Geometric Algorithms

**Geometric search: Overview**

**Types of data.** Points, lines, planes, polygons, circles, ...

**This lecture.** Sets of N objects.

**Geometric problems extend to higher dimensions.**
- Good algorithms also extend to higher dimensions.
- Curse of dimensionality.

**Basic problems.**
- Range searching.
- Nearest neighbor.
- Finding intersections of geometric objects.
7.3 Range Searching
1D Range Search

Extension to symbol-table ADT with comparable keys.

- Insert key-value pair.
- Search for key k.
- How many records have keys between $k_1$ and $k_2$?
- Iterate over all records with keys between $k_1$ and $k_2$.

Application: database queries.

Geometric intuition.
- Keys are point on a line.
- How many points in a given interval?

```
insert B   B
insert D   B D
insert A   A B D
insert I   A B D I
insert H   A B D H I
insert F   A B D F H I
insert P   A B D F H I P
count G to K 2
search G to K H I
```
1D Range Search Implementations

**Range search.** How many records have keys between $k_1$ and $k_2$?

**Ordered array.** Slow insert, binary search for $k_1$ and $k_2$ to find range.

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

**BST.** In each node $x$, maintain number of nodes in tree rooted at $x$. Search for smallest element $\geq k_1$ and largest element $\leq k_2$.

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>count</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>$N$</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>
</tr>
<tr>
<td>BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$R + \log N$</td>
</tr>
</tbody>
</table>

$N = \#$ records
$R = \#$ records that match
2D Orthogonal Range Search

Extension to symbol-table ADT with 2D keys.
- Insert a 2D key.
- Search for a 2D key.
- Range search: find all keys that lie in a 2D range?
- Range count: how many keys lie in a 2D range?

Applications: networking, circuit design, databases.

Geometric interpretation.
- Keys are point in the plane.
- Find all points in a given h-v rectangle?
2D Orthogonal Range Search: Grid Implementation

**Grid implementation.** [Sedgewick 3.18]

- Divide space into $M$-by-$M$ grid of squares.
- Create linked list for each square.
- Use 2D array to directly access relevant square.
- Insert: insert $(x, y)$ into corresponding grid square.
- Range search: examine only those grid squares that could have points in the rectangle.
Space-time tradeoff.
- Space: $M^2 + N$.
- Time: $1 + N / M^2$ per grid cell examined on average.

Choose grid square size to tune performance.
- Too small: wastes space.
- Too large: too many points per grid 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 is a well-known phenomenon in geometric data.

Ex: USA map data.
- 80,000 points, 20,000 grid squares.
- Half the grid squares are empty.
- Half the points have ≥ 10 others in same grid square.
- Ten percent have ≥ 100 others in same grid square.

Need data structure that *gracefully* adapts to data.
Space Partitioning Trees

**Space partitioning tree.** Use a tree to represent the recursive hierarchical subdivision of d-dimensional space.

**BSP tree.** Recursively divide space into two regions.

**Quadtrees.** Recursively divide plane into four quadrants.

**Octrees.** Recursively divide 3D space into eight octants.

**kD trees.** Recursively divide k-dimensional space into two half-spaces.

**Applications.**
- Ray tracing.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.
Quad Trees

**Quad tree.** Recursively partition plane into 4 quadrants.

**Implementation:** 4-way tree.

```
public class QuadTree {
    private Quad quad;
    private Value value;
    private QuadTree NW, NE, SW, SE;
}
```

Good clustering performance is a primary reason to choose quad trees over grid methods.
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 into $2^{100}$ centrants?

Raytracing with octrees
2D Trees

2D tree. Recursively partition plane into 2 halfplanes.

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

**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.

- Adapts well to clustered data.
- Adapts well to high dimensional data.
- Discovered by an undergrad in an algorithms class!
Summary

Basis of many geometric algorithms: search in a planar subdivision.

