6.3 Geometric Search

- range search
- space partitioning trees
- intersection search

Overview

Geometric objects. Points, lines, intervals, circles, rectangles, polygons, ...

This lecture. Intersection among N objects.

Example problems.
- 1D range search.
- 2D range search.
- Find all intersections among h-v line segments.
- Find all intersections among h-v rectangles.

Example problems.
- 1D range search.
- 2D range search.
- Find all intersections among h-v line segments.
- Find all intersections among h-v rectangles.

1d range search

Extension of ordered symbol table.
- Insert key-value pair.
- Search for key k.
- Rank: how many keys less than k?
- Range search: find all keys between k₁ and k₂.

Application. Database queries.

Geometric interpretation.
- Keys are point on a line.
- How many points in a given interval?
1d range search: implementations

Ordered array. Slow insert, binary search for \( lo \) and \( hi \) to find range.

Hash table. No reasonable algorithm (key order lost in hash).

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>rank</th>
<th>range count</th>
<th>range search</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
<tr>
<td>hash table</td>
<td>1</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>BST</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
</tbody>
</table>

\( N \) = # keys
\( R \) = # keys that match

BST. All operations fast.

1d range search: BST implementation

Range search. Find all keys between \( lo \) and \( hi \)?
- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

Worst-case running time. \( R + \log N \) (assuming BST is balanced).

2d orthogonal range search

Extension of ordered symbol-table to 2d keys.
- Insert a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range?

Applications. Networking, circuit design, databases.

Geometric interpretation.
- Keys are point in the plane.
- How many points in a given h-v rectangle.

Grid implementation.
- Divide space into \( M \)-by-\( M \) grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add \( (x, y) \) to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.
2d orthogonal range search: grid implementation costs

Space-time tradeoff.
- Space: \( M^2 + N \).
- Time: \( 1 + N / M^2 \) per square examined, on average.

Choose grid square size to tune performance.
- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: \( \sqrt{N} \)-by-\( \sqrt{N} \) grid.

Running time. [if points are evenly distributed]
- Initialize: \( O(N) \).
- Insert: \( O(1) \).
- Range: \( O(1) \) per point in range.

Clustering

Grid implementation. Fast, simple solution for well-distributed points.
Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.

- 13,000 points, 1000 grid squares
  - Half the squares are empty
  - Half the points are in 10% of the squares
Space-partitioning trees

Use a tree to represent a recursive subdivision of 2D space.

- **Quadtree.** Recursively divide space into four quadrants.
- **2d tree.** Recursively divide space into two halfplanes.
- **BSP tree.** Recursively divide space into two regions.

Quadtree

**Idea.** Recursively divide space into 4 quadrants.

**Implementation.** 4-way tree (actually a trie).

**Benefit.** Good performance in the presence of clustering.

**Drawback.** Arbitrary depth!

Space-partitioning trees: applications

**Applications.**

- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.

Quadtree: larger example

Quadtree: 2D range search

**Range search.** Find all keys in a given 2D range.
- Recursively find all keys in NE quad (if any could fall in range).
- Recursively find all keys in NW quad (if any could fall in range).
- Recursively find all keys in SE quad (if any could fall in range).
- Recursively find all keys in SW quad (if any could fall in range).

**Typical running time.** $O(R \times \log N)$.

N-body simulation

**Goal.** Simulate the motion of $N$ particles, mutually affected by gravity.

**Brute force.** For each pair of particles, compute force.

$$F = \frac{G m_1 m_2}{r^2}$$

Subquadratic N-body simulation

**Key idea.** Suppose particle is far, far away from cluster of particles.
- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and center of mass of aggregate particle.

Barnes-Hut algorithm for N-body simulation.

**Barnes-Hut.**
- Build quadtree with $N$ particles as external nodes.
- Store center-of-mass of subtree in each internal node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to quad is sufficiently large.
Curse of dimensionality

Range search / nearest neighbor in k dimensions?
Main application: Multi-dimensional databases.

3d space. Octrees: recursively divide 3d space into 8 octants.
100d space. Centrees: recursively divide 100d space into $2^{100}$ centra???

Raytracing with octrees

2d tree

Recursively partition plane into two halfplanes.

2d tree

Implementation. BST, but alternate using x- and y-coordinates as key.
• Search gives rectangle containing point.
• Insert further subdivides the plane.

Range search. Find all points in a query axis-aligned rectangle.
• Check if point in node lies in given rectangle.
• Recursively search left/top subdivision (if any could fall in rectangle).
• Recursively search right/bottom subdivision (if any could fall in rectangle).

Typical case. $R + \log N$
Worst case (assuming tree is balanced). $R + \sqrt{N}$. 
2d tree: nearest neighbor search

Nearest neighbor search. Given a query point, find the closest point.
- Check distance from point in node to query point.
- Recursively search left/top subdivision (if it could contain a closer point).
- Recursively search right/bottom subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Typical case. $\log N$
Worst case (even if tree is balanced). $N$

Kd tree

Kd tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.

Efficient, simple data structure for processing k-dimensional data.
- Widely used.
- Discovered by an undergrad in an algorithms class!
- Adapts well to high-dimensional and clustered data.

Search for intersections

Problem. Find all intersecting pairs among N geometric objects.
Applications. CAD, games, movies, virtual reality.

Simple version. 2D, all objects are horizontal or vertical line segments.

Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.
**Orthogonal segment intersection search: sweep-line algorithm**

Sweep vertical line from left to right.
- x-coordinates define events.
- Left endpoint of h-segment: insert y-coordinate into ST.
- Right endpoint of h-segment: remove y-coordinate from ST.

**Remark.** Sweep-line solution extends to 3D and more general shapes.
**Immutable h-v segment data type**

```java
public final class SegmentHV implements Comparable<SegmentHV> {
    public final int x1, y1;
    public final int x2, y2;

    public SegmentHV(int x1, int y1, int x2, int y2) {
        ...    
    }

    public boolean isHorizontal() {
        ...    
    }

    public boolean isVertical() {
        ...    
    }

    public int compareTo(SegmentHV that) {
        ...    
    }
}
```

**Sweep-line event subclass**

```java
private class Event implements Comparable<Event> {
    private int time;
    private SegmentHV segment;

    public Event(int time, SegmentHV segment) {
        this.time = time;
        this.segment = segment;
    }

    public int compareTo(Event that) {
        return this.time - that.time;
    }
}
```

**Sweep-line algorithm: initialize events**

```java
MinPQ<Event> pq = new MinPQ<Event>();
for (int i = 0; i < N; i++) {
    if (segments[i].isVertical()) {
        Event e = new Event(segments[i].x1, segments[i].y);
        pq.insert(e);
    }
    else if (segments[i].isHorizontal()) {
        Event e1 = new Event(segments[i].x1, segments[i].y);
        Event e2 = new Event(segments[i].x2, segments[i].y);
        pq.insert(e1);
        pq.insert(e2);
    }
}
```

**Sweep-line algorithm: simulate the sweep line**

```java
int INF = Integer.MAX_VALUE;
SET<SegmentHV> set = new SET<SegmentHV>();
while (!pq.isEmpty()) {
    Event event = pq.delMin();
    int sweep = event.time;
    SegmentHV segment = event.segment;
    if (segment.isVertical()) {
        SegmentHV seg1, seg2;
        seg1 = new SegmentHV(-INF, segment.y1, -INF, segment.y1);
        seg2 = new SegmentHV(+INF, segment.y2, +INF, segment.y2);
        for (SegmentHV seg : set.range(seg1, seg2))
            StdOut.println(segment + " intersects " + seg);
    }
    else if (sweep == segment.x1) set.add(segment);
    else if (sweep == segment.x2) set.remove(segment);
}
```
General line segment intersection search

Extend sweep-line algorithm
• Maintain segments that intersect sweep line ordered by y-coordinate.
• Intersections can only occur between adjacent segments.
• Add/delete line segment ⇒ one new pair of adjacent segments.
• Intersection ⇒ swap adjacent segments.

Line segment intersection: implementation

Efficient implementation of sweep line algorithm.
• Maintain PQ of important x-coordinates: endpoints and intersections.
• Maintain set of segments intersecting sweep line, sorted by y.
• \( O(R \log N + N \log N) \).

Implementation issues.
• Degeneracy.
• Floating point precision.
• Use PQ, not presort (intersection events are unknown ahead of time).

Rectangle intersection search

Goal. Find all intersections among h-v rectangles.

Application. Design-rule checking in VLSI circuits.

Microprocessors and geometry

Early 1970s. microprocessor design became a geometric problem.
• Very Large Scale Integration (VLSI).
• Computer-Aided Design (CAD).

Design-rule checking.
• Certain wires cannot intersect.
• Certain spacing needed between different types of wires.
• Debugging = rectangle intersection search.
Algorithms and Moore’s law

“Moore’s law.” Processing power doubles every 18 months.
- 197x: need to check N rectangles.
- 197(x+1.5): need to check 2N rectangles on a 2x-faster computer.

Bootstrapping. We get to use the faster computer for bigger circuits.

But bootstrapping is not enough if using a quadratic algorithm:
- 197x: takes M days.
- 197(x+1.5): takes (4M)/2 = 2M days. (!)

Bottom line. Linearithmic CAD algorithm is necessary to sustain Moore’s Law.

Interval search trees

Rectangle intersection search

Sweep vertical line from left to right.
- x-coordinates of rectangles define events.
- Maintain set of y-intervals intersecting sweep line.
- Left endpoint: search set for y-interval; insert y-interval.
- Right endpoint: delete y-interval.

Reduces 2D orthogonal rectangle intersection search to 1D interval search!

Running time of sweep line algorithm.
- Put x-coordinates on a PQ (or sort). $O(N \log N)$
- Insert y-interval into ST. $O(N \log N)$
- Delete y-interval from ST. $O(N \log N)$
- Interval search. $O(R + N \log N)$

Efficiency relies on judicious use of data structures.
# Geometric search summary: algorithms of the day

<table>
<thead>
<tr>
<th>1D range search</th>
<th>45</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>kD range search</td>
<td></td>
<td>kD tree</td>
</tr>
<tr>
<td>1D interval intersection search</td>
<td>interval search tree</td>
<td></td>
</tr>
<tr>
<td>2D orthogonal line intersection search</td>
<td>sweep line reduces to 1D range search</td>
<td></td>
</tr>
<tr>
<td>2D orthogonal rectangle intersection search</td>
<td>sweep line reduces to 1D interval intersection search</td>
<td></td>
</tr>
</tbody>
</table>