Exercise 2

for the lecture

Computational Geometry

Dominik Bendle, Stefan Fritsch, Marcel Rogge and Matthias Tschöpe

November 28, 2018

Exercise 1 (Point between Line Segments)

Given a balanced binary tree which lexicographically stores a tuple (x_{hi}, x_{lo}) for each line segment, where x_{hi} and x_{lo} are the x coordinate for the upper and lower end points.
Find the two disjunct line segments that define the region in which the query point $p \in D$ is:

```
node = tree.root
left_line_segment = null
right_line_segment = null

for i = 0; i < tree.height; i++
    if node.x_hi < p_x && node.x_lo < p_x
        left_line_segment = node
        node = node.right_child
    else if node.x_hi >= p_x && node.x_lo >= p_x
        right_line_segment = node
        node = node.left_child
    else
        position = (x_lo − x_hi) ∗ (p_y − 1) − (−1) ∗ (p_x − x_hi)
        if position > 0
            left_line_segment = node
            node = node.right_child
        else
            right_line_segment = node
            node = node.left_child
```

Query point p is in the region between[1] the two lines segments:
$\overline{(left\_line\_segment.x_{hi}, 1) \;\; (left\_line\_segment.x_{lo}, 0)}$
$\overline{(right\_line\_segment.x_{hi}, 1) \;\; (right\_line\_segment.x_{lo}, 0)}$

[1]If a point lies exactly on a line, it is treated as if the point lies in the region to the left of that line.

The algorithm does not work for arbitrary line segments.
Arbitrary line segments can have intersections which causes multiple problems. For example could the two line segments, that define the region in which p lies, be on different branches. This would require to traverse the tree two times. At the same time, could the region, in which the point p lies, be defined by more than two line segments. This would mean that the tree would then have to be traversed an unknown amount of times.

Exercise 2 (Line Segment Intersection)

$Q = tree(p_1, p_2, p_3, q_1, q_3, p_4, q_2, p_5, q_5, q_4);\ T = tree()$

$p = p_1, U = \{s_1\}, L = \emptyset, C = \emptyset$
$|L \cup U \cup C| > 1$ is false
no deletion, $T.insert(s1) \Rightarrow T = tree(s_1)$
$U \cup C = \{s_1\}$
$s' = s_1, s_l = null$
FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.
$s'' = s_1, s_r = null$
FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r do not intersect below the current sweep line.
$\Rightarrow Q = tree(p_2, p_3, q_1, q_3, p_4, q_2, p_5, q_5, q_4)$
intersection_points $= \emptyset$

$p = p_2, U = \{s_2\}, L = \emptyset, C = \emptyset$
$|L \cup U \cup C| > 1$ is false
no deletion, $T.insert(s_2) \Rightarrow T = tree(s_2, s_1)$
$U \cup C = \{s_2\}$
$s' = s_2, s_l = null$
FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.
$s'' = s_2, s_r = s_1$
FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r intersect below the current sweep line in p_{s_1,s_2}.
$\Rightarrow Q.insert(p_{s_1,s_2})$
$\Rightarrow Q = tree(p_{s_1,s_2}, p_3, q_1, q_3, p_4, q_2, p_5, q_5, q_4)$
intersection_points $= \emptyset$

$p = p_{s_1,s_2}, U = \emptyset, L = \emptyset, C = \{s_1, s_2\}$
$|L \cup U \cup C| > 1$ is true $\Rightarrow p = p_{s_1,s_2}$ is intersection of s_1 and s_2
$T.delete(s_1),\ T.delete(s_2),\ T.insert(s_1),\ T.insert(s_2)\ \Rightarrow T = tree(s_1, s_2)$
$U \cup C = \{s_1, s_2\}$
$s' = s_1, s_l = null$
FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.
$s'' = s_2, s_r = null$
FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r do not intersect below the current sweep line.
$\Rightarrow Q = tree(p_3, q_1, q_3, p_4, q_2, p_5, q_5, q_4)$
intersection_points $= \{p_{s_1,s_2}\}$

$p = p_3, U = \{s_3\}, L = \emptyset, C = \emptyset$

$|L \cup U \cup C| > 1$ is false

no deletion, $T.insert(s_3) \Rightarrow T = tree(s_3, s_1, s_2)$

$U \cup C = \{s_3\}$

$s' = s_3, s_l = null$

FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.

