CS603: Geometric Algorithms

Project

BSSR Saran, BVSS Prabandh

Roll_no: 210050036,210050037

APRIL 28 2025

Question 1

Point and Segment Representation

- **Point:** A simple struct with x and y coordinates.
- **Segment:** Stores left and right endpoints (ensuring left is always the smaller x). Includes utility functions like computing the y-coordinate for a given x on the segment and checking if a point lies above the segment.

Node and PNode

The `Node` structure represents a tree node containing a segment and pointers to left and right child nodes. The `PNode` structure is a persistent node storing a vector of (timestamp, Node) pairs, representing versions of the segment tree.

PersistentTree

The `PersistentTree` class manages the persistent binary search tree. It supports:

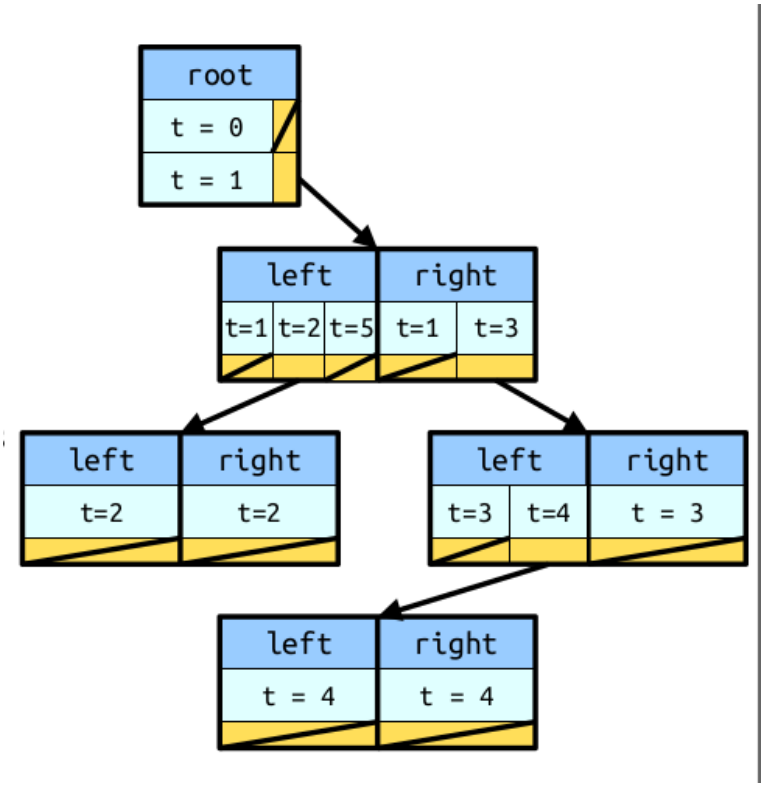
- `insert()` and `delSegment()` to modify the tree at a specific timestamp.
- `createVersion()` to build new tree versions from insertions and deletions.
- `findAbove()` and `findBelow()` to locate the nearest segments relative to a query point.

PointLocation

The `PointLocation` class manages point location queries. It sorts segment endpoints to create x-slabs and builds corresponding versions of the persistent tree. Given a point, it efficiently determines the segments immediately above and below.

- We solve the planar point location problem using a persistent balanced binary search tree and vertical slab decomposition.
- Each segment is stored in the tree as a node, with persistence achieved via Journal node. A new version is created whenever a segment is inserted or deleted at a vertical slab boundary (timestep)

- Slabs are formed using the sorted set of distinct x-coordinates of all segment endpoints. Each slab corresponds to a version of the tree.
- A journal tree is a BST where each field in each node is replaced by a journal describing its values over time.



- At each node in the persistent BST, we perform a binary search over timestamps to find the appropriate version of the node.Each binary search takes $O(\log n)$ time. Since the height of the tree is $O(\log n)$, we perform $O(\log n)$ such binary searches during a query.
- **Total query time:** $O(\log^2 n)$.
- Total of atmost 2n nodes are inserted for 2n edits to the orginal tree. So total **preprocesing time** is $O(n \log n)$ and **space complexity** is $O(n)$
- References: Stanford CS166 Lecture 11

Question 2

We implemented a trapezoidal map data structure for dynamic planar point location using a Directed Acyclic Graph (DAG) as a search structure. The key components of the implementation are as follows:

Point and Segment Representation

- **Point:** A simple struct with x and y coordinates.
- **Segment:** Stores left and right endpoints (ensuring left is always the smaller x). Includes utility functions like computing the y-coordinate for a given x on the segment and checking if a point lies above the segment.

Trapezoid Structure

Each trapezoid is defined by:

- Top and bottom bounding segments.
- Left and right bounding points.
- Pointers to its four neighbors (upper/lower left and right).
- A pointer to the corresponding terminal node in the DAG.

Search DAG Nodes

We use three types of nodes to form the search DAG:

- **XNode:** Binary decision based on the x-coordinate.
- **YNode:** Binary decision based on whether a point is above or below a segment.
- **TerminalNode:** Leaf node representing a trapezoid.

Each node maintains links to left/right children and back-pointers to its parents for efficient replacement and updates.

Graph and DAG Maintenance

- `GraphNode::replaceWith`: Replaces a node in the DAG while preserving parent connections.
- `Trapezoid::changeLeftWith`, `changeRightWith`: Update neighboring trapezoids after splits.

Trapezoid Map Class

The `TrapezoidMap` class maintains:

- A root pointer to the DAG.
- A list of segments added to the structure.
- Functions for building the map, querying trapezoids containing a point, and inserting new segments by splitting existing trapezoids.

Time Complexity

Expected **Query time** is $O(logn)$
Expected **Preprocessing time** is $O(nlogn)$