<table>
<thead>
<tr>
<th></th>
<th>grid</th>
<th>2D tree</th>
<th>Voronoi diagram</th>
<th>intersecting lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>basis</td>
<td>$\sqrt{N}$ h-v lines</td>
<td>N points</td>
<td>N points</td>
<td>$\sqrt{N}$ lines</td>
</tr>
<tr>
<td>representation</td>
<td>2D array of N lists</td>
<td>N-node BST</td>
<td>N-node multilist</td>
<td>$\sim$N-node BST</td>
</tr>
<tr>
<td>cells</td>
<td>$\sim$N squares</td>
<td>N rectangles</td>
<td>N polygons</td>
<td>$\sim$N triangles</td>
</tr>
<tr>
<td>search cost</td>
<td>1</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>extend to kD?</td>
<td>too many cells</td>
<td>easy</td>
<td>cells too complicated</td>
<td>use (k-1)D hyperplane</td>
</tr>
</tbody>
</table>
7.4 Geometric Intersection
**Geometric Intersection**

**Problem.** Find all intersecting pairs among set of \( N \) 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.

**Sweep line.** Efficient solution extends to 3D and general objects.
Orthogonal Segment Intersection: Sweep Line Algorithm

Sweep vertical line from left to right.

- Event times: x-coordinates of h-v line segments.
- Left endpoint of h-segment: insert y coordinate into ST.
- Right endpoint of h-segment: remove y coordinate from ST.
- v-segment: range search for interval of y endpoints.
Orthogonal Segment Intersection: Sweep Line Algorithm

**Sweep line:** reduces 2D orthogonal segment intersection problem to 1D range searching!

**Running time of sweep line algorithm.**
- Put x-coordinates on a PQ (or sort). \( O(N \log N) \)
- Insert y-coordinate into SET. \( O(N \log N) \)
- Delete y-coordinate from SET. \( O(N \log N) \)
- Range search. \( O(R + N \log N) \)

Efficiency relies on judicious use of data structures.
Immutable H-V Segment ADT

```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 b) { ... }
    public String toString() { ... }
}
```

compare by y-coordinate;
break ties by x-coordinate

(x1, y1)   (x2, y1)
horizontal segment

(x1, y1)   (x1, y2)
vertical segment
public class Event implements Comparable<Event> {
    int time;
    SegmentHV segment;

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

    public int compareTo(Event b) {
        return a.time - b.time;
    }
}
// initialize events
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]);
        pq.insert(e);
    }
    else if (segments[i].isHorizontal()) {
        Event e1 = new Event(segments[i].x1, segments[i]);
        Event e2 = new Event(segments[i].x2, segments[i]);
        pq.insert(e1);
        pq.insert(e2);
    }
}
// simulate the sweep line
int INF = Integer.MAX_VALUE;
SET<SegmentHV> set = new SET<SegmentHV>();
while (!pq.isEmpty()) {
    Event e = pq.delMin();
    int sweep = e.time;
    SegmentHV segment = e.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))
            System.out.println(segment + " intersects " + seg);
    }

    else if (sweep == segment.x1) set.add(segment);
    else if (sweep == segment.x2) set.remove(segment);
}
Use horizontal sweep line moving from left to right.

- Maintain order of segments that intersect sweep line by y-coordinate.
- Intersections can only occur between adjacent segments.
- Add/delete line segment ⇒ one new pair of adjacent segments.
- Intersection ⇒ two new pairs of adjacent segments.
Line Segment Intersection: Implementation

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

Implementation issues.
- Degeneracy.
- Floating point precision.
- Use PQ since intersection events aren't known ahead of time.
VLSI Rules Checking
Rectangle intersection. Find all intersections among h-v rectangles.
Application. VLSI rules checking in microprocessor design.

Early 1970s: microprocessor design became a geometric problem.
  - Very Large Scale Integration (VLSI).
  - Computer-Aided Design (CAD).
  - Design-rule checking.
"Moore's Law." Processing power doubles every 18 months.

