partition the plane into regions that are trapezoids and triangles, which may be regarded as degenerate trapezoids. Thus we have a "trapezoidal decomposition" of the plane. To avoid dealing with infinite trapezoids, we may assume that all original line segments are contained in a large rectangular bounding box; extensions stop when they meet this box. We define an **edge** to be a section of a line segment or an extension between two consecutive vertices.

Each endpoint and intersection point creates exactly two new vertices when its extensions meet another line (or the bounding box). Thus the total number of vertices is $O(n + k)$, where k is the number of intersection points. The vertices of largest degree are the intersection points, which have degree six. (Each line segment through an intersection point gives two incident edges, and each extension gives one edge). The total number of edges is at most the sum of the vertex degrees, which is $O(n + k)$. The number of faces is also at most the sum of the vertex degrees. Thus the number of faces is $O(n + k)$.

Our goal is to maintain a data structure representing the trapezoidal decomposition. We use the so-called "clockwise pointer representation", in which each vertex lists its edges in clockwise order. This representation makes it easy to walk around the faces of the decomposition, which we will do frequently during construction.

18.1.3 Construction

To build our data structure representing the trapezoidal decomposition, we use randomized incremental construction. Let $S = (s_1, \ldots, s_n)$ be a random ordering of the line segments. We insert the segments one-by-one in this random order, updating the data structure after each insertion.

Informally, we do the following work in each insertion step.

1. Draw extensions from each endpoint of the line segment.
2. Determine if the new segment blocks an existing extension. If so, remove the blocked section of each such extension.
3. Determine if the new segment intersects any existing segments. If so, draw extensions from each intersection point.

To implement these operations, it would be helpful if the line segment being inserted had a pointer to a face that (partially) contains it. We can achieve this by keeping a bidirectional pointer from each line segment that has not yet been inserted to the face that contains its left endpoint. That is, each face has a list of the line segments that have not yet been inserted and whose left endpoint lies in that face; additionally, these line segments each have a pointer to that face. These pointers are helpful during insertion, but of course they must be maintained as we split and merge faces.

To insert a line segment, we start at the face containing its left endpoint. We can find this face in $O(1)$ time using the line segment's bidirectional pointer. We then we walk around this face determining which edges are vertically above and below the left endpoint and which edge (if any) intersects the line segment. Call this edge the "leaving edge".

After walking around the face and finding the edges above and below the left endpoint we can easily create the new extensions from this endpoint. Adding these new extensions will split the current face into three smaller faces. We will consider the work required for this splitting operation below.

Now we consider the leaving edge, assuming that it exists. If the leaving edge is on an extension, the new line segment blocks that extension. We shorten the extension and merge the faces that were bordered by the removed section. If the leaving edge is on another line segment, we create a new intersection point and extensions from it. This causes to split the current face and the neighboring face.

We then repeat this same insertion process in the face on the other side of the leaving edge. If we ever discover that there is no leaving edge then the right endpoint of the line segment must be contained in this face. We complete the insertion process by creating extensions from the right endpoint, just as was done for the left endpoint.

18.1.4 Analysis of Construction

The algorithm for each insertion step walks around every face that intersects the new segment, and may split or merge that face. Walking around the face requires time linear in the number of vertices on that face. For a given face f, let us denote the number of vertices on that face by $n(f)$. When splitting or merging a face f we must update the bidirectional pointers from line segments that are yet to be inserted whose left endpoint lies in that face. Let us denote the number of these pointers by $\ell(f)$. It follows that splitting or merging a face additionally requires time linear in $\ell(f)$. Thus the total amount of work for a single insertion is

$$\sum_{f \in \{\text{faces traversed}\}} O\big(n(f) + \ell(f)\big)$$

We now use backwards analysis to bound the expected value of this sum. Suppose we are inserting the ith line segment. Let $S_i = \{s_1, \ldots, s_i\}$ denote the set consisting of the first i line segments in the random order. Let F_i denote the faces determined by the line segments in S_i. For any $s \in S_i$, let $F_i(s)$ denote the faces in F_i that intersect s. For the purposes of analysis, let us assume that the elements of S_i are known, but their order is not. Then the element inserted in the ith step is equally likely to be any element of S_i. Then the expected amount of work, conditioned on the particular elements of S_i, is

$$\begin{aligned}
\mathrm{E}[\text{ Work on } i\text{th insertion} \mid \text{elements of } S_i\ ] &= \frac{1}{i} \sum_{s \in S_i} \mathrm{E}[\text{ Work to insert } s \mid \text{elements of } S_i\ ] \\
&= \frac{1}{i} \sum_{s \in S_i} \sum_{f \in F_i(s)} O\big(n(f) + \ell(f)\big)
\end{aligned}$$

