

Homework Assignment 2

Sebastian Kanthak

November 25, 2003

Part A

1

a)

Let L be a set of horizontal line segments $[x : x'] \times y$. The search structure can be built by algorithm 1.

Algorithm 1 BuildLineRangeTree

Input: A set L of horizontal line segments

Output: Root of a range tree with interval trees as associated structure

Construct interval tree I on line segments L , ignoring their y -coordinates

if L contains only one segment **then**

 Create a leaf v store the line segment's y -coordinate and make I its associated structure

else

 Split L into two subsets L_{left} and L_{right} at the median y_{mid} of their y -coordinates

$v_{\text{left}} \leftarrow \text{BuildLineRangeTree}(L_{\text{left}})$

$v_{\text{right}} \leftarrow \text{BuildLineRangeTree}(L_{\text{right}})$

 Create a node v storing y_{mid} with left child v_{left} and right child v_{right} and store I as its associated structure.

end if

return v

The algorithm to construct the interval tree is almost exactly the same as on page 215 of our text book. It ignores the y -coordinates of the line segments so that it can work with 1-dimensional intervals but may store them for reporting.

To report all line segments intersecting a vertical segment $x \times [y : y']$ a range query on the range tree is used. Let $I(v)$ be the associated interval tree of a node v and T the range tree built by algorithm 1. Then all line segments intersecting the query segment can be reported with algorithm 2.

The QueryIntervalTree algorithm is exactly the same as the one found on page 216 of the text book. Instead of only reporting the intervals, the y -coordinate can be included to report full line segments.

Algorithm 2 QueryRangeLineTree

Input: A range-line tree as built by algorithm 1 and a query interval $x \times [y : y']$

Output: Report all segments intersecting the query interval

$v_{\text{split}} \leftarrow \text{FindSplitNode}(T, y, y')$

if $y_{v_{\text{split}}}$ is a leaf **then**

if $y_{\text{split}} \in [y : y']$ **then**

 QueryIntervalTree($I(v_{\text{split}}), x$)

end if

else

 (*Follow the path to y and query interval trees associated with subtrees right of the path* *)

$v \leftarrow lc(v_{\text{split}})$

while v is not a leaf **do**

if $y \leq y_v$ **then**

 QueryIntervalTree($I(rc(v)), x$)

$v \leftarrow lc(v)$

else

$v \leftarrow rc(v)$

end if

end while

 (*Check if the line segment stored at the leaf v must be reported* *)

if $y_v \in [y : y']$ **then**

 QueryIntervalTree($I(v), x$)

end if

 Similarly follow the path to y' and query all interval trees associated with roots of subtrees lying on the left of that path

end if

b)

To prove the correctness it is crucial to observe that all line segments stored in an associated structure $I(v)$ have y -coordinates that lie in the *canonical subset* of this node. That is, the y -coordinates are in the interval $[y_{\text{left}} : y_{\text{right}}]$ formed by the leftmost and rightmost child of v . This invariant is maintained by the construction algorithm of the range tree.

I will first prove that all line segments reported by the algorithm intersect the query segment $x_q \times [y : y']$. The QueryIntervalTree algorithm is only called for nodes whose *canonical subset* is a subset of the vertical query interval $[y : y']$. Thus all line segments reported by a QueryIntervalTree method contain the x -coordinate x_q and have a y -coordinate in the interval $[y : y']$. Thus they intersect the query segment.

