Geometric Algos Project

Karthikeya Satti (210050142), Hrushikesh (210050097)

Computing All Segment Intersections using Sweep Line Algorithm

Overview

The goal is to compute all pairwise intersections among a set of n line segments in the plane using a sweep line algorithm with AVL-tree-based status structure. The algorithm achieves a time complexity of $O(n \log n + k)$ where k is the number of intersection points.

Key Concepts and Assumptions

- The sweep line moves from top to bottom (descending y-coordinate).
- Events are handled in order of y-coordinate, with intersections processed before insertions, and insertions before removals.
- Each segment is uniquely identified using an integer id.
- A custom AVL tree is used to maintain the sweep line status, where only leaf nodes store segments.
- Internal nodes only store the leftmost segment in their left subtree to support segment comparison and balancing.
- Intersections between segments are detected and inserted as events dynamically during the sweep.
- Point-on-segment intersection points (e.g., T-junctions) are also handled carefully.

Important Functions

segmentLess()

```
bool segmentLess(const Segment* a, const Segment* b) {
    double ax = a->getXatY(sweepY);
    double bx = b->getXatY(sweepY);
    return ax < bx || (fabs(ax - bx) < 1e-9 && a->id > b->id);
}
```

Determines the relative ordering of two segments at the current sweep line level (global `sweepY`).

insert(), erase()

```
AVLNode* insert(AVLNode* node, Segment* s);
AVLNode* erase(AVLNode* node, Segment* s);
```

AVL tree operations customized such that only leaves store actual segments, while internal nodes store metadata to preserve tree structure and facilitate balancing.

above(), below()

```
Segment* above(AVLNode* root, Segment* s);
Segment* below(AVLNode* root, Segment* s);
```

These functions retrieve the segments immediately above and below a given segment in the status structure. These are used to check for new intersections.

getIntersection()

```
bool getIntersection(const Segment* a, const Segment* b, Point& ip);
```

Checks if two segments intersect and calculates the intersection point. Precision is handled using a small epsilon ($1e{-}9$).

addEvent()

```
void addEvent(priority_queue<Event>& events, Segment* a, Segment* b,
              const Point& p, unordered_set<Point, PointHash>& seen);
```

Adds intersection events to the event queue if an intersection is found between two segments and it has not been previously seen.

Horizontal Segment Handling

To handle horizontal segments properly and avoid degeneracies, we nudge their insertion and removal points slightly above and below respectively using a small epsilon (e.g., $1e{-}6$). This prevents them from being skipped or causing ambiguities at shared endpoints.

Sweep Line Status Structure

We use a modified AVL tree to maintain the sweep line status. Only leaf nodes contain actual segments; internal nodes are used to maintain balance and fast lookup. Each internal node stores `maxLeft`, which is the rightmost segment in the left subtree.

Intersection Detection Logic

- On segment insertion, check for intersections with its neighbors (above and below).
- On removal, check whether the neighbors (previously above and below) now intersect with each other.
- On an intersection event, remove and reinsert both involved segments to update their order in the sweep line.

Complexity

- Insertion, deletion, and search in AVL tree take $O(\log n)$ time.
- Each intersection is processed exactly once.
- Total time complexity is $O(n \log n + k)$.

Conclusion

This implementation is a robust and efficient adaptation of the Bentley-Ottmann line sweep algorithm for segment intersection. By customizing an AVL tree to manage only leaf-stored segments and handling precision issues carefully, we ensure correctness in a variety of geometric configurations, including horizontal and collinear cases.

Dynamic Segment Intersection Maintenance

Overview

This extension adapts the sweep line algorithm to support dynamic updates (insertions and deletions) while maintaining $O(n \log n + k)$ time complexity per operation. The implementation handles real-time updates of line segments while efficiently tracking intersection points.

Key Modifications for Dynamic Operation

- **State Preservation**: Maintain AVL tree and event queue between operations
- **Lazy Event Processing**: Process events only when needed for reporting
- **Segment Tracking**: Use unique IDs and active segment set for efficient updates
- **Intersection Invalidation**: Clear stale intersections during deletions

Data Structure Extensions

- **DynamicIntersectionTracker** class encapsulates:
  - Active segments set
  - Persistent event queue avlRoot - Current sweep line status
  - Intersection cache with invalidation
- Enhanced AVL tree operations with neighbor tracking

Core Operations

Insert Segment

```
int insertSegment(Point p, Point q) {
int id = segments.size();
segments.emplace_back(make_unique&lt;Segment&gt;(p, q, id));
activeSegments.insert(id);
// Add events with epsilon adjustment for horizontals
eventQueue.push({seg->p, 0, seg, nullptr});
eventQueue.push({seg->q, 1, seg, nullptr});
processEvents();
```

```
return id;
}
```

Delete Segment

```
bool deleteSegment(int segmentId) {
activeSegments.erase(segmentId);
// Filter event queue
priority_queue<Event> newQueue;
while (!eventQueue.empty()) {
Event e = eventQueue.top();
if (e.s1 != seg && (!e.s2 || e.s2 != seg))
newQueue.push(e);
}
// Update AVL tree and reprocess
avlRoot = erase(avlRoot, seg);
processEvents();
}
```

Dynamic Event Handling

- **Incremental Processing**: Only process events affected by updates
- **Intersection Cache**: Maintain valid intersections between operations
- **Neighbor Revalidation**: After deletion, check previous neighbors for new intersections

Complexity Analysis

- **Insertion**: $O(\log n + k')$ where k' = new intersections
- **Deletion**: $O(\log n + m')$ where m' = affected intersections
- **Reporting**: $O(1)$ access to cached results
- **Space**: $O(n + k)$ for segments and intersections

Special Case Handling

- **Concurrent Modifications**: Handle overlapping operations through event queue versioning
- **Segment Lifespan**: Track active/inactive state using IDs
- **Cascading Invalidations**: Remove dependent intersections when parent segments are deleted

Conclusion

This dynamic extension maintains the original algorithm's efficiency while enabling real-time updates. Key innovations include:

- Stateful event processing with lazy evaluation
- Efficient neighbor tracking during deletions
- Robust invalidation mechanisms for intersection cache