- 197\(x\): need to check \(N\) rectangles.
- 197\((x+1.5)\): need to check \(2N\) rectangles on a \(2^{x}\)-faster computer.

**Quadratic algorithm.** Compare each rectangle against all others.
- 197\(x\): takes \(M\) days.
- 197\((x+1.5)\): takes \((4^M)/2 = 2^M\) days. (!)

**Need** \(O(N \log N)\) CAD algorithms to sustain Moore's Law.
VLSI Database Problem

Move a vertical "sweep line" from left to right.

- **Sweep line:** sort rectangles by x-coordinate and process in this order, stopping on left and right endpoints.
- **Maintain set of intervals** intersecting sweep line.
- **Key operation:** given a new interval, does it intersect one in the set?
Support following operations.

- **Insert** an interval \((lo, hi)\).
- **Delete** the interval \((lo, hi)\).
- **Search** for an interval that intersects \((lo, hi)\).

**Non-degeneracy assumption.** No intervals have the same x-coordinate.
Interval tree implementation with BST.

- Each BST node stores one interval.
- BST nodes sorted on lower endpoint.
Interval tree implementation with BST.
- Each BST node stores one interval.
- BST nodes sorted on lower endpoint.
- Additional info: store and maintain max endpoint in subtree rooted at node.
Finding an Intersecting Interval

Search for an interval that intersects (lo, hi).

Node x = root;
while (x != null) {
    if (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null) x = x.right;
    else if (x.left.max < lo) x = x.right;
    else x = x.left;
}
return null;

Case 1. If search goes right, no overlap in left.

1. (x.left == null) ⇒ trivial.
2. (x.left.max < lo) ⇒ for any interval (a, b) in left subtree of x, we have b ≤ max < lo.

\[
\begin{align*}
\text{defn of max} & \quad \uparrow \\
\text{reason for going right} & \quad \uparrow \\
\end{align*}
\]

(a, b)  (lo, hi)
left subtree of x
Finding an Intersecting Interval

Search for an interval that intersects \((lo, hi)\).

```java
Node x = root;
while (x != null) {
    if (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null) x = x.right;
    else if (x.left.max < lo) x = x.right;
    else x = x.left;
}
return null;
```

Case 2. If search goes left, then either (i) there is an intersection in left subtree or (ii) no intersections in either subtree.

Pf. Suppose no intersection in left. Then for any interval \((a, b)\) in right subtree, \(a \geq c > hi \Rightarrow no intersection in right.\)
## Interval Search Tree: Analysis

### Implementation

Use a balanced BST to guarantee performance.

- can maintain auxiliary information using \( \log N \) extra work per op

### Operation Analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert interval</td>
<td>( \log N )</td>
</tr>
<tr>
<td>delete interval</td>
<td>( \log N )</td>
</tr>
<tr>
<td>find an interval that intersects ((lo, hi))</td>
<td>( \log N )</td>
</tr>
<tr>
<td>find all intervals that intersect ((lo, hi))</td>
<td>( R \log N )</td>
</tr>
</tbody>
</table>

\( N = \# \) intervals
\( R = \# \) intersections
Move a vertical "sweep line" from left to right.

- **Sweep line**: sort rectangles by x-coordinates and process in this order.
- **Store set of rectangles that intersect the sweep line in an interval search tree (using y-interval of rectangle).**
- **Left side**: interval search for y-interval of rectangle, insert y-interval.
- **Right side**: delete y-interval.
VLSI Database Problem: Sweep Line Algorithm

Sweep line: reduces 2D orthogonal rectangle intersection problem to 1D interval searching!

Running time of sweep line algorithm.

- Sort by x-coordinate. \( O(N \log N) \)
- Insert y-interval into ST. \( O(N \log N) \)
- Delete y-interval from ST. \( O(N \log N) \)
- Interval search. \( O(R \log N) \)

Efficiency relies on judicious extension of BST.

\( N = \# \text{ line segments} \)
\( R = \# \text{ intersections} \)