Let otherwise l be a line segment intersected by the query segment. As its y coordinate lies in the vertical query interval it is contained in a *canonical subset* for visited by the range tree query algorithm (assuming that a range tree is proven correct). As it contains the x -coordinate x_q it will be reported by the QueryIntervalTree algorithm.

c)

As for the 2-dimensional range tree the key property is, that a line segment is stored only in associated structures of nodes lying on the path to the line segment's y -coordinate. This follows directly from the corresponding proof for the 2-dimensional range tree. Thus a line segment appears at most in $O(\log n)$ interval trees

The range tree without the associated interval trees can be constructed in $O(n)$ time if the line segments are sorted to their y -coordinates. An interval tree can be constructed in $O(n \log n)$ time. As a line segment appears at most once in each interval tree and at most in $O(\log n)$ interval trees the total time to construct the interval trees can be estimated as follows: Let V be the set of vertices in the range tree and n_v be the number of segments in the associated interval tree of $v \in V$. The total construction time for the interval trees is

$$O\left(\sum_{v \in V} n_v \underbrace{\log n_v}_{\leq \log n}\right) \leq O(\log n \sum_{v \in V} n_v) = O(n \log^2 n)$$

where the last step follows as every line segment appears only in $O(\log n)$ many nodes.

This subsumes the cost for sorting the line segments on y -coordinates and building the range tree, resulting in a total construction cost of $O(n \log^2 n)$.

As each interval tree requires a linear amount of memory and every line segment is in at most $O(\log n)$ associated structures, all interval trees require $O(n \log n)$ memory. Together with the $O(n)$ memory for the range tree, this gives a total of $O(n \log n)$ storage requirements.

For querying a search in the range tree reports $O(\log n)$ *canonical subsets* in $O(\log n)$ time. Querying each of these interval trees $I(v)$ requires another

$O(\log n + k_v)$ time, where k_v is the number of line segments reported in interval tree $I(v)$. This adds up to $O(\log^2 n + k)$ query time.

2

a)

As is shown in the text every interval is stored at most twice at every level of the segment tree (proof of Lemma 10.10). This implies that every segment is stored in at most $O(\log N)$ spanning segment lists. With a similar argument, we can show that every red segment is stored in at most two list partial spanning segment lists at every level of the tree.

Proof: Assume that a line segment is stored in three partial spanning segment lists at the same level. Let R_v^* be the list of the node lying in the middle. As the segment is stored in R_v^* it cannot span the entire canonical interval of v . However, as v is in the middle and the line segment intersects the canonical intervals of the left and right node, it must completely span the canonical interval of v , which leads to a contradiction. ζ

As the height of the tree is $O(\log N)$ we get that every line segment is stored in at most $O(\log N)$ partial spanning segment lists, leading to a total size of $O(N \log N)$.

b)

To count the number of intersections, I walk through the red segments in vertical order and maintain the uppermost and lowermost blue segment intersected by the current red segment. When moving to the next red segment, these upper and lower bounds can be updated by only increasing them (or keeping them). As all pointers are only increased during the algorithm, the running time is clearly $O(|R_u| + |R_b|)$.

To merely count the intersections, the size of the interval of blue segments intersecting a red segment can simply be added up. If the intersections have to be reported as well, one needs to visit all blue segments in this range for every red segment. As there is an intersection for all those pairs, this adds another $O(I_u)$ summand to the time requirements.

c)

I will only regard intersections of segments stored in R_u with segments stored in B_u^* . The other case can be handled symmetrically. For every segment in B_u^* we can locate the first and the last segment in R_u that is intersected by the blue segment with a binary search in $O(\log n)$ time. Thus, the number of intersections can be reported by calculating the difference of the position of these two segments. This requires a total time of $O(|B_u^*| \log |R_u| + |R_u^*| \log |B_u|)$. Reporting all intersections is easy as all line segments between the the lower and upper bound will intersect the blue segment, resulting in an additional $O(I_u^*)$ summand.

d)

For a given intersection it is clear that it can only be reported at nodes whose canonical interval contains the point of intersection. These nodes form a path from a leaf to the root in the segment tree, so we can restrict our attention to these nodes.