We now proceed to bound this expression. Observe that for each face f, there are at most four segments s such that $f \in F(s)$: the top segment, the bottom segment, and one per side if the side is defined by the extensions from a segment endpoint. Thus

$$\begin{aligned}
\mathrm{E}[\text{ Work on } i\text{th insertion} \mid \text{elements of } S_i\ ] &= O\Big(\frac{1}{i} \sum_{f \in F_i} \big(n(f) + \ell(f)\big)\Big) \\
&= O\Big(\frac{1}{i} \sum_{f \in F_i} n(f)\Big) + O\Big(\frac{1}{i} \sum_{f \in F_i} \ell(f)\Big)
\end{aligned}$$

Since the number of remaining line segments at the ith step is $n-i$, we must have $\sum_{f \in F_i} \ell(f) = n-i$. Therefore the second term in the expectation is $O(n/i)$.

We now consider the first term in the expectation. Since each vertex has degree at most six it appears in at most six faces, so $\sum_{f \in F_i} n(f) = O(\text{vertices determined by } S_i)$. Let k_i denote the number of intersections of the line segments in S_i. Then the number of vertices determined by S_i is $O(i + k_i)$. Letting S_i be a random subset of S, we obtain that

$$\mathrm{E}[\text{ Work on } i\text{th insertion }] = O((i + \mathrm{E}[\ k_i\ ])/i) + O(n/i)$$

Since S_i is randomly chosen, the probability that a given intersection point has both its segments in S_i is at most $(i/n)^2$. Thus $\mathrm{E}[\ k_i\ ] = O(k(i/n)^2)$. The expected amount of work to insert all n line segments is therefore

$$\sum_{i=1}^{n} \Big(O(1 + k(i/n^2)) + O(n/i) \Big) \;=\; O(n) + \frac{k}{n^2} \sum_{i=1}^{n} O(i) + O(n \log n) \;=\; O(k + n \log n)$$

18.1.5 Point Location

We have shown how to build the trapezoidal decomposition data structure. To address the problem of point location, we will construct an auxiliary data structure while constructing the trapezoidal decomposition. Call this auxiliary structure the **search structure**. The search structure is similar to a *kd*-tree in the sense that there are x-nodes that compare x-coordinates, y-nodes that compare y-coordinates, and the leaves correspond to regions of the plane (i.e., trapezoids). Whereas *kd*-trees have a unique path from the root to each region, our search structure may have multiple paths to any given region, so it is actually a DAG.

To perform a point location query with this structure, we start at the root. At each node of the tree we perform a comparison against the query point. The result of this comparison determines whether to proceed to the left or right child. Since the structure is acyclic, the query will eventually terminate at a leaf. This leaf node points to the trapezoid containing the query point.

The search structure is best understood by considering how it is constructed. Initially it contains a single leaf node, which points to a trapezoid containing the entire plane (or bounding box). The search structure is only updated when we split and merge trapezoids while building the trapezoidal decomposition.

When a trapezoid is split due to a new extension, we replace its leaf node with a x-node that compares against the x-coordinate of the extension. When a trapezoid is split due to a new line segment through it, we replace its leaf node with a y-node that compares whether a query point is above or below that line segment. If two trapezoids are merged, we replace their two leaf nodes with a single leaf node. All pointers that previously pointed to the original two leaves are updated to point to the single new leaf.

We now analyze the expected time required to perform a point location query with this structure. The query time is clearly linear in the length of the path followed from the root to the leaf. By linearity of expectation, the expected length of this path equals the expected number of nodes added to this path during the ith step of the construction algorithm. The ith step adds only $O(1)$ nodes to the search structure; these new nodes are on the query path with some probability p_i. The expected query time is therefore $\sum_{i=1}^{n} O(1) \cdot p_i = \sum_{i=1}^{n} O(p_i)$.

We can bound p_i via backwards analysis. Consider the data structure at the ith insertion step. The query point lies in some trapezoid of this structure. Observe that there are at most four line segments in S_i whose removal would cause this trapezoid to be split: the one on top, the one on the bottom, and the ones that determine the left and right sides, either by intersection points or extensions. Thus the probability that the line segment inserted in the ith step borders the query trapezoid is at most $4/i$. Thus $p_i \leq 4/i$. The expected query time is therefore $\sum_{i=1}^{n} O(4/i) = O(\log n)$.

