ICS 621: Analysis of Algorithms	Fall 2019

Lecture 12: Interval Trees, Segment Trees

Prof. Nodari Sitchinava — *Scribe: Mojtaba Abolfazli, Muzamil Yahia*

1 Range queries: Cont

2 Interval trees for orthogonal stabbing queries

In certain geometric applications like planar graphs, we might be interested in reporting the set of line segments L associated with a window query $w = [x_0, x_1] \times [y_0, y_1]$. This set L contains all line segments with at least one end point inside w, or those segments that pass through the window w.

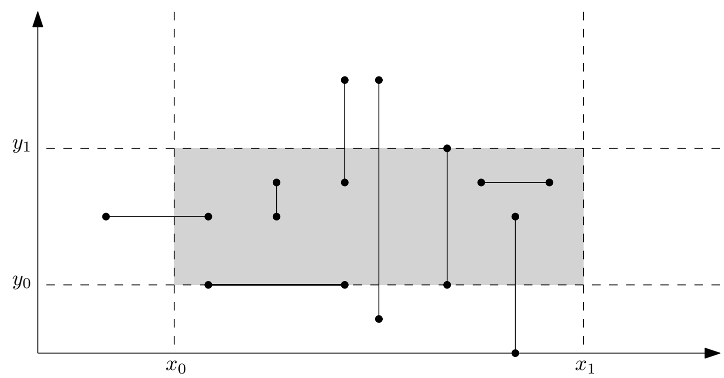


Figure 1: Query $w = [x_0, x_1] \times [y_0, y_1]$.

We can report all line segments with at least one endpoint inside w using algorithms and techniques from Section 1. We can use 2 instances of 3-sided query i.e. $q_0 = [x_0, x_1] \times [y_0, +\infty]$ and $q_1 = [x_0, x_1] \times [-\infty, y_1]$, and then report the intersections of the two queries in time $O(\log n + k_0) + O(\log n + k_1)$. However, in the worst case scenario, the two queries may report all $n = k_0 + k_1$ endpoints while the intersection may contain only one element. By constructing 2D range trees with storage $O(n \log n)$, and time $O(\log^2 n + k)$ we can report all line segments with ends points in w. Applying fractional cascading can reduce the query time further to $O(\log n + k)$ which is faster than $(\log n + n)$.

What is left is how to handle efficiently those line segments that pass through our window query w? In other words, we are looking for line segments that stab two boundaries of our window query w. If we know how to report all line segments that stab one boundary, say $[x_0, x_1]$, of w, we can easily check if they stab any of the remaining boundaries. In this section, we present a data structure that can handle stabbing queries for orthogonal line segments in $O(\log n + k)$ time, and in the next section, we show how to report stabbing queries for slanted line segments in $O(\log n + k)$.

Stabbing query: Given a set of intervals $I = \{i_1, \ldots, i_n\}$, and a vertical line query $\ell(x = t)$, find all intervals that are stabbed by ℓ.

We apply divide and conquer strategy to solve this problem. We construct recursively a balanced BST over all end points of intervals in I, i.e. $E_I = \bigcup_{[i_l, i_r] = i \in I} \{i_l, i_r\}$. While doing so, we augment this BST with extra space at each node and fill this space with appropriate intervals from I as shown in Algorithm 2.

Algorithm 1

function $\textsc{BuildInt}(I)$
  $v.val = median(E_I)$ ▷ key stored at node v
  $I_{mid} \leftarrow \{i \in I \mid i_l \leq v.val \leq i_r\}$
  $I_{left} \leftarrow \{i \in I \mid i_r < v.val\}$
  $I_{mid} \leftarrow \{i \in I \mid v.val < i_l\}$
  $v.data \leftarrow I_{mid}$ ▷ augmented data
  $v.left \leftarrow \textsc{BuildInt}(I_{left})$
  $v.right \leftarrow \textsc{BuildInt}(I_{right})$
  return v.

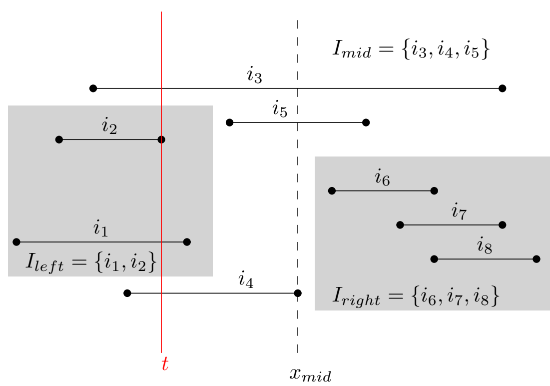


Figure 2: $I = I_{left} \cup I_{mid} \cup I_{right}$, and $\textsc{Stab}(I, t) = \{i_3, i_4, i_1, i_2\}$.