I will first show, that an intersection is reported at most once. Suppose an intersection is reported at some node u . W.l.o.g. we can assume that the red segment is stored in the list R_u as one of the two segments has to be stored in a spanning segment list for an intersection to be reported. This red segment cannot be stored in any child node of u , thus the intersection will not be reported at any child node. Nor can it be stored in a spanning segment list $R_{u'}$ of a parent node. The blue segment is either stored in the spanning segment list B_u (*wide-wide*) or in the partial spanning segment list B_u^* (*wide-narrow*). In either case it can only appear in partial spanning segment lists $B_{u'}$ at parent nodes u' of u . As we never checked for *narrow-narrow* intersections, the intersection will not be reported in any parent node either.

To show that every intersection is reported, take one intersection. Following the path of the leaf containing the intersection point to the root, we will find nodes u_r and u_b s.t. the segments are stored in R_{u_r} and B_{u_b} . If $u_r = u_b$ it will be reported as a *wide-wide* intersection at this node. Otherwise take the node closer to the root. W.l.o.g. let this be u_r , then the blue segment will be stored in $B_{u_r}^*$ and the intersection will be reported as a *wide-narrow* intersection.

e)

The segment tree contains $O(N)$ nodes. In every node, we have to search for *wide-wide* and *wide-narrow* intersections. The time for *wide-wide* intersections can be estimated as follows:

$$\sum_u |R_u| + |B_u| = O(N \log N)$$

as the size of all spanning lists is bound according to a) and using the time requirements from b).

The time for *wide-narrow* intersections can be estimated as follows, using the bounds from a) and c).

$$\sum_u |R_u^*| \log |B_u| + |B_u^*| \log |R_u| \leq \sum_u \log N (|R_u^*| + |B_u^*|) = O(N \log^2 N)$$

This subsumes the cost of $O(N \log N)$ required to build the segment tree and maintain the spanning segment lists and the partial spanning segment lists.

2

a)

To capture that S_j is chosen randomly I first define a random variable $X_{a,b}$ that is one if the line segments a and b are in S_j . Furthermore I define a function

$\sigma(a, b)$ that is one if the line segments a and b intersect. So we have:

$$\begin{aligned} X_{a,b} &= \begin{cases} 1 & \text{if } a, b \in S_j \\ 0 & \text{otherwise} \end{cases} \\ \sigma(a, b) &= \begin{cases} 1 & \text{if } a \text{ and } b \text{ intersect} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Note that $E(X_{a,b}) = \frac{j(j-1)}{n(n-1)}$ as S_j is randomly chosen. Now we can compute the expectation for I_j as follows:

$$\begin{aligned} E(I_j) &= E\left(\sum_{\{a,b\} \subseteq S} X_{a,b} \sigma(a, b)\right) \\ &= \sum_{\{a,b\} \subseteq S} E(X_{a,b}) \sigma(a, b) \\ &= \frac{j(j-1)}{n(n-1)} \underbrace{\sum_{\{a,b\} \subseteq S} \sigma(a, b)}_{=I} \\ &= O\left(I \frac{j^2}{n^2}\right) \end{aligned}$$

b)

Updating the trapezoidal decomposition can be done by following the newly inserted line segment and updating trapezoids by merging and splitting trapezoids. Let k_j be the number of new trapezoids inserted into the decomposition in the j th step, that is the number of trapezoids that are in Δ^j but not in Δ^{j-1} . Then the update cost of the j th insertion is proportional to k_j . Define $\sigma(s, T)$ for a line segment s and a trapezoid T as follows:

$$\sigma(s, T) = \begin{cases} 1 & \text{if } s \text{ defines } T \text{ by bounding one of } T\text{'s edges} \\ 0 & \text{otherwise} \end{cases}$$

As every line segment $s \in S_j$ was equally likely to be the j th insertion, we can compute $E(k_j)$ by taking the average over the cost for inserting every line segment $s \in S_j$.

$$\begin{aligned} E(k_j) &= \frac{1}{j} \sum_{s \in S_j} \sum_{T \in \Delta^j} \sigma(s, T) \\ &= \frac{1}{j} \sum_{T \in \Delta^j} \sum_{s \in S_j} \sigma(s, T) \\ &= \frac{|\Delta^j|4}{j} \end{aligned}$$

as every trapezoid is still bounded by at most four line segments. The expected size of the j th trapezoidal decomposition Δ^j can be estimated with Euler's relation for planar graphs.