18.1.6 Triangulating Non-convex Polygons

Given a trapezoidal decomposition of a non-convex polygon, it is possible to compute a triangulation in linear time. First, iterate over all trapezoids. If a trapezoid contains two non-consecutive vertices of the original polygon, connect them by an edge. The original polygon is now partitioned into polygons that are x-monotone, that is, the x-coordinates of the vertices on each polygon strictly increase along its top and bottom. Next, we can triangulate these x-monotone polygons as follows: for every convex vertex (i.e., one with interior angle $> 180°$), add an edge between its two neighbors.

18.2 Range Queries

Suppose we have a set of n points in d dimensions. A dimension may correspond to physical position or may correspond to some non-geometric notion, such as price, date, etc. A **range query** over these points asks which points lie inside in some axis-parallel rectangular box. For simplicity of exposition we assume that the points are in general position, so that no two coordinates are equal.

18.2.1 Range Queries in 1-d

The objective is to perform a range query over some interval $[x_1, x_2]$. A simple static solution would be to store the data in a sorted array. Binary search could then be used to search for x_1 and x_2 in the array. The array elements between x_1 and x_2 are the result of the query. Constructing the sorted array clearly takes $O(n \log n)$ time and a range query clearly takes $O(\log n + k)$ time where k is the size of the result set.

A simple dynamic solution would be to use a "skip list". Skip lists are like a sorted linked list that supports insertions, deletions and searches in expected $O(\log n)$ time. Range queries can be performed in the same manner as the sorted array. First search for x_1 and x_2. This takes expected $O(\log n)$ time. Then, since the elements are arranged into a linked list, a linear scan from x_1 to x_2 will find all query results in $O(k)$ time.

Another solution is to use a balanced binary search tree. We store each point in a unique leaf node. Each internal node contains the maximum value in its left subtree. When performing a range query over the interval $[x_1, x_2]$, we want to identify all leaf nodes between x_1 and x_2. To do this, we search from the tree root for x_1 and x_2. At some vertex v (possibly the root) the two search paths will diverge. Whenever the path from v to x_1 branches left, the right-subtree clearly contains leaves between x_1 and x_2. Similarly, whenever the path from v to x_2 branches right, the left-subtree contains leaves between x_1 and x_2. A moment's thought shows that *all* leaves between x_1 and x_2 are in fact contained in these subtrees.

The range query algorithm performs a depth-first search in these identified subtrees and outputs all points contained in their leaves. Identifying such subtrees clearly requires at most $O(\log n)$ time, since that is the length of the paths from the root to x_1 and x_2. The total time required for each depth-first search is linear in the number of leaves, since the trees are balanced. Thus the total time required is $O(\log n + k)$.

18.2.2 Range Queries in 2-d

The binary search tree approach for 1-d range queries can be extended to a structure for 2-d range queries. The structure consists of a binary search tree (called the **primary tree**) containing the x-coordinates of all points; as before, the points are in the leaves. For each internal node n in the primary tree, we create an associated **secondary structure** containing all points in the subtree beneath n, ordered by their y-coordinates.

Suppose we want to perform a range query over the region $[x_1, x_2]$, $[y_1, y_2]$ using this structure. We first perform a range query in the primary tree for the interval $[x_1, x_2]$. As described in the 1-d case, this operation identifies a set of internal nodes whose subtrees contain all points in the interval $[x_1, x_2]$. For each internal node, we perform a range query in its secondary structure over the interval $[y_1, y_2]$. This clearly identifies all points that lie in the region $[x_1, x_2]$, $[y_1, y_2]$.

We now analyze the space required by this structure. Every point is stored in exactly one leaf of the primary tree. For each ancestor of this leaf, the point must be stored in a secondary structure. Since we assume that the primary tree is balanced, each leaf has $O(\log n)$ ancestors. It follows that each point is stored in $O(\log n)$ secondary structures. Since the space used by each secondary structure is linear in the number of points that it contains, the total space required is $O(n \log n)$.

We now analyze the time required by a range query using this structure. Let k be the total number of query results. Let S be the set of secondary structures that we query. By our arguments in the 1-d case, $|S| = O(\log n)$. For any $s \in S$, let k_s denote the number of query results in tree s. Clearly s contains at most n points, so the time required to query s is $O(\log n + k_s)$. The total time required to query all $s \in S$ is therefore

$$\sum_{s \in S} O(\log n + k_s) \;=\; \sum_{s \in S} O(\log n) + \sum_{s \in S} O(k_s) \;=\; O(\log^2 n) + O(k)$$

Since it takes only $O(\log n)$ time to search the primary tree, the total time required for a range query using this structure is $O(\log^2 n + k)$.