Figure 2 shows 8 line segments (intervals) with median being the right end point of interval i_4. The root of the interval tree divides the set of intervals $I = \{i_1, \ldots, i_8\}$ into 3 disjoint sets $I_{mid}, I_{left}, I_{right}$, where the root contains $I_{mid} = \{i_3, i_4, i_5\}$. The left subtree will recursively store $I_{left} = \{i_1, i_2\}$ and the right subtree will recursively store $I_{right} = \{i_6, i_7, i_8\}$.

How to store I_{mid}? In order to retrieve all intervals that are stabbed by $q = t$, we can scan the list of intervals stored at node v for stabbed intervals before going to the left or right subtree depending on whether t less than or greater than $v.val$. However, it may happened that all n intervals are stored in the root, and checking for stabbed intervals may take $O(n)$ then instead of $O(k)$, the output size!

In order to prevent that, we store I_{mid} in a more efficient way by storing I_{mid} sorted by the start point in an increasing order, and another copy of I'_{mid} of I_{mid} ordered by the end point in a decreasing order. To check if $q = t$ stabs some intervals in v, we compare t with $v.val$ and accordingly scan the list I_{mid} or I'_{mid} till we find an interval that does not intersect q. In Figure 2, at the root of the tree, $I_{mid} = \{i_3 \leq i_4 \leq i_5\}$ and $I'_{mid} = \{i_3 \geq i_5 \geq i_4\}$. The stabbing query would then run over I_{mid} and would have $i_3 \leq i_4 \leq t < i_5$.

Algorithm 2

function QueryInt(v, t)
    if $t == v.val$ or $I_{mid} = \phi$ **then** return I_{mid}
    if $t < v.val$ **then**
        return $\{i \in I_{mid} \mid i_l \leq t\} \cup$ QueryInt$(v.left, t)$
    else
        return $\{i \in I'mid \mid i_r \geq t\} \cup$ QueryInt$(v.right, t)$
    return v.

Since, we store only 2 copies of each interval i in the entire tree, interval trees takes $O(n)$ space. To report all stabbed intervals at node v, we need $k_v + 1$ comparisons. The recurrence relation is then is given by

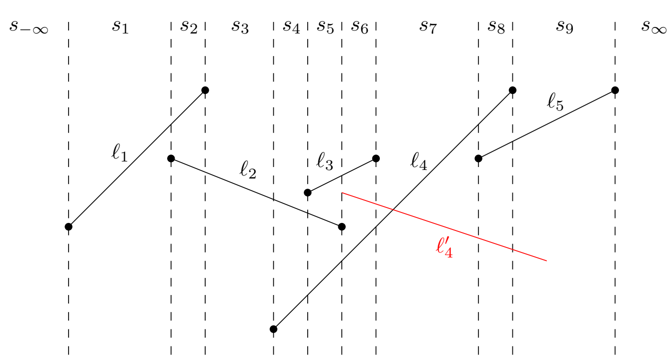
$$\begin{aligned} T(v) &= O(1 + k_v) + \max\{T(v.left), T(v.right)\} \\ &= O(\log n + k). \end{aligned}$$

The preprocessing time takes sorting complexity, as we need to build a balanced BST, sort all intervals by start point, and again by end points. We can write the recurrence relation of BuiltInt as follows:

$$\begin{aligned} T(n, h) &= O(n) + T(I_{left}, h - 1) + T(I_{right}, h - 1) \\ &= O(nh) = O(n \log n). \end{aligned}$$

3 Segment trees

Consider the set of line segments $L = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5\}$ in the following diagrams:



Observe that we can't always define a total ordering over slanted line segments that preserves the transitivity property. For example, if we consider $\ell_5 \leq \ell_4 \leq \ell_2$ then by transitivity we would have $\ell_5 \leq \ell_2$. On the other hand, if we consider $\ell_2 \leq \ell'_4 \leq \ell_5$ then we would have $\ell_2 \leq \ell_5$. However, if we divide the plane into slabs shown by the dashed lines, we can define a total ordering within each slab that is transitive.

We can build a BST for slabs based on x-values of end points with leaves $S = \{s_{-\infty}, s_1, \ldots, s_9, s_\infty\}$, and internal node v being the union of all slabs within the subtree rooted at v. The root is then all x-axis given by the only slab $[-\infty, \infty]$. Next we augment the BST node v with list of line segments ℓ that crosses any slab with its subtree but do not cross the parent slab.

Invariant Each line segment ℓ is stored in a node of v such that

$$[v.x_{left}, v.x_{right}] \subseteq [\ell.x_{left}, \ell.x_{right}] \ \wedge \ [parent(v).x_{left}, parent(v).x_{right}] \nsubseteq [\ell.x_{left}, \ell.x_{right}].$$

Claim 1. *Each segment will be stored in at most 2 nodes at each level of segment tree.*

Based on the result above, the segment tree would take $O(n \log n)$ space and answer stabbing queries in $O(\log n + k)$.

References