The number of vertices in Δ^j is proportional to the number of line segments j plus the number of intersections in S_j as it can be thought of as a decomposition of non-intersecting line segments, where segments are split into two segments at intersections. As the number of edges is bound linearly by the number of vertices it follows, that the number of faces (= trapezoids) is proportional to the number of vertices. However, as S_j is randomly chosen¹, the expected number of intersections is $O(I \frac{j^2}{n^2})$. Thus we have $E(|\Delta^j|) = O(j + I \frac{j^2}{n^2})$ and we get:

$$E(k_j) = O(1 + I \frac{j}{n^2})$$

c)

The total building cost is composed of locating n segment endpoints in the trapezoidal decomposition and inserting the n segments. The expected cost for the n insertions is

$$O(\sum_{j=1}^n 1 + I \frac{j}{n^2}) \leq O(\sum_{j=1}^n 1 + I \frac{1}{n}) = O(n + I)$$

To locate query points q in the trapezoidal decomposition Δ^j a tree-like structure similar to the non-intersecting case is maintained. Insertion of new segments, however, requires some new cases, where line segments intersect other line segments. These cases can be detected in constant time by simply following the line segment to be inserted, once an endpoint has been located. Furthermore, cases where a line segments intersects another line segment can be treated as if a shorter line segment (ending at the intersection point) is inserted, reducing it to a known cases. As the next change to the tree happens in another triangle, it is still valid that the height of the tree only increases by a constant amount, giving rise to a similar argument.

Let q be an arbitrary query point and let $\Delta^j(q)$ denote the trapezoid containing q in Δ^j . Define X_j as

$$X_j = \begin{cases} 1 & \text{if } \Delta^j(q) \neq \Delta^{j-1}(q) \\ 0 & \text{otherwise} \end{cases}$$

As before we have $\text{pr}(X_j = 1) \leq \frac{4}{j}$ as the trapezoid only changes if it is defined by the line segment inserted in the j th insertion and there are at most 4 line segments defining this trapezoid and all segments are equally likely to be

¹I assume this although the question says "conditional on a fixed S_j ". For the analysis of the algorithm, it is certainly true that S_j is randomly chosen as every possible subset of size j is equally likely to be S_j

the j th insertion. The search cost for q is $\leq 3 \sum_{j=1}^n X_j$, so the expected search cost c_q for an arbitrary query point q is as follows:

$$E(c_q) = O\left(\sum_{j=1}^n E(X_j)\right) \leq O\left(\sum_{j=1}^n \frac{4}{j}\right) = O(\log n)$$

So the expected search cost for n points is $O(n \log n)$ giving a total expected construction cost of $O(I + n \log n)$.

d)

As described in c) intersections are found while inserting line segments by simply following the inserted line segments. It is obvious, that all intersections are found during the construction of the trapezoidal decomposition thus the expected cost of reporting all I intersecting pairs among a collection of n line segments is $O(I + n \log n)$.

Part B

1

I first observe, that a face F is horizontally convex if and only if the intersection with any vertical line is a contiguous line segment as a lemma.

Lemma 1 *A face f is horizontally convex if and only if the intersection with any vertical line is a contiguous (possibly empty) line segment.*

Proof: Suppose f is horizontally convex. If it is bounded by two x -monotone chains any vertical line intersects the boundary at most twice, thus the intersection of such a line with f is contiguous. On the other hand, if any vertical line intersects f in a contiguous line segment it can intersect the upper and lower chain bounding f at most once. Thus both chains are x -monotone.

a)

W.l.o.g. suppose, that a vertex v is not adjacent to any vertex on its left as shown in figure 1. Let u and l be the uppermost and lowermost edges leaving v . As there are no other edges above or below u and l the face above u and the face below l are the same face A . This face, however, is not horizontally convex, as there is a vertical line (shown dashed) in the sense of lemma 1 whose intersection with A is not contiguous.

