Advanced Data Structures

Amalia Duch

CS-UPC

March 5, 2020

Geometric Problems under consideration

- Vertical Line Stabbing
- Windowing
- Line-Segment Intersection

The vertical line stabbing problem:

INPUT: A collection $I = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n], \}$ of n closed intervals and a query $q \in \mathbb{R}$.

OUTPUT: The list of intervals in I that contain q.

The name of the problem comes from one way to visualize it (drawing!)

Interval Tree

- The idea is to address the problem by a divide and conquer approach.
- Consider the median m of the $2n$ endpoints in I.
- Some intervals will lie entirely to the left of m, some entirely on the right, and some may cross m.
- Those to the left and to the right can be dealt recursively.
- What do we do with the intervals crossing m?

Interval Tree

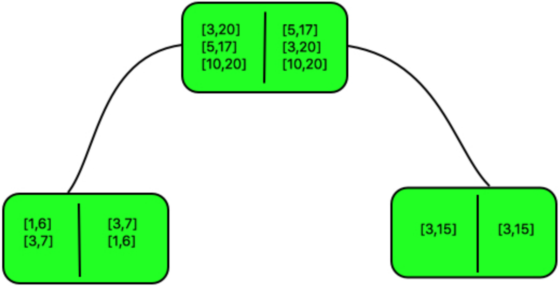
Suppose we have a query $q < m$.

We sort the intervals crossing m by their left endpoints.

With this sorting, the intervals containing q can be listed easily, starting from the leftmost interval and stopping when we reach an interval lying to the right of q.

All the intervals we examine, except for the last, contain q, so the searching process requires time linear in the output size.

If q were on the right of m, instead, we would sort the intervals by their right endpoints.


[3,20]
[5,17]
[10,20]
[5,17]
[3,20]
[10,20]
[1,6]
[3,7]
[3,7]
[1,6]
[3,15]
[3,15]

Interval Trees

- Preprocessing: $O(n \log n)$. Sorting I_m requires $O(n \log n)$ time and the median $O(n)$.
- Storage: $O(n)$. Each interval appears twice in only one node of the tree.
- Query: $O(\log n + K)$. Search through each node requires time linear in the output size $O(K)$, the height $O(\log n)$.

Segment Tree. Construction.

- First: Split the real line into elementary intervals. If $\{c_1, c_2, ..., c_k\}$ is the set of endpoints of I with no repetitions, in increasing order, the elementary intervals are the single points $[c_i, c_i]$ together with the open intervals between them (c_i, c_{i+1}).
- Second: We build a BST over the elementary intervals. Note that each node in the BST can be associated with the union of the elementary intervals belonging to the leaf nodes below it.

Segment Tree. Construction.

- Third: We store in each leaf node of the tree (elementary interval) the set of intervals in I containing that nodes' interval.

Up to here the query time is good: $O(\log n)$ for searching and $O(k)$ for reporting, however the storage can be quadratic.

Segment Tree. Construction.

To improve the storage cost we look to see if any two siblings store the same interval $[a_i, b_i]$. If so, we remove $[a_i, b_i]$ from the two siblings and store it in the parent node, and repeat until no two siblings store the same intervals.
After this process the set of nodes storing any interval will subdivide that interval. This provides a canonical subdivision for each interval in I.

Segment Tree. Querying.

Binary search for q and report all intervals stored in the nodes we visit. Every internal node encountered is reported. The time is $O(\log n + k)$ as with interval trees.

Segment Tree. Storage.

Theorem

Every level of the tree stores any interval at most twice.

Proof.

By contradiction. Suppose that some interval J is stored in at least 3 vertices v_1, v_2, v_3 in some level of a segment tree T. There can not be any gaps between the subtrees rooted at each of these three vertexes since J covers all the real line between them. Therefore some pair v_i, v_{i+1} must be siblings, which is a contradiction. □

The claim shows that the segment tree requires $O(n \log n)$ storage.

Segment Tree. Preprocessing.

To build the BST takes $O(n)$ time. To insert the intervals $O(\log n)$ each, for a total of $O(n \log n)$ time.

Windowing.

Suppose we have a geometric data set containing a huge number of line segments, and want only the portion of this data lying within some window W.

For instance, we may have a state road map and need to a small portion of it for viewing in a GPS.

We need to report all line segments that intersect W.

Windowing

There are two parts of this problem:

A: Finding everything with one or two endpoints inside W and,

B: Finding everything that intersects the border of W twice.

Part A

This is a range query, so we can use any system to solve range queries.

In particular a 2D orthogonal range tree on the end points.

We need to take care if a line segment has 2 endpoints in W to avoid report them twice.

Part B

For simplicity we consider the case of orthogonal line segments (every segment is either orthogonal or vertical). Show figure of the problem.