$s'' = s_3, s_r = s_2$

FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree(q_1, q_3, p_4, q_2, p_5, q_5, q_4)$

intersection_points $= \{p_{s_1,s_2}\}$

$p = q_1 = q_3, U = \emptyset, L = \{s_1, s_3\}, C = \emptyset$

$|L \cup U \cup C| > 1$ is true $\Rightarrow p = q_1 = q_3$ is intersection of s_1 and s_3

$T.delete(s_1), T.delete(s_3)$, no insertion $\Rightarrow T = tree(s_2)$

$U \cup C = \emptyset$

$s_l = null, s_r = null$

FindNewEvent$(s_l, s_r, p) \Rightarrow s_l$ and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree(p_4, q_2, p_5, q_5, q_4)$

intersection_points $= \{p_{s_1,s_2}, q_1 = q_3\}$

$p = p_4, U = \{s_4\}, L = \emptyset, C = \emptyset$

$|L \cup U \cup C| > 1$ is false

no deletion, $T.insert(s_4) \Rightarrow T = tree(s_4, s_2)$

$U \cup C = \{s_4\}$

$s' = s_4, s_l = null$

FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.

$s'' = s_4, s_r = s_2$

FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree(q_2, p_5, q_5, q_4)$

intersection_points $= \{p_{s_1,s_2}, q_1 = q_3\}$

$p = q_2 = p_5, U = \{s_5\}, L = \{s_2\}, C = \emptyset$

$|L \cup U \cup C| > 1$ is true $\Rightarrow p = q_2 = p_5$ is intersection of s_2 and s_5

$T.delete(s_2), T.insert(s_5) \Rightarrow T = tree(s_4, s_5)$

$U \cup C = \{s_5\}$

$s' = s_5, s_l = s_4$

FindNewEvent$(s_l, s', p) \Rightarrow s_l$ and s' do not intersect below the current sweep line.

$s'' = s_5, s_r = null$

FindNewEvent$(s'', s_r, p) \Rightarrow s''$ and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree(q_5, q_4)$

intersection_points $= \{p_{s_1,s_2}, q_1 = q_3, q_2 = p_5\}$

$p = q_5, U = \emptyset, L = \{s_5\}, C = \emptyset$

$|L \cup U \cup C| > 1$ is false

$T.delete(s_5)$, no insertion $\Rightarrow T = tree(s_4)$

$U \cup C = \emptyset$

$s_l = null, s_r = null$

FindNewEvent$(s_l, s_r, p) \Rightarrow$ s_l and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree(q_4)$

intersection_points $= \{p_{s_1,s_2}, q_1 = q_3, q_2 = p_5\}$

$p = q_4, U = \emptyset, L = \{s_4\}, C = \emptyset$

$|L \cup U \cup C| > 1$ is false

$T.delete(s_4)$, no insertion $\Rightarrow T = tree()$

$U \cup C = \emptyset$

$s_l = null, s_r = null$

FindNewEvent$(s_l, s_r, p) \Rightarrow$ s_l and s_r do not intersect below the current sweep line.

$\Rightarrow Q = tree()$

intersection_points $= \{p_{s_1,s_2}, q_1 = q_3, q_2 = p_5\}$

Exercise 3 (Pyramids Skyline)

We use a divide-and-conquer approach to compute the pyramid skyline efficiently, which follows the concept of merge sort. The following algorithm operates on sets of skylines (which are themselves assumed to be sets of line segments), so we have to convert the pyramid representations to skylines: But this is easy: if a pyramid is represented by the top corner-coordinate (x, y), then this yields the skyline $\{\overline{(0, y-x), (x, y)}, \overline{(x, y), (0, y+x)}\}$.

As we see in the above pyramid skyline, the assumption is that the x-axis is the horizon and every pyramid has topmost corner with positive y-coordinate. In particular, every skyline constructed from such pyramid skylines will start and end with a point with coordinate $y = 0$.

Before applying the following algorithm, we need to (potentially) simplify the set of pyramids: We want to ensure that no pyramids share a left or right side, as this will create difficult special cases in the following computations

This is easy: Sort the set of pyramids (possible in $\mathcal{O}(n \log n)$ lexicographically with respect to the key (x-coordinate of left corner, y-coordinate of top corner). We can then iterate through this set and compare x-coordinates of the left corners of subsequent pyramids: if they are equal, then we remove the one with lower top coordinates as it will be fully contained in the other pyramid and thus cannot contribute to the skyline. This can clearly be done in $\mathcal{O}(n)$. We can do the same with the right coordinate contributing another $\mathcal{O}(n \log n)$.