Suppose on the other hand that a framed subdivision is not horizontally convex. Then there must be a vertical line intersecting a face A in a non-contiguous way as shown in figure 2 by lemma 1. Let u and l denote the upper and lower endpoints of the vertical line's interval that do not intersect A . As A is a connected face, we can assume w.l.o.g. there is a left-most vertex v connected to u and l with an x -monotone chain of edges. This vertex does not have any edges going to the left.

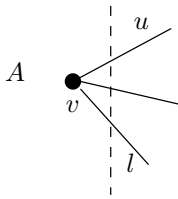


Figure 1: vertex v not adjacent to any vertex on its left

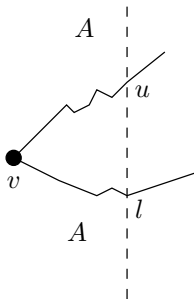


Figure 2: face A not horizontally convex

b)

I first correct all vertices having no vertex adjacent to its left by sweeping a vertical line from the left to the right maintaining the invariant that all vertices to its left have at least one adjacent vertex to its left (except for the leftmost point). I then repeat this process with a symmetrical sweep from right to left to correct vertices having no adjacent vertex to their right.

While sweeping the line I keep a horizontally order list of points seen so far for every face. Once the sweep line finds a vertex v having no adjacent vertex to its left, we can check vertices on the face left of v to find a vertex v' visible from and lying left of v . Such a vertex v' must exist as v cannot be one of the three leftmost points. By adding an edge between v and v' to the subdivision, v has now an a vertex adjacent to its right. Using the sorted lists maintained during the sweep such a vertex can be found in constant time as it has to be among the 4 rightmost vertexes seen so far on this face. So including sorting the points, the sweeping algorithm takes $O(n \log n)$ time.

c)

I first define a face a to be below b (written as $a \sqsubset b$) if they share an edge that lies on the upper chain of a and the lower chain of b . I then define \leq to be the reflexive and transitive closure of \sqsubset . I claim that \leq is a partial order. As \leq is reflexive and transitive by definition it remains to show that it is anti-symmetric.

To prove this, I first need some notation and a lemma. For a horizontal

convex face f , let l_f and u_f denote its lower and upper chain respectively.

Lemma 2 *Let a and b be two horizontal convex faces with $a \sqsubset b$. There exists the horizontal convex closure c of $a \cup b$ and it holds that $l_a \subseteq l_c$ and $u_b \subseteq u_c$.*

Proof: Horizontal convex faces with intersection form a hull system. The full space is horizontal convex and the intersection of horizontal convex faces is horizontal convex². Thus the horizontal convex closure of $a \cup b$ is well-defined and as a and b share an edge it consists of one connected face. Furthermore, no part of b can be vertically below l_a as $a \sqsubset b$, giving us $l_a \subseteq l_c$. By duality it follows that $u_b \subseteq u_c$.

I can now prove the anti-symmetry of \leq . Assuming that $a \leq b$ and $b \leq a$ I have to show that $a = b$. Assume otherwise. Then there is a sequence of faces s.t. $a \sqsubset c_1 \sqsubset \dots \sqsubset c_n \sqsubset a$. Using lemma 2 I can merge c_1, \dots, c_n into a horizontally convex face c . Now c shares an edge with a on its lower and its upper chain which is a contradiction to c being horizontally convex.