However we need to get rid of the interval lying above or below W. We will consider two approaches to doing so:

[a] We form an interval tree for the horizontal lines, but store a 2D range tree with their y-values instead of the pair of sorted lists. Queries only require $O(\log^2 n + k)$ but huge storage requirements.

[b] We use a priority search tree described next.

Priority Search Trees

A priority search tree is a mixture of BST and a $Heap$.

Like 2D range trees, these trees represent a 2D point set $p_1, p_2, \ldots, p_n$.

Unlike 2D range trees, the queries of a priority search tree are rectangles with one open side:

$$(-\infty, x_{max}] \times [y_{min}, y_{max}]$$

Priority Search Trees

Construction:

- We store a point with least x coordinate in the root of the tree.
- Let y_m be the median on the remaining points y-coordinates.
- In the left subtree, we store all the points with $y < y_m$ and in the right subtree the remaining points.

Priority Search Trees

Queries:

- We search for y_{min}, y_{max}.
- Intuitively we want to check all points along both search paths to see if they lie inside the query region.
- More precisely, if v_{split} is the point where the two paths diverge, we go from v_{split} to y_{min}, reporting all right subtrees recursively, and stopping if we reach a node with x-coordinate greater than x_{max}.

Preprocessing: $O(n \log n)$ to sort the x and y coordinates.
Storage: $O(n)$. Query: $O(\log n + k)$

The line-segment intersection problem:

INPUT: n line segments in 2D.

OUTPUT: The k intersections.

Obvious Algorithm

For every pair (ℓ, ℓ') of line segments check for their intersection. The cost of this solution is $O(n^2)$.

Sweeping

We are going to solve the problem using the sweep lines technique which is very useful in the solution of several geometric problems.

The running time to determine if there is an intersection among the n segments is $O(n \log n)$.

Listing all the intersections takes time $\Omega(n^2)$ in the worst case.

Assumptions:

- No line segment of the input is vertical.
- No three input segments intersect at the same point.

Sweeping

In sweeping, an imaginary *sweep* line (vertical line) passes through the input set of geometric objects (usually from left to right).

We have to maintain the intersection of the line with the input.

Ordering Segments

Since there are no vertical segments in the input, the segments intersects every (vertical) sweep line at a single point.

Therefore, the segments that intersect a vertical sweep line can be ordered by the y-coordinate of the point of intersection.

Moving the Sweep Line

Sweeping algorithms usually manage two sets of data:

[a] The sweep-line status gives the relationships among the objects that the sweep line intersects at every moment (usually stored in balanced binary tree).

[b] The event-point schedule. This is a sequence of points, which we order from left to right. Whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point and then resumes.

J. L. Bentley.
Decomposable searching problems.
Information Processing Letters, 8(5):133–136, 1979.

J. L. Bentley and J. H. Friedman.
Data structures for range searching.
ACM Computing surveys, 11(4):397–409, 1979.

J.L. Bentley and J. H. Friedman.
Data structures for range searching.
ACM Computing Surveys, 11(4):397–409, 1979.

W. A. Burkhard.
Hashing and trie algorithms for partial match retrieval.
ACM Transactions on Database Systems, 1(2):175–187, 1976.

L. Devroye.

Branching processes in the analysis of the height of trees.
Acta Informatica, 24:277–298, 1987.

- L. Devroye and L. Laforest.
  An analysis of random d-dimensional quadtrees.
  SIAM Journal on Computing, 19(5):821–832, 1990.

- R. A. Finkel and J. L. Bentley.
  Quad trees: a data structure for retrieval on composite key.
  Acta Informatica, 4(1):1–9, 1974.

- Ph. Flajolet, G. Gonnet, C. Puech, and J. M. Robson.
  Analytic variations on quad trees.
  Algorithmica, 10:473–500, 1993.

- Ph. Flajolet and T. Lafforgue.
  Search costs in quad trees and singularity perturbation analysis.

Discrete and Computational Geometry, 12(4):151–175, 1993.

M. Friedman, F. Baskett, and L. J. Shustek.
An algorithm for finding nearest neighbors.
IEEE Transactions on Computing, C–24(10):1000–1006, 1975.

D. E. Knuth.
The Art of Computer Programming: Sorting and Searching, volume 3.
Addison–Wesley, 2nd edition, 1998.

J. Nievergelt, H. Hinterberger, and K. C. Sevcik.
The grid file: An adaptable symmetric multikey file structure.
ACM Transactions on Database Systems, 1(9):38–71, 1984.

M. Regnier.
Analysis of the grid file algorithms.
BIT, 25(2):335–357, 1985.

R. L. Rivest.
Partial-match retrieval algorithms.
SIAM Journal on Computing, 5(1):19–50, 1976.

H. Samet.
Deletion in two-dimensional quad-trees.
Communications of the ACM, 23(12):703–710, 1980.

H. Samet.
The Design and Analysis of Spatial Data Structures.
Addison-Wesley, 1990.