Algorithm 1 SKYLINE(T)

Input: T the set of individual skylines
Output: S the combined skyline

```
1: l := length(T)
2: if l < 2 then
3:     return T
4: end if
5: Partition T into two sets T_1, T_2 of (almost) same size, (e.g. split in the middle)
6: fst = Skyline(T_1)
7: snd = Skyline(T_2)
8: return MergeSkylines(fst, snd)
```

This is the outer part of the algorithm, the actual computations happen in MERGESKYLINES. We give an informal description:

Assume we merge two skylines T_1, T_2 which consist of $|T_1| = n$, $|T_2| = m$ line segments respectively. In the following, a line segment will *start* at its left end and *end* at its right end, since the algorithm will traverse both skylines from left to right. We can assume that the skylines are sorted from left to right.

1. Ensure that the first (i.e. left-most) line segment in T_1 starts at lower x-coordinate than the first one of T_2. If this is not the case, rename the sets accordingly.

   Note that, by the preprocessing, T_1 and T_2 cannot start with the same x-coordinate.

2. If the end point of the last (right-most) line segment of T_1 has x-coordinate less than or equal to the x-coordinate of the start point of the first line segment of T_1, then there is nothing to do as the skylines are disjoint, so return $S = T_1 \cup T_2$.

3. Initialize the new skyline $S = \emptyset$.

4. Move the first line segment from T_1 to S (by the choices above it is the left-most and must be in the skyline)

5. While $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$ do the following, where t_1 and t_2 shall be the first line segments in T_1 and T_2 respectively.

   a) If the endpoint of t_2 lies left of the starting point of t_1 (by construction thus far the end point of the skyline S), then remove t_2 as it cannot occur in the skyline and restart the loop.

   b) Otherwise check if t_1 and t_2 intersect:

      - If they do intersect, then they cannot be collinear (by preprocessing) and must intersect in a point x: Then denote by $t_{1,x}$ the line segment from the start of t_1 to x and by $t_{2,x}$ the line segment from x to the end of t_2. But $t_{2,x}$ cannot be a point (again by preprocessing: if some line segments only meet in an endpoint of one, then we must also have collinear intersecting line segments which is impossible) and by construction all line segments from T_2 thus far must lie below the constructed skyline, so $t_{1,x}$ and $t_{2,x}$ form a "valley". We do the following
        - Add $t_{1,x}$ to S,
        - Remove t_1 from T_1,
        - Replace t_2 in T_2 with $t_{2,x}$
        - Swap T_1 and T_2. This is necessary as such an intersection means we switch over from one skyline to another.

        Then restart the loop.

      - If they do not intersect then there are again two cases:
        - The endpoint of t_2 lies right of the endpoint of t_1. Then no other element in T_2 can intersect with t_1 and hence t_1 must be part of the skyline. Hence move t_1 from T_1 to T_2 and restart the loop.

          Note that t_2 might still intersect other elements of T_1.
        - If the endpoint of t_2 lies left of the endpoint of t_1, then t_2 cannot intersect any element in T_1. Hence t_2 cannot lie on the combined skyline, so remove it from T_2 and restart the loop.

6. If elements in T_2 remain, they must be disjoint from the constructed skyline S so we simply return $S \cup T_2$.

In less formal terms, the above algorithm "walks along" T_1 until it intersects T_2, then continues on T_2 and so on until the skyline is complete. Further, for each iteration of the while loop (5.) we discard one element of T_1 or T_2 and all the checks we need to do are clearly possible in constant time, so MergeSkylines has runtime in $\mathcal{O}(m+n) = \mathcal{O}(|T_1| + |T_2|)$. Also, each line segment in T_1 or T_2 can intersect at most one line segment from the other set by the structure of skylines,

so the amount of line segments in the new skyline can be at most $m + n$ as well. Hence, if T is the complexity function for Skyline, then we obtain the following recurrence relation

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \mathcal{O}(\text{MergeSkylines}) \\ &= 2T\left(\frac{n}{2}\right) + \mathcal{O}\left(\frac{n}{2} + \frac{n}{2}\right) \\ &= 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \end{aligned}$$

By the Master Theorem we now know that $T \in \mathcal{O}(n \log n)$, so, together with the preprocessing, the algorithm has a total complexity of $\mathcal{O}(n \log n)$.