As every partial order can be extended to a total order we can now assign numbers $1, \dots, f$ to the faces according to such a total order. This total order respects the *below*-relation as the relation \sqsubset is a subset of the \leq relation.

d)

The \sqsubset relation from c) induces a directed acyclic graph on the faces. This graph can be built in linear time by inspecting all edges of the subdivision and introducing an appropriate directed edge into the graph. The graph is acyclic as the \sqsubset relation can be extended to a partial order as shown in c). Now, the numbering can be obtained in linear time by applying a standard *topsort*-algorithm on the graph.

e)

Assume there would be an edge on a $chain_i$ violating x -monotonicity as shown in figure 3. According to the assumption that $chain_i$ separates faces numbered $\leq i$ from faces numbered $> i$ the face a on its right side³ must have a lower number than face b on its left side. Since $b \sqsubset a$, however, b will have the smaller number. The case of an edge going from bottom-right to top-left leads to a contradiction in a symmetrical way.

f)

As $chain_i$ is x -monotone one can do a binary search to determine in which vertical slab q lies. Once this slab has been found it can be determined in constant time on which side of $chain_i$ q lies. As a chain can have at most n edges, the running time is clearly $O(\log n)$.

²Although it is not connected in general

³“right” is defined by the direction of the chain, here

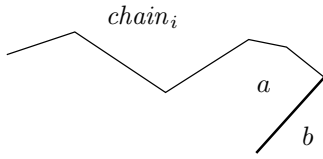


Figure 3: assuming $chain_i$ as being not x -monotone

g)

Within a node a binary search for the vertical slab is performed as before. If the vertical slab is defined by an edge stored in this node, the side q lies on can be determined in constant time. If the slab is defined by a *gap*, however, things are different. As the edges corresponding to the *gap* are stored in an ancestor it has already been computed if q lies above or below those edges. Thus, while searching the tree we can always keep track of the edges q has already been compared to. If we locate q within a gap, we check all edges visited so far, which are at most $O(\log n)$ many. For each of these edges, we still have to check if it really lies on the current chain. This can be done in constant time, if we store with every edge the lowest and highest index of chains it lies.

To optimize this even further, while going down the tree, we only need to keep the highest edge above which q lay and the lowest edge below which q has been located, as the current chain will always lie between those edges. Thus, if q hits a gap, only a constant number of edges has to be checked.

h)

As suggested in the question I augment the list bottom-up with a fixed fraction of elements from the children. These elements, however, do not have to be “fit into” the list stored at the node. As I am only interested in the vertical slabs, I only consider the x -interval defined by the elements. If an element to be inserted overlaps with an element currently in the list, I simply split the current element of the list at the x -coordinates of the inserted element, so that the list can maintain a horizontal order of the elements.

I can now store references with every element pointing to elements in both children. Such a reference points to the closest element to the left, that has been pulled-up from the child.

The search for a query point can now be done as follows. At the root node, a binary search is done as before. Instead of doing binary searches in the children, a linear search to the right is done in the children, starting at the elements pointed to by the newly added references. As a fixed fraction of elements of the child has been inserted into the parent, the linear search only requires a constant number of steps at every level.

The number of additional elements is bounded by a geometric sum: Every fourth of the original lists is pulled-up one level. Of those, another quarter is pulled-up another level and so on. As the height of the tree is logarithmic, an

element can be pulled-up at most $\log n$ time. So the size of the augmented tree n' can be estimated as follows:

$$n' \leq \sum_{i=0}^{\log n} \frac{n}{4^i} = O(n)$$

So, the total size of the data structure does not increase asymptotically. Augmenting the tree takes $O(n \log n)$ time as the insertion point for every element can be found in logarithmic time with a binary search. The query time consists of $O(\log n)$ for the initial binary search and doing a constant time linear search on every level (of which there are at most $O(\log n)$) leading to a total query time of $O(\log n)$.

Collaboration

I discussed the problems with Wolfgang Hess, Markus Moll and, to a lesser degree, with Mik Kerstens. Besides our text book I did not use any other sources⁴.

⁴I guess I should add that I shared my LaTeX experience with James King and got a lot of useful hints from Markus Moll. :-)